ESCUELA SUPERIOR POLITÉCNICA DEL LITORAL

Facultad de Ingeniería en Electricidad y Computación

"DISEÑO Y EVALUACIÓN DE ALGORITMO PARA CLASIFICACIÓN DE ACTIVIDADES HUMANAS UTILIZANDO REDES NEURONALES RECURRENTES"

INFORME DE PROYECTO INTEGRADOR

Previo a la obtención del Título de:

INGENIERO EN COMPUTACIÓN

JUAN JOSÉ GARCÍA CEDEÑO

GUAYAQUIL – ECUADOR

AÑO: 2018

AGRADECIMIENTOS

$\forall e \in U$

DEDICATORIA

Para mi familia, este es el producto de su amor.

TRIBUNAL DE EVALUACIÓN

Ph.D. Sixto García	**Ph.D. Daniel Ochoa**
PROFESOR TUTOR	PROFESOR COLABORADOR

DECLARACIÓN EXPRESA

"La responsabilidad y la autoría del contenido de este Trabajo de Titulación, me corresponde exclusivamente; y doy mi consentimiento para que la ESPOL realice la comunicación pública de la obra por cualquier medio con el fin de promover la consulta, difusión y uso público de la producción intelectual"

.....

Juan José García Cedeño

RESUMEN

Las redes neuronales son modelos matemáticos con una capacidad de generalización extraordinaria y misteriosa.

Son capaces de resolver tareas complejas como reconocer actividades en un video, clasificar imágenes, o traducir de un lenguae a otro. Sin embargo, el fundamento de los modelos suele ser empírico y no matemático.

Este proyecto se desvia de la tendencia y propone una red neuronal recurrente para reconocer actividades en un video; pero justificada en intuiciones matemáticas.

La arquitectura propuesta (denominada D-RNN) logra resultados de clasificación comparables a una LSTM (arquitectura recurrente más común para reconocer actividades), pero con 0.68% de los parámetros. Convirtiendose en una alternativa viable cuando el porcentaje de error puede sacrificarse por un modelo más liviano.

Adicionalmente, se desmitifica a las redes neuronales y a TensorFlow. El capítulo dos y los anexos elaboran en los temas respectivamente, y cada uno construye una base teórica para sustentar las ideas que se plantean.

En resumen, el capítulo uno da una visión general del problema a resolver, el capítulo dos explica las redes neuronales y las redes neuronales recurrentes, el capítulo tres explica la solución propuesta, y el capítulo cuatro analiza los resultados. En los anexos se explica TensorFlow.

Palabras Clave: RNN, gradientes, desvanecen, retardo, actividad, reconocimiento.

ABSTRACT

Neural networks are machine learning models with extraordinary generalization capabilities.

These models have proven best when solving complex tasks, such as: human activity recognition, machine translation and text analysis. Nevertheless, architectures that solve these complex tasks are often based on empirical evidence rather than mathematical intuition.

This project aims to design a recurrent neural network to do human activity recognition, but using mathematical guidelines to justify the design choices. The end result is an architecture that performs comparable to the most popular recurrent architecture (i.e. LSTMs) with only 0,68% of the parameters.

Chapter one provides an overrall notion of the problem. Chapter two elaborates on the theoretical foundation to understand neural networks and recurrent neural networks. Chapter three proposes a way to mitigate the vanishing gradient problem of recurrent neural networks. Finally, in chapter four, the architecture proposed is tested to do human activity recognition.

Aditionally, this project aims to demistify neural networks and TensorFlow, elaborated on chapter two and the annexes, respectively.

Keywords: RNN, gradients, vanishing, delay, activity, recognition

ÍNDICE GENERAL

ABREVIATURAS

ANN Artificial Neural Networks

RNN Recurrent Neural Networks

LSTM Long-Short Term Memory

D-RNN Delayed Recurrent Neural Network

SIMBOLOGÍA

$\mathbb{R}$	Conjunto de los reales
$f: A \to B$	Función con dominio A y rango B
a	Escalar
$\boldsymbol{a}$	Vector
A	Matriz
$\frac{\partial y}{\partial \boldsymbol{x}}$	Derivada de un escalar con respecto a un vector
$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}}$	Derivada de un vector con respecto a un vector (Matriz Jacobiana)
$\frac{\partial y}{\partial X}$	Derivada de un escalar con respecto a una matriz
$\nabla_x y$	Gradiente de y con respecto a x
$\mathrm{x} \sim P(X)$	Variable aleatoria x es generada por una distribución $P(X)$

ÍNDICE FIGURAS

ÍNDICE TABLAS

CAPITULO 1

1. INTRODUCCIÓN

1.1 Definición del problema

Las redes neuronales recurrentes, son una variación de las redes neuronales, diseñadas para procesar información secuencial.

Su característica es la unidad recurrente, un parámetro que conecta la salida de la red con todas las entradas procesadas hasta el momento.

A pesar de ser un modelo teóricamente más fuerte que las redes neuronales, no son igual de populares. Esto se debe, en parte, a que la misma unidad recurrente que las caracteriza, inestabiliza la decisión de la red. Es decir, hace a la salida poco dependiente o muy dependiente a una entrada.

El problema causa que la red solo considere entradas que están muy cerca de a salida, o muy lejos. Reduciendo la capacidad del modelo para resolver problemas donde la clasificación deba considerar entradas temporalmente distantes (e.g. reconocimiento de actividades).

1.2 Justificación

Resolver el problema tiene varias ventajas:

Primero, permite al modelo ser capaz de considerar más información al tomar su decisión. Consecuentemente, mejorando la capacidad para modelar sistemas con dependencias temporales extensas

Segundo, la recurrencia en una red neuronal recurrente se puede comparar con una red neuronal de muchas capas (i.e. muchas transformaciones). La generalización de una solución podria impactar no solo modelos recurrentes, sino también modelos de deep learning en general.

Tercero, actualmente existe una carrera por encontrar una base matemática que fundamente a las redes neuronales. Esta es una oportunidad de indagar teoricamente en uno de los problemas principales de las redes neuronales: El aprendizaje.

Cuarto, el problema de la inestabilidad también enlentece el entrenamiento de la red neuronal recurrente. Una solución no solo puede mejorar la capacidad del modelo sino también acelerar el entrenamiento.

1.3 Soluciones anteriores

La arquitectura más utilizada para mitigar el problema se conoce como LSTM (Long short term memory unit). Esta arquitectura añade trece operaciones más al modelo recurrente básico, con el fin de modelar el flujo de la información. Es decir, los parámetros modelan mecanismos de decisión, para conservar o eliminar, la entrada, salida, o estado de la red neuronal recurrente. Las LSTM son parte de un grupo de unidades recurrentes llamados *Gated Recurrent Units* [2].

1.4 Objetivos

1.4.1 Objetivo General

- Reconocer actividades humanas utilizando redes neuronal recurrentes.

1.4.2 Objetivos específicos

- Diseñar una red neuronal recurrente con justificación teórica.
- Alcanzar resultados de clasificación comparables a una LSTM.
- Proponer un modelo más simple que una LSTM.

1.5 Metodología

Para evaluar el modelo se cuantifica su rendimiento al resolver una tarea con dependencias temporales extensas, y se lo compara con el modelo recurrente base, y la arquitectura más popular (i.e. LSTM) en el mismo contexto. Esta comparación valida que el modelo sea una solución viable antes de indagar más en el.

La tarea que modelaremos es el reconocimiento de actividades en un video. Esto consiste en clasificar, las entrada que caracteriza a la actividad (e.g. posición en 3D de las junturas del cuerpo), en una de varias actividades.

Las métricas de evaluación seran el porcentaje de clasificaciones correctas, velocidad de convergencia, y el número de parámetros.

1.6 Solución propuesta

El problema de la inestabilidad en la decisión es causado por el número de transformaciones entre la entrada y la salida. La solución consiste en introducir diferentes unidades recurrentes, que procesan sus entradas a diferentes escalas de tiempo; reduciendo asi el número de tranformaciones entre las entradas y la salida.

1.7 Alcance

El problema de la inestabilidad en la solución tiene dos aspectos:

1. La dependencia de la salida con respecto a las entradas tiende a explotar.
2. La dependencia de la salida con respecto a las entradas tiende a desvanecerse.

El proyecto mitiga teoricamente el problema de los gradientes que se desvanecen (*vanishing gradients*), propone una red neuronal con justificación teórica, y valida que la idea puede resolver tareas del mundo real (e.g. reconocimiento de actividades).

Todavía no se cuantifica empiricamente el el impacto de la arquitectura en el gradiente. Además, existen otras formas de mitigar el problema de los gradientes que se desvanecen (e.g. uso de regularizadores, diferentes esquemas de optimización, NARX RNN, echo state networks). De las mencionadas, las arquitecturas NARX RNN [12] y la propuesta en [11] son las que guardan mayor similitud con la planteada en este proyecto. La comparación con estas arquitecturas se plantea como trabajos futuros.

En el siguiente capítulo se explica la fundamentación teórica para entender a las redes neuronales, a las redes neuronales recurrentes y se introduce el problema de las arquitecturas recurrentes.

CAPÍTULO 2

2. DESMITIFICANDO A LAS REDES NEURONALES.

Las redes neuronales son modelos de *machine learning*, que han logrado los mejores resultados en tareas como: reconocimiento de objetos, traducción, y reconocimiento de actividades.

El resurgimiento de las redes neuronales se debe a un incremento en el poder computacional, algoritmos de optimización más eficientes, nuevas arquitecturas de redes neuronales, *datasets* más grandes, y a un desarrollo de heurísticas con fundamento empírico. No obstante, todavía no se ha definido una base teórica que explique o garantice, de forma rigurosa, el comportamiento de una red neuronal.

La falta de fundamentación teórica se puede atribuir, en parte, a la complejidad de las soluciones propuestas. Arquitecturas complejas mejoran la capacidad de generalizar (algo favorable desde un punto de vista pragmático). Pero como resultado, se acompleja el modelo, su optimización, y consecuentemente, su justificación teórica.

El objetivo del proyecto es proponer una arquitectura para el reconocimiento de actividades, pero justificando su diseño teoricamente.

Antes de proponer la solución, se elabora una estructura teórica que explique las redes neuronales y las redes neuronales recurrentes. Finalmente, aprovechamos la estructura teórica para presentar el problema de las redes neuronales recurrentes.

2.1 Redes neuronales artificiales.

Las redes neuronales artificiales simulan el comportamiento de las neuronas del cerebro, con el fin de recrear su poder representativo. Se espera que un modelo matemático con la misma estructura sea capaz modelar soluciones complejas de programar (e.g. clasificar, traducir, predecir, transcribir, sintetizar y muestrear, detectar, limpiar, y estimar).

Como clasificador, la red neuronal asigna el vector de entrada a un número discreto y predefinido de clases. Es decir, la red recibe una muestra $\boldsymbol{x} \in \mathbb{R}^n$, aplica la función $f(\boldsymbol{x}, \boldsymbol{\theta})$, y la asigna a una de p clases. La salida es un vector en $\mathbb{R}^p$, con un valor de uno en el elemento que corresponde a la clase, y cero para el resto (Esto es una forma de codificar la salida, y se conoce como one-hot encoding).

En forma general, la red neuronal es una función de las entradas de la red y los parámetros de la red (i.e. $\boldsymbol{\theta}$). Esta función transforma un vector de $\mathbb{R}^n$ a $\mathbb{R}^p$, donde n es el tamaño del vector de entrada y p el número de clases a clasificar.

$$f(\boldsymbol{x}, \boldsymbol{\theta}): \mathbb{R}^n \rightarrow \mathbb{R}^p \tag{2.1}$$

Una red neuronal, $f(\boldsymbol{x}, \boldsymbol{\theta})$, se compone de tres capas: una capa de entrada, una capa de neuronas (a la que nos referiremos como *hidden layer*) y una capa de salida. A excepción de la primera, cada capa está representada con una transformación.

La capa de entrada consiste en una representación de la entradas de la red. Dependiendo de la arquitectura de la red, esta capa puede ser un tensor de rango {0, …, n}. Este proyecto trabaja con tensores de rango uno (i.e. vectores) como entrada.

El hidden layer consta de varias neuronas. Cada neurona procesa señales de entrada y se "activa" en función del procesamiento. La ecuación (2.2) modela una neurona, donde x es un vector que representa las diferentes entradas de la red, u es un vector que representa los pesos asignados a cada entrada, y b es un escalar que limita la activación de la neurona. ϕ representa la función de activación, una función monotónica creciente (e.g. un escalón, sigmoide, o tangente hiperbólica).

$$\boldsymbol{h} = \phi(\boldsymbol{u}^T \boldsymbol{x} + \boldsymbol{b}) \tag{2.2}$$

En términos biológicos, u representa las denditras de la neurona, x los impulsos recibidos, b el potencial que debe superar la neurona para dispararse, y ϕ la función de disparo para diferentes impulsos. En la Figura 2.1 se puede apreciar una comparación.

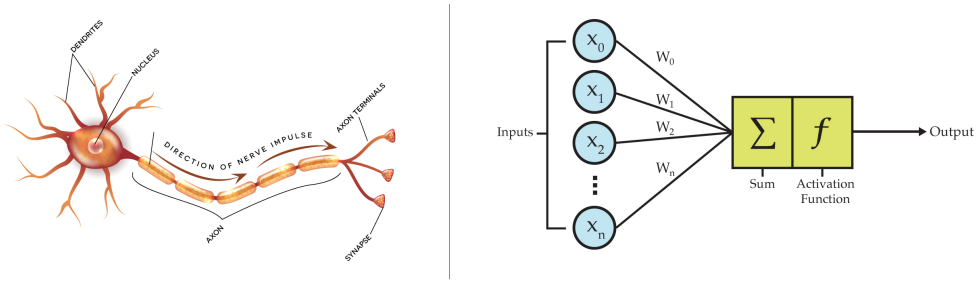

Figura 2.1 Comparación entre una neurona biológica y una artificial: (Izq.) Visualización de una neurona biológica. A la (Der.) la representación de la neurona artifical. Las dendritas actúan como las entradas de la neurona artificial, las terminales del axón actuan como la salida. Para que la neurona dispare un impulso eléctrico, las entradas deben superar una diferencia de potencial; Esta interacción se la representa con los parámetros $(\boldsymbol{w_1}, \boldsymbol{w_2}, \ldots, \boldsymbol{w_n})$, y la función de activación $\boldsymbol{f}$.

La ecuación (2.3) representa el conjunto de neuronas, denominado hidden layer. Es una transformación de $\mathbb{R}^n \rightarrow \mathbb{R}^m$, donde n es el tamaño de la señal de entrada y m el número de neuronas en la capa. W es una matriz, cada fila representa los pesos de una neurona diferente, b es un vector, y la función ϕ se aplica a cada elemento de la transformación afín $(U\boldsymbol{x} + \boldsymbol{b})$.

$$\boldsymbol{h} = \phi(U\boldsymbol{x} + \boldsymbol{b}) \tag{2.3}$$

Por último, la capa de salida usualmente es otra transformación afín. Esta transformación va de $\mathbb{R}^m \rightarrow \mathbb{R}^p$, donde m es el número de neuronas del hidden layer, y p es el número de clases. La capa de salida puede aplicar una transformación monotónica creciente (i.e. ϕ), pero por motivos que se explican luego, limitamos la salida a una transformación afín. En la ecuación (2.4), o representa la salida de la red, y $(\boldsymbol{V}, \boldsymbol{c})$ son los parámetros de la capa.

$$\boldsymbol{o} = V\boldsymbol{h} + \boldsymbol{c} \tag{2.4}$$

El elemento de mayor magnitud en el vector de salida o, corresponde a la clase que la red favorece, dado la entrada y los parámetros actuales. Es decir, si el elemento 5 es el de mayor magnitud, entonces nuestra red asigna la entrada actual a la clase 5. Es completamente válido pensar en redes neuronales que requieren de un vector de salida más pequeño para representar las p clases en la salida (e.g. puedo tener un escalar como salida, y mi clase corresponde al

entero más “cercano” al escalar). Sin embargo, empíricamente la arquitectura propuesta (i.e. una salida por clase) generaliza mejor [1].

Finalmente terminamos con un modelo de la siguiente forma:

$$f(\boldsymbol{x},\theta)\colon \mathbb{R}^n \to \mathbb{R}^p \qquad \text{s.t.} \qquad \theta = \{U, V, \boldsymbol{b}, \boldsymbol{c}\}, \boldsymbol{x} \sim P(X) \tag{2.5}$$

En términos más concretos, una red neuronal es una serie de transformaciones lineales y/o no-lineales, que transforman la entrada de $\mathbb{R}^n \to \mathbb{R}^p$. Esta estructura puede aproximar cualquier función. [1]

2.1.1 Entrenamiento de una red neuronal

Entrenar una red neuronal consiste en encontrar una parametrización (i.e. los valores de θ) que clasifique bien, una entrada en $\mathbb{R}^n$ a una clase en $\mathbb{R}^p$.

La parametrización se puede encontrar de diferentes formas. Por ejemplo, podemos probar parametrizaciones aleatorias hasta encontrar una que consideremos buena. Como corresponde, esto tardaría si tenemos que un espacio de soluciones con billones de parámetros. Actualmente, la parametrización θ se encuentra utilizando un *dataset*.

El dataset contiene pares de muestras $(\mathbb{R}^n, \boldsymbol{y})$, donde $\mathbb{R}^n$ es la entrada que queremos clasificar, y $\boldsymbol{y}$ la clase a la que corresponde $\mathbb{R}^n$. Si el dataset es representativo del fenómeno, se espera, que la red neuronal ajustada para clasificar el dataset, pueda clasificar correctamente muestras que no están en el dataset. Parametrizar la red para que modele un dataset se conoce como *supervised learning*.

Para ajustar la red neuronal al dataset se define una función de error. Esta función va de $\mathbb{R}^n \to \mathbb{R}$, y cuantifica el error que comete el modelo al clasificar el dataset con la parametrización actual. En la literatura, esta función se denomina *cost function* o *loss function*. La función la definimos en la ecuación (2.6).

$$\ell(\boldsymbol{y}, f(\boldsymbol{x},\theta)) = \frac{1}{n}\sum_{i}^{n} \ell(\boldsymbol{y}_i, f(\boldsymbol{x_i},\theta)) \tag{2.6}$$

Donde ℓ es la función de error, n la cantidad de datos en el dataset, $(\boldsymbol{x_i}, \boldsymbol{y_i})$ es un par de elementos del dataset, y $f(x,\theta)$ una red neuronal. La condición es que $\ell \geq 0$ y que ℓ sea diferenciable.

Si obervamos con atención, θ son los parámetros libres de la función, y el par $(\boldsymbol{x_i}, \boldsymbol{y}_i)$ está pre definido por el dataset. También podemos deducir que si reducimos el error mejoramos la clasificación, porque implica que la clasificación de la red se asemeja más a la clasificación real. El proceso de reducir (o minimizar) la función de error se conoce como optimización, y existen varias formas de optimizar una función. Entre las formas de optimizar una función está el algoritmo del descenso del gradiente. A continuación explicamos en que consiste.

2.1.2 Descenso del gradiente

Descenso del gradiente (*gradient descent* en ingles) es un algoritmo de optimización iterativo. Este algoritmo utiliza el gradiente de la función (el vector de derivadas parciales) para ajustar los parámetros, y consecuentemente minimizar la función de error. Las ecuaciones siguientes dan una noción matemática de en que consiste.

Nuestro interés es cambiar θ para garantizar que $\Delta\ell \leq 0$. Es decir, el error final es menor al error inicial. Sabemos que el cambio en ℓ viene dado por:

$$\Delta\ell = \nabla_\theta \ell \cdot \Delta\theta \tag{2.7}$$

Y por el teorema de Cauchy-Schwartz, sabemos que ese producto está limitado de la siguiente forma:

$$-|\Delta_\theta \ell||\nabla\theta| \leq \nabla_\theta \ell \cdot \Delta\theta \leq |\Delta_\theta \ell||\nabla\theta| \tag{2.8}$$

Los limites ocurren cuando el ángulo entre los vectores es $0°$. Aparente, si recordamos la definición geométrica del producto punto, ecuación (2.9).

$$\nabla_\theta \ell \cdot \Delta\theta = |\Delta_\theta \ell||\nabla\theta|\cos(\alpha) \tag{2.9}$$

El límite inferior se da cuando los vectores $(\nabla_\theta \ell, \Delta\theta)$ apuntan en direcciones opuestas (i.e. $\alpha = 180°$). Esto sucede cuando:

$$\Delta\theta = -\eta\nabla_\theta \ell \tag{2.10}$$

Donde, $\eta \in \mathbb{R}$, escala el vector $\nabla_\theta \ell$ sin cambiar su dirección. A η se lo conoce como *learning rate*. Finalmente, si actualizamos los parámetros siguiendo la ecuación (2.11) garantizamos que $\Delta\ell \leq 0$. Minimizando así la función de error.

$$\theta_{\mathrm{f}} = \theta_o - \eta \nabla_\theta \ell \tag{2.11}$$

En resumen, el algoritmo de descenso del gradiente consiste en actualizar los parámetros iterativamente hasta encontrar una buena parametrización.

Así como existen varios algoritmos de optimización, existen varias formas de calcular $\nabla_\theta \ell$. En la siguiente sección presentamos un algoritmo para calcularlo.

2.1.3 Retro-propagación

Es un algoritmo para calcular $\nabla_\theta \ell$ de forma eficiente. Fue desarrollado en 1970, pero se popularizo en 1986 por Rumelhart, Hinton y Williams [2].

En esencia, calcula, eficientemente, la derivada de la función de error con respecto a cada parámetro. La idea es aprovechar la regla de la cadena para calcular el error iterando la red en reversa y almacenando las derivadas. A continuación un ejemplo:

$$\nabla_\theta \ell = \left[\frac{\partial \ell}{\partial U}, \frac{\partial \ell}{\partial V}, \frac{\partial \ell}{\partial \boldsymbol{b}}, \frac{\partial \ell}{\partial \boldsymbol{c}}\right] \tag{2.12}$$

$$\frac{\partial \ell}{\partial U} = \frac{\partial \ell}{\partial \boldsymbol{o}} \frac{\partial \boldsymbol{o}}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial U} \tag{2.13}$$

$$\frac{\partial \ell}{\partial \boldsymbol{b}} = \frac{\partial \ell}{\partial \boldsymbol{o}} \frac{\partial \boldsymbol{o}}{\partial \boldsymbol{h}} \frac{\partial \boldsymbol{h}}{\partial \boldsymbol{b}} \tag{2.14}$$

$$\frac{\partial \ell}{\partial V} = \frac{\partial \ell}{\partial \boldsymbol{o}} \frac{\partial \boldsymbol{o}}{\partial V} \tag{2.15}$$

$$\frac{\partial \ell}{\partial \boldsymbol{c}} = \frac{\partial \ell}{\partial \boldsymbol{o}} \frac{\partial \boldsymbol{o}}{\partial \boldsymbol{c}} \tag{2.16}$$

El algoritmo calcula las derivadas comenzando por la capa salida hasta terminar en los parámetros. (i.e. primero se calcula $\frac{\partial \ell}{\partial \boldsymbol{o}}$ y luego se calcula $\frac{\partial \ell}{\partial \boldsymbol{h}}$).

Asumiendo $\ell = (\boldsymbol{y_i} - f(\boldsymbol{x_i}, \theta))^2$, entonces comenzamos calculando el gradiente con respecto a la salida:

$$\frac{\partial \ell}{\partial \boldsymbol{o}} = 2(\boldsymbol{y}_i - \boldsymbol{o}) \tag{2.17}$$

Después, utilizamos el gradiente con respecto a la salida para calcular el gradiente con respecto al hidden layer.

$$\frac{\partial \ell}{\partial \boldsymbol{h}} = \frac{\partial \ell}{\partial \boldsymbol{o}} \frac{\partial \boldsymbol{o}}{\partial \boldsymbol{h}} = V^T 2(\boldsymbol{y_i} - \boldsymbol{o}) \tag{2.18}$$

Y finalmente, utilizamos el gradiente con respecto al hidden layer para calcular el gradiente con respecto a los parámetros (i.e. $\nabla_\theta \ell$)

$$\frac{\partial \ell}{\partial V} = \frac{\partial \ell}{\partial \boldsymbol{o}} \boldsymbol{h}^{\mathrm{T}} \tag{2.19}$$

$$\frac{\partial \ell}{\partial \boldsymbol{c}} = \frac{\partial \ell}{\partial \boldsymbol{o}} \tag{2.20}$$

$$\frac{\partial \ell}{\partial W} = \frac{\partial \ell}{\partial \boldsymbol{h}} \boldsymbol{x}^T \tag{2.21}$$

$$\frac{\partial \ell}{\partial b} = \frac{\partial \ell}{\partial \boldsymbol{h}} \tag{2.22}$$

Si el lector no está convencido con la aritmética del cálculo de las derivadas, se sugiere [1] para un tratamiento más detallado del algoritmo de retro-propagación. Si el lector quiere un análisis de como se implementa retro-propagación, en la sección 6.5 [2] explica la implementación de TensorFlow.

Hasta el momento hemos propuesto un modelo que puede aproximar cualquier función $\mathbb{R}^n \rightarrow \mathbb{R}^p$; y dado un dataset, podemos encontrar la parametrización θ. Da la impresión que podemos descargar datasets, y entrenar redes neuronales para clasificar cualquier entrada (e.g. Clasificar fotos como gatos o perros). Pero ¿Qué tal si queremos modelar una secuencia de entradas?

El modelo propuesto está especializado en clasificar clases que solo dependen de una entrada. Pero existen decisiones que no solo dependen de una entrada, sino de una secuencia de entradas (e.g. La clasificación de un video - un video es una secuencia de *frames*). Entonces, ¿Cómo utilizar una red neuronal para clasificar una secuencia?

Una solución podría consistir en clasificar la clase para cada entrada de la secuencia, promediar la clasificación de toda la secuencia, y escoger la mayor.

Pero muchos critican la solución por ser agnóstica al orden de las entradas (i.e. el promedio es conmutativo), y puede tener problemas para generalizar (e.g. En un video, el orden de los frames altera el significado).

Otra solución puede consistir en procesar un, tensor de entrada, de rango dos (a diferencia de un tensor de rango uno, como explicamos previamente), y considerar el tiempo como la nueva dimensión. Pero hay que recordar que la parametrización de nuestro modelo es estática, (i.e. el número de parámetros está predefinido) lo que nos obliga a considerar secuencias de un tamaño fijo. Nuevamente, afectando la generalización.

Entonces, ¿Cómo podemos crear una red neuronal que considere toda la secuencia en su decisión, no sea agnóstica al orden, y pueda procesar secuencias de diferente tamaño?

Que tal si manipulamos la capa de neuronas, y conectamos la clasificación en un instante, con la clasificación en el instante anterior. A esta nueva capa la denominaremos unidad recurrente, y es la diferencia entre una red neuronal y una red neuronal recurrente.

2.2 Redes neuronales recurrentes

Una red neuronal recurrente es una arquitectura diseñada para procesar información secuencial [8]. Su característica es la unidad recurrente (**Figura 2.2**), una neurona que conecta la clasificación en un instante de tiempo con instantes de tiempos pasados.

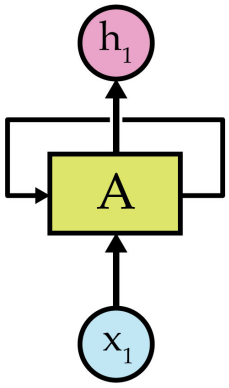

Figura 2.2 Diagrama de una RNN: Visualización de una RNN para una secuencia de longitud t. La unidad de recurrencia es el rectángulo verde "A", "X" es la entrada del modelo, y "h" es la salida del modelo. Imagen reproducida de [9]

Formalmente, el modelo de una red neuronal recurrente es:

$$f^{(t)}\left(\boldsymbol{x}^{(t)}, \boldsymbol{h}^{(t-1)}, \theta\right): \mathbb{R}^n \to \mathbb{R}^p$$
$$\text{s.t.} \quad \theta = \{W, U, V, \boldsymbol{b}, \boldsymbol{c}\}, \boldsymbol{x}^{(t)} \sim P(X) \tag{2.23}$$

$$\boldsymbol{h}^{(t)} = \phi(W\boldsymbol{h}^{(t-1)} + U\boldsymbol{x}^{(t)} + \boldsymbol{b}) \tag{2.24}$$

$$\boldsymbol{o}^{(t)} = V\boldsymbol{h}^{(t)} + \boldsymbol{c} \tag{2.25}$$

En el modelo, la clasificación en cada instante está en función de $(\boldsymbol{x}^{(t)}, \boldsymbol{h}^{(t-1)}, \theta)$. Donde $\boldsymbol{x}^{(t)}$ es la entrada en el instante (t); $\boldsymbol{h}^{(t-1)}$ es la salida de la unidad recurrente en el instante $(t-1)$; y θ son los parámetros del modelo recurrente. Nuevamente, ϕ es una función monotónica creciente.

En términos biológicos, esto implica conectar el axón a un grupo de dendritas y tener otro grupo de dendritas que procesan la entrada. $\boldsymbol{h}^{(t-1)}$ correspode a la salida del axón en el instante (t), y esta salida depende de: la salida del axón en el instante anterior ($\boldsymbol{h}^{(t-1)}$), y la entrada en el instante actual ($\boldsymbol{x}^{(t)}$).

2.2.1 Entrenamiento de una red neuronal recurrente

Existen infinito numero de parametrizaciones distintas para lograr $f^{(t)}(x^{(t)}, h^{(t-1)}, \theta): \mathbb{R}^n \to \mathbb{R}^p$, pero, nuevamente, nos basta una lo suficientemente buena. Recordemos: La búsqueda de una parametrización se conoce como optimización, y es dependiente del dataset, el modelo, y la forma de optimizarlo. El modelo es parametrizado para clasificar correctamente una parte del dataset, y se epera que pueda clasificar bien la parte que no vio del dataset. La sección del dataset utilizada para entrenar se conoce como “dataset de entrenamiento”, la sección utilizada para validar se conoce como “dataset de validación”.

Para ajustar el modelo al dataset, se define una función que cuantifica el error. La ecuación (2.27) cuantifica el error en un instante (t); la ecuación (2.28) cuantifica el error para toda la secuencia.

$$\hat{\boldsymbol{y}}^{(t)} = f^{(t)}(\boldsymbol{x}^{(t)}, \boldsymbol{h}^{(t-1)}, \theta): \mathbb{R}^n \to \mathbb{R}^p \tag{2.26}$$

$$\ell^{(t)}(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)}): \mathbb{R}^n \to \mathbb{R} \tag{2.27}$$

$$\ell = \sum_{t=0}^{\tau} \ell^{(t)}(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)}) : \mathbb{R}^n \to \mathbb{R} \tag{2.28}$$

$$\underset{\theta}{\operatorname{argmin}} \ell = \underset{\theta}{\operatorname{argmin}} \sum_{t=0}^{\tau} \ell^{(t)}(\boldsymbol{y}^{(t)}, f^{(t)}(\boldsymbol{x}^{(t)}, \boldsymbol{h}^{(t-1)}, \theta)) \tag{2.29}$$

Donde $(\boldsymbol{x}^{(t)}, \boldsymbol{y}^{(t)})$ son los pares (entrada, etiqueta) del dataset, θ los parámetros de la red, y ℓ la función de error. Nuestro objetivo es minimizar el ℓ ajustando los parámetros θ.

El proceso es similar al entrenamiento de una red neuronal; Los parámetros se ajustan optimizando la función (2.29) y como algoritmo de optimización utilizamos descenso del gradiente:

$$\theta_f \leftarrow \theta_o - \eta \nabla_\theta \ell \tag{2.30}$$

Pero, ¿Cómo calculamos el gradiente en una red neuronal recurrente? A continuación se presenta un algoritmo para hacerlo: retro-propagación a través del tiempo.

2.2.2 Retro-propagación a través del tiempo

Este algoritmo se fundamenta en la misma idea que el algoritmo de retro propagación.

Para considerar el tiempo, se cambia la presentación propuesta para la red neuronal recurrente. Ya no como la **Figura 2.2**, pero como la **Figura 2.3**. A esta nueva forma se la conoce como el despliegue de la red a través del tiempo.

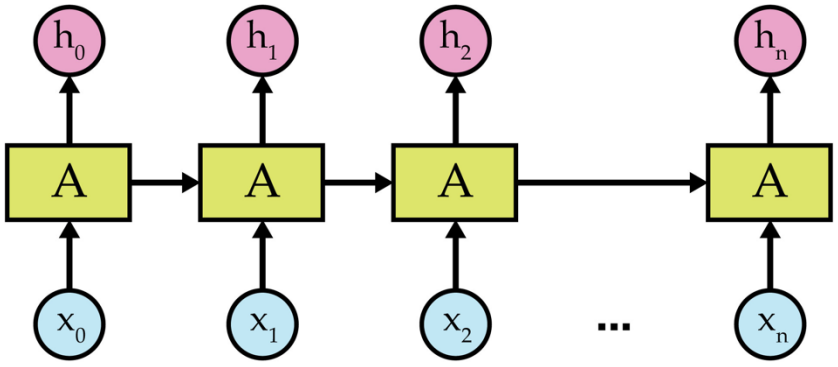

Figura 2.3 Despliegue de una red neuronal recurrente a través del tiempo: Consiste en modelar el tiempo repitiendo la red para cada instante. Imagen reproducida de [9]

Con esta nueva perspectiva, calcular los gradientes es más intuitivo. Primero recordemos:

$$\ell = \sum_{t=1}^{\tau} \ell^{(t)}(\boldsymbol{y}^{(t)}, \hat{\boldsymbol{y}}^{(t)}) \tag{2.28}$$

Al igual que el algoritmo de retro propagación, comenzaremos por la última capa, y por el último instante.

$$\frac{\partial \ell}{\partial \boldsymbol{o}^{(t)}} = \frac{\partial \ell^{(t)}}{\partial \boldsymbol{o}^{(t)}} \tag{2.31}$$

Seguimos con el último estado recurrente:

$$\frac{\partial \ell}{\partial \boldsymbol{h}^{(t)}} = \frac{\partial \ell^{(t)}}{\partial \boldsymbol{o}^{(t)}} \frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{h}^{(t)}} \tag{2.32}$$

Continuamos con la salida en (t-1)

$$\frac{\partial \ell}{\partial \boldsymbol{o}^{(t-1)}} = \frac{\partial \ell^{(t-1)}}{\partial \boldsymbol{o}^{(t-1)}} \tag{2.33}$$

Y nuevamente, con el estado recurrente en (t-1)

$$\frac{\partial \ell}{\partial \boldsymbol{h}^{(t-1)}} = \frac{\partial \ell^{(t-1)}}{\partial \boldsymbol{o}^{(t-1)}} \frac{\partial \boldsymbol{o}^{(t-1)}}{\partial \boldsymbol{h}^{(t-1)}} + \frac{\partial \ell^{(t)}}{\partial \boldsymbol{o}^{(t)}} \frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{h}^{(t)}} \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}} \tag{2.34}$$

Reemplazando la ecuación (1.34) en (1.35) tenemos

$$\frac{\partial \ell}{\partial h^{(t-1)}} = \frac{\partial \ell^{(t-1)}}{\partial \boldsymbol{o}^{(t-1)}} \frac{\partial \boldsymbol{o}^{(t-1)}}{\partial \boldsymbol{h}^{(t-1)}} + \frac{\partial \ell}{\partial \boldsymbol{h}^{(t)}} \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}} \tag{2.35}$$

Hagamos una capa más para tenerlo más claro:

$$\frac{\partial \ell}{\partial \boldsymbol{o}^{(t-2)}} = \frac{\partial \ell^{(t-2)}}{\partial \boldsymbol{o}^{(t-2)}} \tag{2.36}$$

$$\frac{\partial \ell}{\partial \boldsymbol{h}^{(t-2)}} = \frac{\partial \ell^{(t-2)}}{\partial \boldsymbol{o}^{(t-2)}} \frac{\partial \boldsymbol{o}^{(t-2)}}{\partial \boldsymbol{h}^{(t-2)}} + \frac{\partial \ell^{(t-1)}}{\partial \boldsymbol{o}^{(t-1)}} \frac{\partial \boldsymbol{o}^{(t-1)}}{\partial \boldsymbol{h}^{(t-1)}} \frac{\partial \boldsymbol{h}^{(t-1)}}{\partial \boldsymbol{h}^{(t-2)}} + \frac{\partial \ell^{(t)}}{\partial \boldsymbol{o}^{(t)}} \frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{h}^{(t)}} \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}} \frac{\partial \boldsymbol{h}^{(t-1)}}{\partial \boldsymbol{h}^{(t-2)}} \tag{2.37}$$

Factorizando $\frac{\partial \boldsymbol{h}^{(t-1)}}{\partial \boldsymbol{h}^{(t-2)}}$

$$\frac{\partial \ell}{\partial \boldsymbol{h}^{(t-2)}} = \frac{\partial \ell^{(t-2)}}{\partial \boldsymbol{o}^{(t-2)}} \frac{\partial \boldsymbol{o}^{(t-2)}}{\partial \boldsymbol{h}^{(t-2)}} + \left[\frac{\partial \ell^{(t-1)}}{\partial \boldsymbol{o}^{(t-1)}} \frac{\partial \boldsymbol{o}^{(t-1)}}{\partial \boldsymbol{h}^{(t-1)}} + \frac{\partial \ell^{(t)}}{\partial \boldsymbol{o}^{(t)}} \frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{h}^{(t)}} \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}} \right] \frac{\partial \boldsymbol{h}^{(t-1)}}{\partial \boldsymbol{h}^{(t-2)}} \tag{2.38}$$

Reemplazando (2.35) en (2.38)

$$\frac{\partial \ell}{\partial \boldsymbol{h}^{(t-2)}} = \frac{\partial \ell^{(t-2)}}{\partial \boldsymbol{o}^{(t-2)}} \frac{\partial \boldsymbol{o}^{(t-2)}}{\partial \boldsymbol{h}^{(t-2)}} + \left[\frac{\partial \ell}{\partial \boldsymbol{h}^{(t-1)}} \right] \frac{\partial \boldsymbol{h}^{(t-1)}}{\partial \boldsymbol{h}^{(t-2)}} \tag{2.39}$$

Observamos que para calcular el gradiente de un estado recurrente (t), necesitamos el gradiente de la salida en el instante (t), y el gradiente del estado recurrente en (t+1). Si comenzamos desde la última capa y regresamos a la primera, ya habremos calculado el gradiente del estado recurrente en (t+1).

Bueno, hemos logrado calcular los gradientes con respecto a las capas ¿Cómo calculamos los gradientes con respecto a los parámetros?

$$\frac{\partial \ell}{\partial V} = \sum_{t=1}^{\tau} \frac{\partial \ell}{\partial \boldsymbol{o}^{(t)}} (\boldsymbol{h}^{(t)})^T \tag{2.40}$$

$$\frac{\partial \ell}{\partial \boldsymbol{c}} = \sum_{t=1}^{\tau} \frac{\partial \ell}{\partial \boldsymbol{o}^{(t)}} \tag{2.41}$$

$$\frac{\partial \ell}{\partial W} = \sum_{t=1}^{\tau-1} \text{diag}\left(\frac{\partial \boldsymbol{h}^{(t+1)}}{\partial \phi} \right) \frac{\partial \ell}{\partial \boldsymbol{h}^{(t+1)}} (\boldsymbol{h}^{(t)})^T \tag{2.42}$$

$$\frac{\partial \ell}{\partial \boldsymbol{b}} = \sum_{t=1}^{\tau-1} \text{diag}\left(\frac{\partial \boldsymbol{h}^{(t+1)}}{\partial \phi} \right) \frac{\partial \ell}{\partial \boldsymbol{h}^{(t+1)}} \tag{2.43}$$

Hasta ahora hemos visto dos modelos: las redes neuronales y las redes neuronales recurrentes. Sabemos en que consisten y como entrenarlos.

Es claro que la red neuronal recurrente es un modelo más completo que las redes neuronales. Si quisiera que mi red neuronal recurrente se comporte como una red neuronal, simplemente igualo a cero los parámetros recurrentes. Entonces, si las redes neuronales recurrentes son más poderosas ¿Por qué no son tan populares como las redes neuronales?

El problema con las redes neuronales recurrentes es que tienen un comportamiento inestable con respecto sus entradas. Es decir, el impacto que una entrada en el instante (k) puede tener en la clasificación de la red en un instante (t), puede ser muy débil o muy fuerte.

A continuación probamos la premisa anterior.

2.2.3 Inestabilidad de los gradientes

Los gradientes cuantifican la sensibilidad de una variable con respecto a otra. En el caso de las redes neuronales recurrentes, los gradientes de la salida de la red son, o muy sensibles, o poco sensibles a sus entradas. Esto lo denominamos la instabilidad de los gradientes.

Para medir la sensibilidad de la salida en el instante (t) con respecto a la entrada en el instante (k) proponemos calcular la siguiente derivada:

$$\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{x}^{(k)}} = \frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{h}^{(t)}} \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(t-1)}} \frac{\partial \boldsymbol{h}^{(t-1)}}{\partial \boldsymbol{h}^{(t-2)}} \dots \frac{\partial \boldsymbol{h}^{(k+1)}}{\partial \boldsymbol{h}^{(k)}} \frac{\partial \boldsymbol{h}^{(k)}}{\partial \boldsymbol{x}^{(k)}} \tag{2.44}$$

La ecuación (2.44) la podemos resumir como:

$$\frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{x}^{(k)}} = \frac{\partial \boldsymbol{o}^{(t)}}{\partial \boldsymbol{h}^{(t)}} \frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(k)}} \frac{\partial \boldsymbol{h}^{(k)}}{\partial \boldsymbol{x}^{(k)}} \tag{2.45}$$

De la ecuación (2.45)

$$\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(k)}} = \prod_{i=k}^{t-1} \frac{\partial \boldsymbol{h}^{(i+1)}}{\partial \boldsymbol{h}^{(i)}} \tag{2.46}$$

Donde

$$\frac{\partial \boldsymbol{h}^{(i+1)}}{\partial \boldsymbol{h}^{(i)}} = W^T \text{diag}\left(\frac{\partial \boldsymbol{h}^{(i+1)}}{\partial \phi}\right) s.t. \quad \left|diag\left(\frac{\partial \boldsymbol{h}^{(i+1)}}{\partial \phi}\right)\right| \leq \gamma \tag{2.47}$$

De acuerdo con [5], basta que el radio espectral (i.e. el valor propio más grande) de $W^T < \frac{1}{\gamma}$, para que los gradientes desaparezcan. Y es necesario que el radio espectral de $W^T > \frac{1}{\gamma}$, para que los gradientes exploten.

Cuando los gradientes desaparecen, $\boldsymbol{x}^{(k)}$ tiene muy poca o nada de influencia en la salida $\boldsymbol{o}^{(t)}$. Cuando los gradientes explotan, $\boldsymbol{x}^{(k)}$ es un factor decisivo en la salida $\boldsymbol{o}^{(t)}$. La **Figura 2.4** explica la idea.

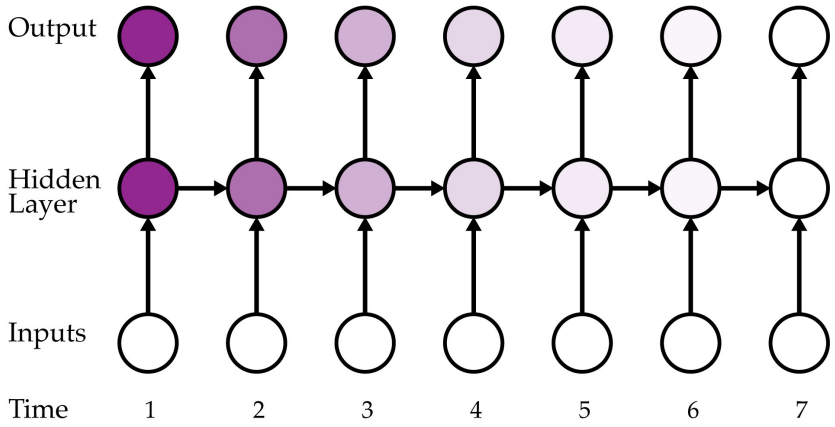

Figura 2.4 Influencia que la primera entrada tiene sobre cada salida: La intensidad del color representa la intensidad de la influencia. Podemos observar que esta influencia tiende a "desvanecerse" en proporción a la distancia temporal. Imagen inspirada por [8]

Esto es un problema cuando la decisión tiene dependencias temporales extensas. Por ejemplo, en un video de 30 segundos a 30 fps, tendremos 900 frames; Si los gradientes desaparecen, la decisión de la red dependerá de los últimos segundos. Por el contrario, si los gradientes explotan, la decisión de la red dependerá, en gran medida, de los primeros segundos.

En el siguiente capitulo vamos a mitigar la inestabilidad de los gradientes, y adicionalmente, definiremos las herramientas para validar la propuesta.

CAPÍTULO 3

3. TODOS LOS CAMINOS LLEVAN A ROMA

En el capitulo anterior presentamos el problema de la inestabilidad de los gradientes. Vimos que la dependencia de un estado recurrente, con respecto a otro estado recurrente, tiende a explotar o desvanecerse.

A continuación presentamos una forma de mitigar el segundo problema: los gradientes que se desvanecen.

3.1 Concepto de la solución

El objetivo es evitar que la influencia de una entrada desaparezca rápidamente (e.g. después de 100 frames). De acuerdo a la ecuación (1.46), esto lo podemos lograr:

- Aumentando el retardo[a] de los parámetro.
- Añadiendo parámetros recurrentes entre dos estados.

$$\frac{\partial \boldsymbol{h}^{(t)}}{\partial \boldsymbol{h}^{(k)}} = \prod_{i=k}^{t-1} \frac{\partial \boldsymbol{h}^{(i+1)}}{\partial \boldsymbol{h}^{(i)}} \qquad (2.46)$$

Los parámetros recurrentes conectan dos estados en el tiempo. Si el retardo de los parámetros recurrentes aumenta, entonces vamos a tener menos transformaciones entre dos estados.

Un contra argumento, es que si el retardo es grande, perderemos la información que suceda dentro del periodo del retardo. Si el retardo es pequeño seguiremos teniendo más o menos el mismo numero de transformaciones. La **Figura 3.1** ilustra la idea.

[a] Con retardo nos referimos a la cantidad de instantes de tiempo antes de transformar el estado recurrente $h^{(t)}$. Las redes neuronas recurrentes básicas tienen un retardo de uno.

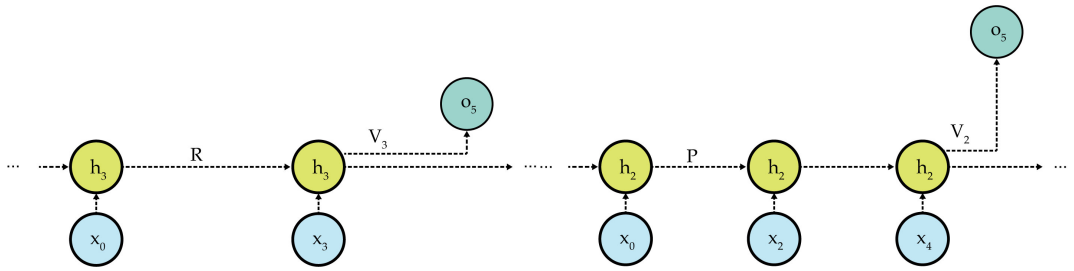

Figura 3.1 Comparación entre redes con diferentes retardos: (Izq.) Una red neuronal recurrente con retardo de tres instantes pierde las entradas $[x_1, x_2, x_4, x_5]$, pero reduce las transformaciones entre los estados 1 y 5. En comparación (Der.), una red neuronal recurrente con retardo de dos instantes, pierde las entradas $[x_1, x_3, x_5]$, pero tiene más transformaciones entre los estados 1 y 5.

La segunda opción consiste en añadir más parámetros recurrentes entre estados, sumando términos similares a (2.46) a la ecuación (2.45). Un contra-argumento, es que si los nuevos términos tienen la misma dinámica inestable, la suma de estos términos tendrá la misma dinámica inestable.

Teóricamente, la primera pierde información, y la segunda mantiene la dinámica que tratamos de resolver. Pero, ¿Cómo se comportan las dos juntas? ¿Qué sucede si utilizamos más parámetros recurrentes, pero con diferente retardo? Respondamos a ambas preguntas con un ejemplo.

Primero planteamos el problema desde otro punto de vista. Recordemos las ecuaciones de una red neuronal recurrente:

$$\boldsymbol{h}^{(t)} = \phi(W\boldsymbol{h}^{(t-1)} + U\boldsymbol{x}^{(t)} + \boldsymbol{b}) \tag{2.24}$$

$$\boldsymbol{o}^{(t)} = V\boldsymbol{h}^{(t)} + \boldsymbol{c} \tag{2.25}$$

Simplificamos el argumento asumiendo que la función ϕ es la identidad. Como resultado:

$$\boldsymbol{h}^{(t)} = \sum_{i=1}^{t} W^{t-i} U\boldsymbol{x}^{(i)} + W\boldsymbol{h}^{(0)} \tag{3.1}$$

Asumiendo $h^{(0)} = 0$, entonces:

$$\boldsymbol{h}^{(t)} = \sum_{i=1}^{t} W^{t-i} U\boldsymbol{x}^{(i)} \tag{3.2}$$

$$\boldsymbol{o}^{(t)} = \sum_{i=1}^{t} VW^{t-i} U\boldsymbol{x}^{(i)} \tag{3.3}$$

Asumiendo que W permite una descomposición en valores singulares $Q^T \Lambda Q$.

$$\boldsymbol{o}^{(t)} = \sum_{i=1}^{t} V Q^T \Lambda^{t-i} Q U \boldsymbol{x}^{(i)} \tag{3.4}$$

Los valores propios de los primeros recurrentes (W) están elevados a una potencia. Consecuentemente, valores propios mayores a uno explotan mientras que valores propios menores a uno se desvanecen. Podemos concluir que la salida de la red, y por ende su decisión, depende de entradas alineadas con los valores propios más grandes.

En la Figura 3.2 se presenta un ejemplo de tres neuronas recurrentes, $\boldsymbol{h_1^{(t)}}$, $\boldsymbol{h_2^{(t)}}$, $\boldsymbol{h_2^{(t)}}$, con retardos de uno, dos y tres instancias respectivamente. Los parámetros recurrentes son W, P, R; y la salida tiene como parámetros V_1, V_2, V_3

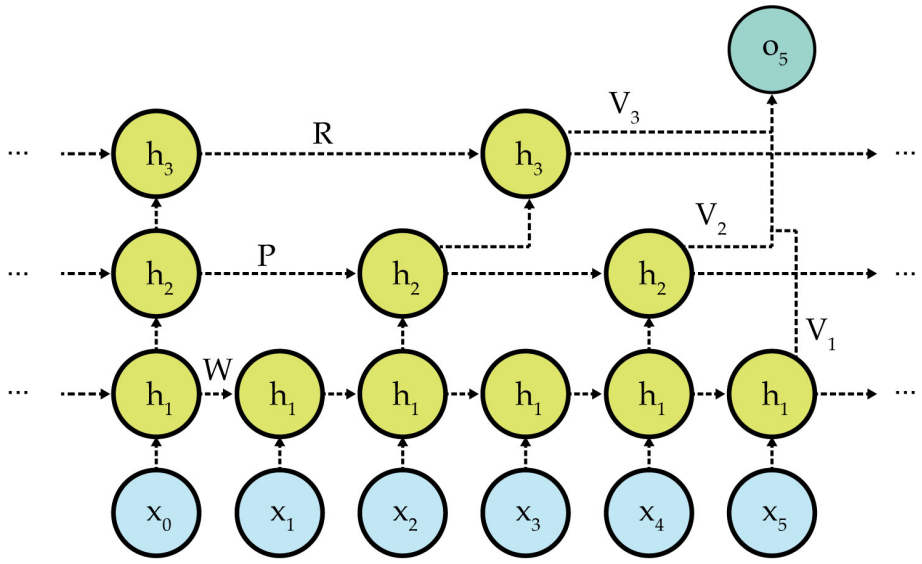

Figura 3.2 Ejemplo de D-RNN: Las neuronas recurrentes son los nodos (h_1, h_2, h_3). ($x_1, \dots, x_5$) y ($o_1, o_2, \dots, o_5$) corresponden a las capas de entrada y salida respectivamente.

Ahora calculamos la salida $\boldsymbol{o}^{(5)}$en función de las entradas de la red. En las ecuaciones que se muestran $\left(\boldsymbol{o}^{(t)}, \boldsymbol{x}^{(t)}\right)$ corresponden a $(\boldsymbol{o_t}, \boldsymbol{x_t})$ en la Figura 3.2

$$\boldsymbol{o}^{(5)} = V_1 \boldsymbol{h_1^{(5)}} + V_2 \boldsymbol{h_2^{(4)}} + V_3 \boldsymbol{h_3^{(3)}} \tag{3.5}$$

$$\boldsymbol{h_3^{(3)}} = R \boldsymbol{h_3^{(0)}} + S \boldsymbol{h_2^{(2)}} \tag{3.6}$$

$$\boldsymbol{h_2^{(4)}} = P\boldsymbol{h_2^{(2)}} + T\boldsymbol{h_1^{(4)}} \tag{3.7}$$

$$\boldsymbol{h_1^{(5)}} = \sum_{i=1}^{5} W^{5-i} U\boldsymbol{x^{(i)}} \tag{3.8}$$

$$\boldsymbol{h_2^{(4)}} = \sum_{i=1}^{2} PTW^{2-i} U\boldsymbol{x^{(i)}} + \sum_{i=1}^{4} TW^{4-i} U\boldsymbol{x^{(i)}} \tag{3.9}$$

$$\boldsymbol{h_3^{(3)}} = \sum_{i=1}^{2} SPTW^{2-i} U\boldsymbol{x^{(i)}} \tag{3.10}$$

$$\begin{aligned}\boldsymbol{o^{(5)}} = \sum_{i=1}^{5} V_1 W^{5-i} U\boldsymbol{x^{(i)}} + \sum_{i=1}^{2} V_2 PTW^{2-i} U\boldsymbol{x^{(i)}} \\ + \sum_{i=1}^{4} V_2 TW^{4-i} U\boldsymbol{x^{(i)}} \\ + \sum_{i=1}^{2} V_3 SPTW^{2-i} U\boldsymbol{x^{(i)}}\end{aligned} \tag{3.11}$$

Calculando las dependencias con respecto a $\boldsymbol{x^{(1)}}$:

$$\begin{aligned}\boldsymbol{o^{(5)}} = V_1 W^4 U\boldsymbol{x^{(1)}} + V_2 PTWU\boldsymbol{x^{(1)}} + V_2 TW^3 U\boldsymbol{x^{(1)}} \\ + V_3 SPTWU\boldsymbol{x^{(1)}}\end{aligned} \tag{3.12}$$

$$\begin{aligned}\boldsymbol{o^{(5)}} = (V_1 W^4 U + V_2 PTWU + V_2 TW^3 U \\ + V_3 SPTWU)\boldsymbol{x^{(1)}}\end{aligned} \tag{3.13}$$

Comparado a una RNN de una neurona recurrente:

$$o^{(5)} = V_1 W^4 U\boldsymbol{x^{(1)}} \tag{3.14}$$

Si asumimos que el radio espectral (i.e. el valor singular de mayor magnitud) de W, P, R es menor a uno, entonces nuestro modelo está en mayor capacidad, con respecto al modelo de una neurona recurrente, de considerar a $\boldsymbol{x^{(1)}}$ en la decisión final. En conclusión, las nuevas neuronas recurrentes aumentan las dependencias temporales que se pueden modelar.

3.2 Presentación formal de la solución

Dado dos neuronas recurrentes (i, j), con periodos (τ_i, τ_j), parámetros (W_i, U_i), (W_j, U_j), estados recurrentes $(\boldsymbol{h_i}, \boldsymbol{h_j})$; $\tau_i > \tau_j$; $\{\nexists \tau_k \in \tau | \tau_i > \tau_k > \tau_j, \tau \in \mathbb{Z}^+\}$ donde τ es el conjunto de periodos de las neuronas recurrentes en la red, entonces:

$$\boldsymbol{h_i^{(t)}} = \phi\left(W_i \boldsymbol{h_i^{(t-1)}} + U_i \boldsymbol{h_j^{(t)}}\right) \quad \text{s.t.} \quad t(\text{mod}\tau_i) = 0 \tag{3.15}$$

$$\boldsymbol{h_i^{(t)}} = \boldsymbol{h_i^{(t-1)}} \quad \text{s.t.} \quad t(\text{mod}\tau_i) \neq 0 \tag{3.16}$$

Si calculamos la sensibilidad de una unidad recurrente con respecto a sus estados anteriores, tenemos:

$$\frac{\partial \boldsymbol{h_i^{(t)}}}{\partial \boldsymbol{h_i^{(k)}}} = \prod_{p=k}^{t-1} \frac{\partial \boldsymbol{h_i^{(p+1)}}}{\partial \boldsymbol{h_i^{(p)}}} \tag{3.17}$$

$$\frac{\partial \boldsymbol{h_i^{(p+1)}}}{\partial \boldsymbol{h_i^{(p)}}} = W_i^T \text{diag}\left(\frac{\partial \boldsymbol{h_i^{(p+1)}}}{\partial \phi}\right) \quad \text{s.t.} \quad t(\text{mod}\tau_i) = 0 \tag{3.18}$$

$$\frac{\partial \boldsymbol{h_i^{(p+1)}}}{\partial \boldsymbol{h_i^{(p)}}} = 1 \quad \text{s.t.} \quad t(\text{mod}\tau_i) \neq 0 \tag{3.19}$$

Asumiendo que el radio espectral de $W^T < \frac{1}{\gamma}$, donde $\left|\frac{\partial h_i^{(p+1)}}{\partial \phi}\right| = \gamma$; El modelo incrementa $\frac{\partial h_i^{(t)}}{\partial h_i^{(k)}}$ porque $W_i^T \text{diag}\left(\frac{\partial h_i^{(p+1)}}{\partial \phi}\right) = I$, para instantes que no sean múltiplos de τ_i.

3.3 Modelo de la solución

$$\boldsymbol{h_0^{(t)}} = \boldsymbol{x^{(t)}} \tag{3.20}$$

$$\boldsymbol{h_i^{(t)}} = \phi\left(W_i \boldsymbol{h_i^{(t-1)}} + U_i \boldsymbol{h_j^{(t)}}\right) \quad \text{s.t.} \quad t(\text{mod}\tau_i) = 0 \tag{3.21}$$

$$\boldsymbol{h_i^{(t)}} = \boldsymbol{h_i^{(t-1)}} \quad \text{s.t.} \quad t(\text{mod}\tau_i) \neq 0 \tag{3.22}$$

$$\boldsymbol{h^{(t)}} = \left[\boldsymbol{h_1^{(t)}}, \ldots, \boldsymbol{h_{\tau_n}^{(t)}}\right] \quad \text{s.t.} \quad \tau = \{0,1,\ldots,\tau_n\} \tag{3.23}$$

$$\boldsymbol{o^{(t)}} = V\boldsymbol{h^{(t)}} + c \tag{3.24}$$

$$\hat{\mathbf{y}}^{(t)} = \text{softmax}(\boldsymbol{o^{(t)}}) \tag{3.25}$$

$$\ell^{(t)} = -ln((\boldsymbol{y^{(t)}})^T \hat{\mathbf{y}}^{(t)}) \tag{3.26}$$

$$\ell = \sum_{t=1}^{\tau} \ell^{(t)} \tag{3.27}$$

En el diseño aparece una transformación de la que no hemos hablado (i.e. *softmax*) y una función de costo conocida como *cross-entropy*.

De acuerdo a [1] y [2] softmax es una forma de representar una distribución de probabilidad sobre p diferentes clases. Esto convierte la salida de la red en una distribución, donde cada elemento representa la probabilidad de que la secuencia pertenezcan a esa clase. Softmax tiene la forma:

$$\text{softmax}(o_i^{(t)}) = \frac{e^{o_i^{(t)}}}{\sum_j e^{o_j^{(t)}}} \tag{3.28}$$

Donde los sub-índices i, j representan un elemento del vector de salida.

Cross-Entropy es una ecuación de la forma:

$$\ell^{(t)} = \boldsymbol{y}^{(t)} ln(\hat{\boldsymbol{y}}^{(t)}) - (1 - \boldsymbol{y}^{(t)}) ln(1 - \hat{\boldsymbol{y}}^{(t)}) \tag{3.29}$$

Como la representación de nuestras etiquetas está en *one-hot,* se convierte en:

$$\ell^{(t)} = -ln((\boldsymbol{y}^{(t)})^T \hat{\boldsymbol{y}}^{(t)}) \tag{3.30}$$

Juntas, cross-entropy y softmax evitan que la derivada en la última capa se desvanezca.

Esto es importante porque, como efecto secundario de la desaparición de los gradientes, la red tarda más tiempo en aprender de entradas que suceden lejos en el pasado. Softmax y cross-entropy evitan que el primer término de los gradientes $\nabla_\theta \ell$ tienda a desvanecerse (i.e. $\frac{\partial \ell^{(t)}}{\partial \boldsymbol{o}^{(t)}}$).

Ahora que tenemos un modelo preparado, vamos a escoger: la plataforma para implementarlo, un dataset para entrenarlo, y un esquema para validarlo.

3.4 Deep Learning Frameworks

El desarrollo de una red neuronal se puede resumir en las siguientes partes: Procesamiento del dataset, definición del modelo, entrenamiento, validación, debug, y exportar el modelo para hacer inferencias. Para facilitar el proceso

descrito anteriormente, existen varios API conocidos como *Deep Learning Frameworks*.

A largo de los últimos años, compañías han liberado a la comunidad sus frameworks de Deep Learning, entre estos están: Theano, Torch, PyTorch, Caffe, Keras, TensorFlow.

Las características que necesitamos de un deep learning framework son:

- Flexibilidad: La red que se va a implementar es nueva, y necesitamos un framework que nos permita realizar las conexiones entre diferentes instantes de tiempo.
- Portabilidad: Una de las ventajas de la solución es que reduzca los requerimientos computacionales de soluciones actuales, y consecuentemente, que se pueda implementar en varios dispositivos.
- Escalabilidad: Para agilizar entrenamiento, es importante que el modelo escale fácilmente con los dispositivos computacionales disponíbles en el sistema.
- Documentación: Como el modelo es nuevo, es importante tener buena documentación disponible.

Caffe y Keras son frameworks que implementan TensorFlow a bajo nivel, tienen interfaces más simples de usar pero sacrifican flexibilidad por facilidad de aprendizaje. Torch y PyTorch son frameworks flexibles pero no escalables [10]. Theano tiene buena documentación por sus casi diez años en la industria, pero Yoshua Benigno (Director de MILA) anunció en Septiembre 2017 que MILA dejará de desarrollar Theano. A diferencia de los frameworks propuestos, TensorFlow es flexible, escalable, portable, ademas de estar bien documentado y en constante desarrollo.

3.5 TensorFlow

TensorFlow es una librería Open Source para computación numérica que utiliza el paradigma DataFlow.

El paradigma de programación DataFlow es común en sistemas distribuidos. Consiste en expresar el sistema como un grafo, donde las operaciones son

nodos, y los arcos son arreglos multidimensionales (tensores) [6]. Las redes neuronales se pueden expresar naturalmente como grafos, donde cada transformación es un nodo, y la salida de cada transformación son los arcos; esta similitud entre redes neuronales y grafos vuelve DataFlow un paradigma flexible al momento de implementar redes neuronales con nuevas conexiones (Justo lo que necesitamos para nuestra solución).

A diferencia de un paradigma de programación imperativa, no existe un orden de ejecución pre definido, las operaciones se ejecutan en cuanto sus entradas sean válidas. Esto sugiere dos cosas: la ejecución no está restringida a ser secuencial, sino que puede suceder de forma paralela; y cada parte del grafo funciona como una “caja negra” porque solo está pendiente de la validez de sus entradas. Ambas razones facilitan la paralelización del sistema y consecuentemente su escalabilidad.

Google liberó TensorFlow en Noviembre del 2015. Desde entonces se ha convertido en la librería con mayor interés público (**Figura 3.3**). Con una comunidad más grande que los otros frameworks, es natural que su documentación sea más amplia.

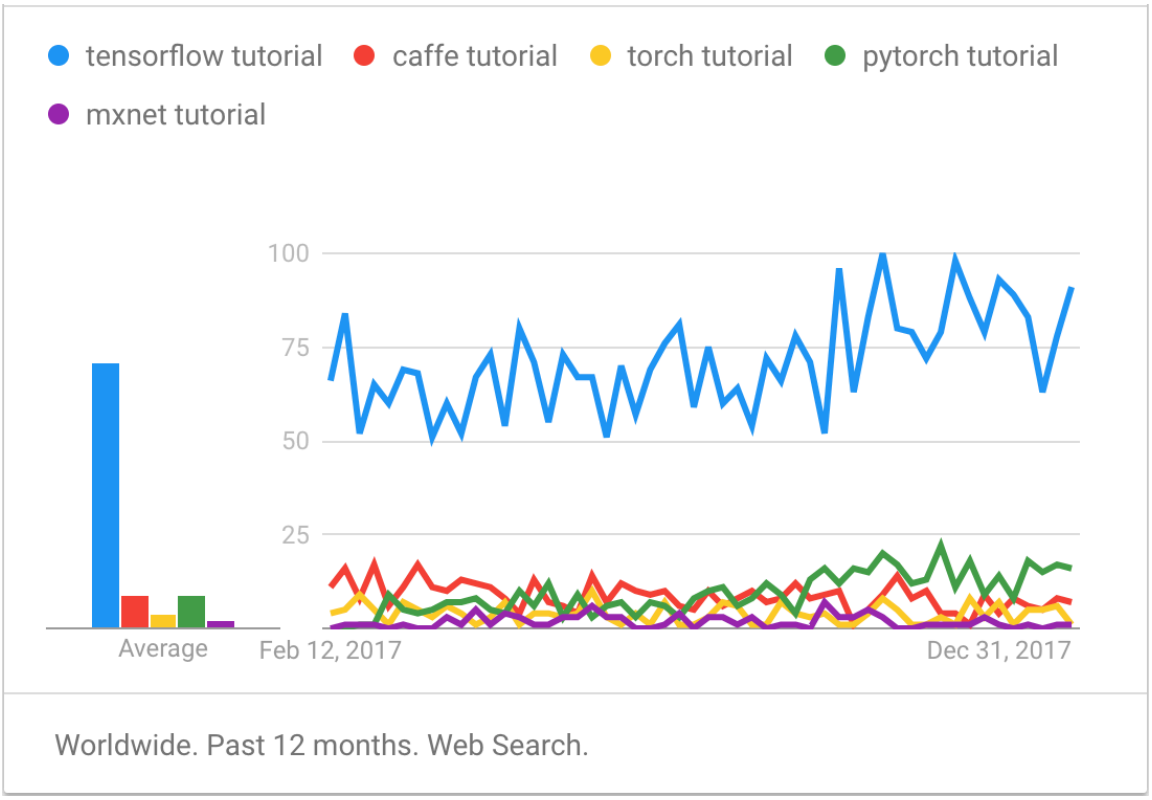

Figura 3.3 Popularidad de diferentes frameworks de deep learning: desde 02/2017 – 02/2018. Inspirado por [7]

Finalmente, TensorFlow es portable por sus diferentes interfaces: Python, C/C++, Java, Go, R; y por TensorFlow Lite, una solución de bajo costo computacional para exportar modelos en TensorFlow a plataformas móviles o sistemas embebidos.

En forma general, TensorFlow es un sistema de machine learning escalable, que opera en ambientes heterogéneos [6].

Escalable porque escala con la cantidad de recursos del sistema; y "opera en ambientes heterogéneos" porque puede ejecutar en diferentes dispositivos a la vez (e.g. CPU, GPU, TPU).

Los tres principios de diseño de TensorFlow son:

Operadores primitivos dentro del grafo computacional: TensorFlow define todas las operaciones, desde multiplicación entre matrices hasta la actualización de los parámetros, como nodos dentro de un grafo computacional. TensorFlow luego distribuye los nodos entre todos los dispositivos disponibles; entre más dispositivos tenemos, menor es la carga sobre cada dispositivo y consecuentemente mayor es el *throughput* del sistema.

Construcción y ejecución separados: El grafo computacional consta de *placeholders* para las entradas, *operations* para las operaciones, y *variables* para el estado del sistema. Todo grafo en TensorFlow es definido previo a su ejecución. Luego, TensorFlow aprovecha la definición completa del grafo para optimizar su ejecución.

Abstracciones para diferentes dispositivos: Cada operador puede tener diferentes implementaciones para diferentes dispositivos (conocidos como *kernels*). Esto vuelve a TensorFlow capaz de operar en ambientes heterogéneos.

Los principios de diseño 1 y 2 son la guía principal al momento de construir una red neuronal: TensorFlow brinda los nodos para realizar las transformaciones que necesitamos, y la construcción y ejecución del grafo están separadas.

Los elementos principales para construir el grafo computacional son:

Tensores: Son arreglos n-dimensionales donde los elementos comparten el mismo tipo de dato (e.g. int64, float32, string, uint8). Un tensor se caracteriza por su grado (i.e. el número de dimensiones), su forma (i.e. el número de elementos en cada dimensión), y su tipo de dato.

Operaciones: Son los nodos del grafo computacional. Representan las transformaciones de la red (e.g. multiplicación entre matrices, suma entre vectores,), y/o acciones que del sistema (e.g. guardar, cargar). Pueden recibir cero o más tensores como entrada, y su salida puede ser de cero o más tensores. Y su comportamiento puede cambiar en función de los parámetros que recibe.

Operaciones con estado - Variables: Son un tipo de operaciones cuyo estado persiste en memoria. Cada operación de tipo variable tiene un buffer para almacenar su estado, y un identificador para leer y/o escribir al buffer. Este tipo de operaciones no tiene entrada.

Operaciones con estado - Colas: TensorFlow implementa colas para coordinar la ejecución. Al igual que las variables, cada cola tiene un identificador para manipularla. Las colas implementan dos operaciones: encolar y desencolar. La operación encolar se bloquea si la cola está llena, y la operación desencolar se bloquea si no tenemos entradas disponibles. Los utilizaremos para separar el pre-procesamiento de los datos, y la ejecución del modelo.

En resumen, utilizaremos las colas para almacenar el dataset, las variables para almacenar los parámetros del modelo, las operaciones para representar las transformaciones, y los tensores serán las salidas y entradas de cada nodo.

Para explicar TensorFlow comenzaremos implementando una red neuronal, luego implementaremos una red neuronal recurrente, y finalmente implementaremos la solución propuesta en el capítulo 2 (i.e. D-RNN). En cada implementación se detalla la idea detrás del código. La explicación de la implementación está en los anexos y en el github del proyecto.

Antes de comenzar a implementar el modelo en TensorFlow, vamos a escoger un esquema que nos permita validar que tan bien generaliza nuestro nuevo modelo.

3.6 Dataset: NTU-RGB+d

En el capítulo anterior definimos la función de error como:

$$\underset{\theta}{\operatorname{argmin}} \sum_{t=0}^{\tau} \ell^{(t)}(\boldsymbol{y}^{(t)}, f^{(t)}(\boldsymbol{x}^{(t)}, \boldsymbol{h}^{(t-1)}, \theta)) \tag{2.29}$$

Donde $(\boldsymbol{x}^{(t)}, \boldsymbol{y}^{(t)})$ es el par entrada y etiqueta, en el instante (t) de una secuencia de longitud (τ).

Teóricamente, la secuencia, y por ende sus pares $(\boldsymbol{x}^{(t)}, \boldsymbol{y}^{(t)})$, provienen del dataset, y contribuyen a la forma de la función que vamos a minimizar. Empíricamente, la variabilidad observada en el dataset limita la capacidad de la parametrización para generalizar. Ademas, modelos recurrentes requieren de bastantes muestras para converger a una buena solución. Consecuentemente, se prefiere el dataset que considere:

- Primero, la variabilidad entre las muestras. Esto es importante para que el modelo pueda generalizar a otros contextos. Ejemplos de variabilidad son el número de sujetos distintos, o número de puntos de vista
- Segundo, el número de clases. Entre más actividades hay, el modelo está desafiado a distinguir entre clases similares. Adicionalmente, la decisión del modelo es más precisa en un contexto real.
- Tercero, el número de muestras. Las redes neuronales, y en especial las recurrentes, necesitan de bastantes muestras para converger a una buena solución.
- Cuarto, que presente información sobre los esqueletos. El objetivo de este proyecto es proponer un modelo recurrente y compararlo con otros. Por esta razón, no se distribuye esfuerzos en utilizar una CNN (i.e. Convolutional Neural Network) para procesar RGB frames.

En [3] se comparan varios datasets para reconocimiento de actividades en 3D. El NTU-RGB+d dataset consiste de 56,880 secuencias en cuatro formatos distintos: Videos RGB, Mapas de profundidad, Data de esqueletos en 3D, Videos infrarrojos. Filma a 40 sujetos, desde 80 puntos de vista distintos, realizando un total de 60 acciones distintas. Con respecto a los datasets más completos hasta

el momento (e.g. Act4, UWA3D, RGBD-HuDaAct), NTU-RGB+d tiene, aproximadamente, 8 veces más muestras, 2 veces más clases, 1.75 veces más sujetos , y 16 veces más puntos de vista.

Las secuencias fueron capturadas por tres cámaras Kinect V.2 funcionando concurrentemente. La resolución de los videos son 1920x1080, la resolución de los mapas de profundidad y videos infrarrojos son 512x424, y los datos del esqueleto en 3d contiene la ubicación en tres dimensiones de las 25 junturas principales del cuerpo.

El dataset consta de 60 acciones, divididas en tres grupos principales: 40 actividades diarias, 9 actividades relacionadas a la salud, y 11 actividades entre dos personas.

Cada acción fue filmada desde tres ángulos diferentes: -45°, 0°, +45°. Las acciones fueron repetidas, dos veces, por 40 sujetos diferentes; una vez mirando a la cámara a -45°, y otra viendo a la cámara de +45°. De esta forma, se captura la acción desde 5 perspectivas diferentes. La cámara 1 es la cámara central (siempre con vistas de 45°); la cámara 2 y 3 son las cámaras laterales (siempre con vistas frontales y laterales).

Ademas, también se cambiaron la altura y distancia de las tres cámaras. En el dataset esto corresponde al número de setup.

El dataset tiene dos modalidades de prueba: Cross-Subject y Cross-View.

Cross-View consiste en entrenar el modelo con la vista de la cámaras 2 y 3, y generalizar a la vista de la cámara 1. Cross-Subject consiste en entrenar la red con 20 sujetos (1, 2, 4, 5, 8, 9, 13, 14, 15, 16, 17, 18, 19, 25, 27, 28, 31, 34, 35, 38), y generalizar a los que faltan.

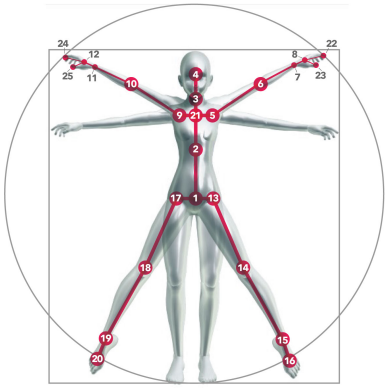

Figura 3.4 Configuración de las 25 junturas registradas por Kinect V2: 1-base de la columna, 2-centro de la columna 3-cuello, 4-cabeza, 5-hombro izquierdo, 6-codo izquierdo, 7-muñeca izquierda, 8-mano izquierda, 9-hombro derecho, 10-codo derecho, 11-muñeca derecha, 12-mano derecha, 13-cadera izquierda, 14-rodilla izquierda, 15-tobillo izquierdo, 16-tobillo derecho, 17-cadera derecha, 18-rodilla derecha, 19-tobillo derecho, 20-pie derecho. Imagen inspirada por [3]

1. Beber agua	21. Quitarse gorra	41. Estornudar/toser
2. Comer/snack	22. Alentar	42. Tambalearse
3. Lavarse los dientes	23. Sacudir mano	43. Caerse
4. Peinarse	24. Patear algo	44. Dolor de cabeza
5. Soltar	25. Guardar dentro de bolsillo	45. Dolor de pecho
6. Recoger	26. Saltar sobre un pie	46. Dolor de espalda
7. Lanzar	27. Saltar	47. Dolor de cuello
8. Sentarse	28. Llamar/contestar	48. Nausea o vómito
9. Levantarse (sentado)	29. Jugar con celular/tablet	49. Ventilarse (con mano/papel)
10. Aplaudir	30. Teclear (Teclado)	50. Golpear/bofetear al otro
11. Leer	31. Apuntar con un dedo	51. Patear al otro
12. Escribir	32. Tomar un selfie	52. Empujar al otro
13. Arrancar papel	33. Ver la hora (reloj)	53. Palmada en la espalda
14. Ponerse chaqueta	34. Frotar manos entre si	54. Apuntar con el dedo al otro
15. Quitarse chaqueta	35. Cabeceando/Reverencia	55. Abrazar al otro
16. Ponerse zapato	36. Sacudir cabeza	56. Dar algo
17. Quitarse zapato	37. Limpiarse la cara	57. Tocar el bolsillo del otro
18. Ponerse lentes	38. Saludar	58. Apretón de manos
19. Quitarse lentes	39. Juntar las palmas	59. Caminar hacia el otro
20. Ponerse gorra	40. Cruzar las manos (detener)	60. Alejarse del otro

Tabla 3.1 Listado de las 60 actividades del NTU-RGB+d Dataset: 1-40 son actividades diarias, 41-49 son actividades relacionadas a la salud, 50-60 son actividades de dos personas.

Utilizaremos el NTU-RGB+d como dataset para entrenar la red neuronal recurrente, y validaremos la solución con la modalidad cross-view. La métrica de evaluación es el porcentaje de clasificaciones correctas.

Si nuestra solución funciona, esperamos tener mejores resultados que una RNN. Adicionalmente, compararemos los resultados de D-RNN, con los resultados de arquitecturas más complejas propuestas en [4].

En los anexos se explica la implementación de una red neuronal, una red neuronal recurrente y a D-RNN; y se aprovecha la oportunidad para explicar TensorFlow en forma abstracta.

En el siguiente capítulo se analizan los resultados.

CAPÍTULO 4

4. EXPERIMENTOS Y RESULTADOS

4.1 Metodología de los experimentos

De acuerdo con [1], la mejor validación para un modelo es probarlo con situaciones que no ha visto antes. Por ende, se utiliza la modalidad cross-view del NTU-RGB+d dataset para validar los modelos propuestos.

La modalidad consiste en entrenar el modelo con las cámaras laterales (denominadas cámaras 2 y 3 en el cap. 2), y validar el modelo con la cámara central (denominado cámara 1 en el cap. 2). Es decir, nuestro modelo aprenderá a reconocer acciones desde cuatro puntos de vista, y generalizará a un quinto punto de vista.

El prototipo (denominado D-RNN) lo vamos a comparar con otras dos arquitecturas recurrentes: una LSTM y una RNN. DRNN es instanciado con 150 neuronas recurrentes, separadas en 5 grupos de 30 neuronas; Los grupos tienen retardos de 1, 3, 9, 27, y 81 frames respectivamente. La RNN y LSTM tienen 150 neuronas recurrentes con retardos de 1 frame cada una. Las figuras **4.1**, **4.2**, **4.3** muestran el grafo computacional de las tres arquitecturas:

Los modelos se entrenan una instancia en Google Cloud con un GPU Nvidia Tesla K80 con 12Gb de memoria. El entrenamiento se detiene después de 100K batches de entrenamiento.

Como mencionamos en el capítulo uno, la función de error es construida con una muestra de todo el dataset (denominada batch). En los experimentos aproximamos la función de error con *batches* de 60 secuencias.

Los gradientes son calculados utilizando el optimizador *RMSProp*. Esto lo hacemos por dos razones:

Para guardar consistencia con el esquema de entrenamiento propuesto por [4].

Porque empiricamente converge a mínimos más rápido que SGD (*Stochastic Gradient Descent*) o Adagrad.

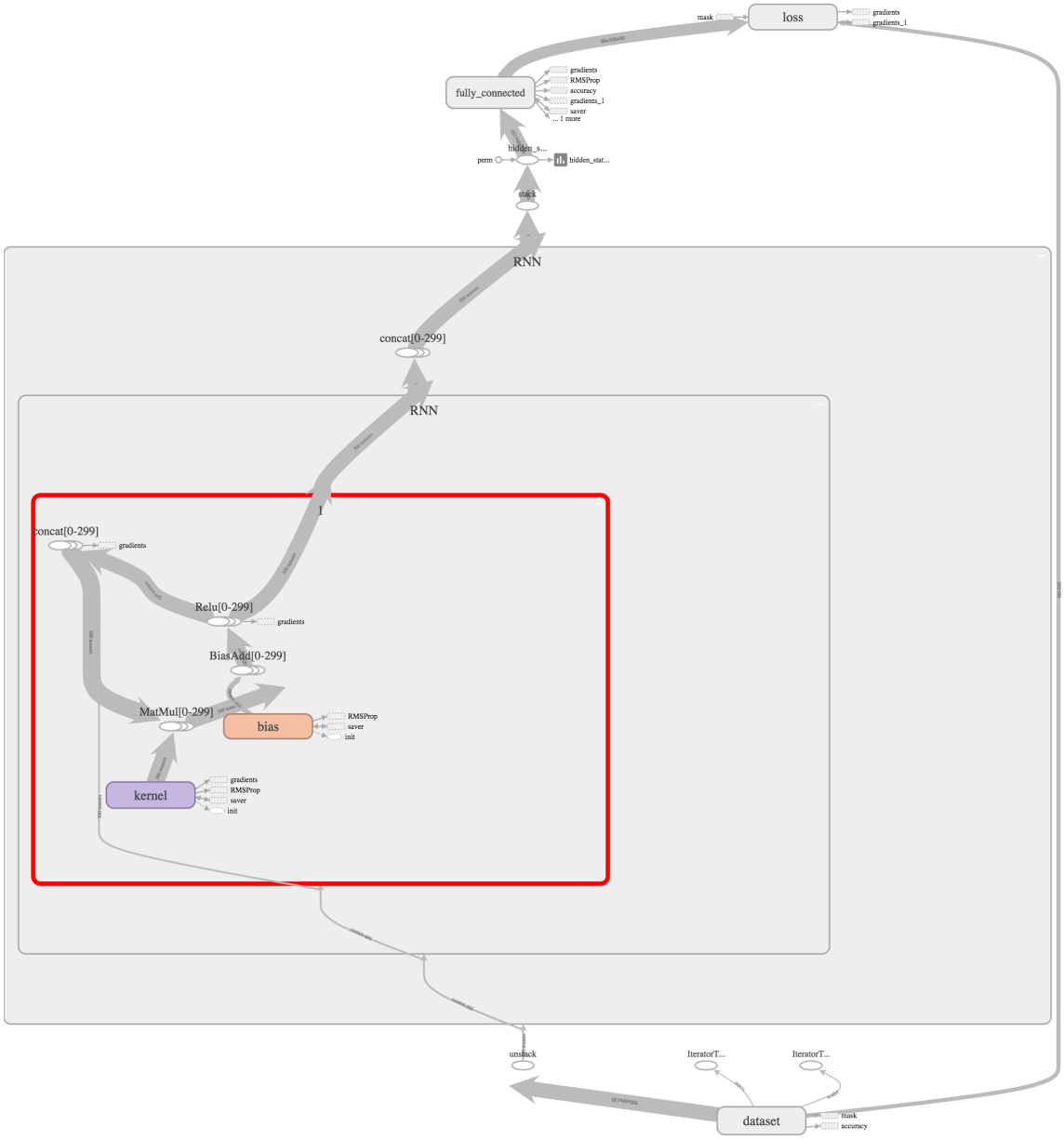

Figura 4.1 Grafo computacional de la arquitectura RNN

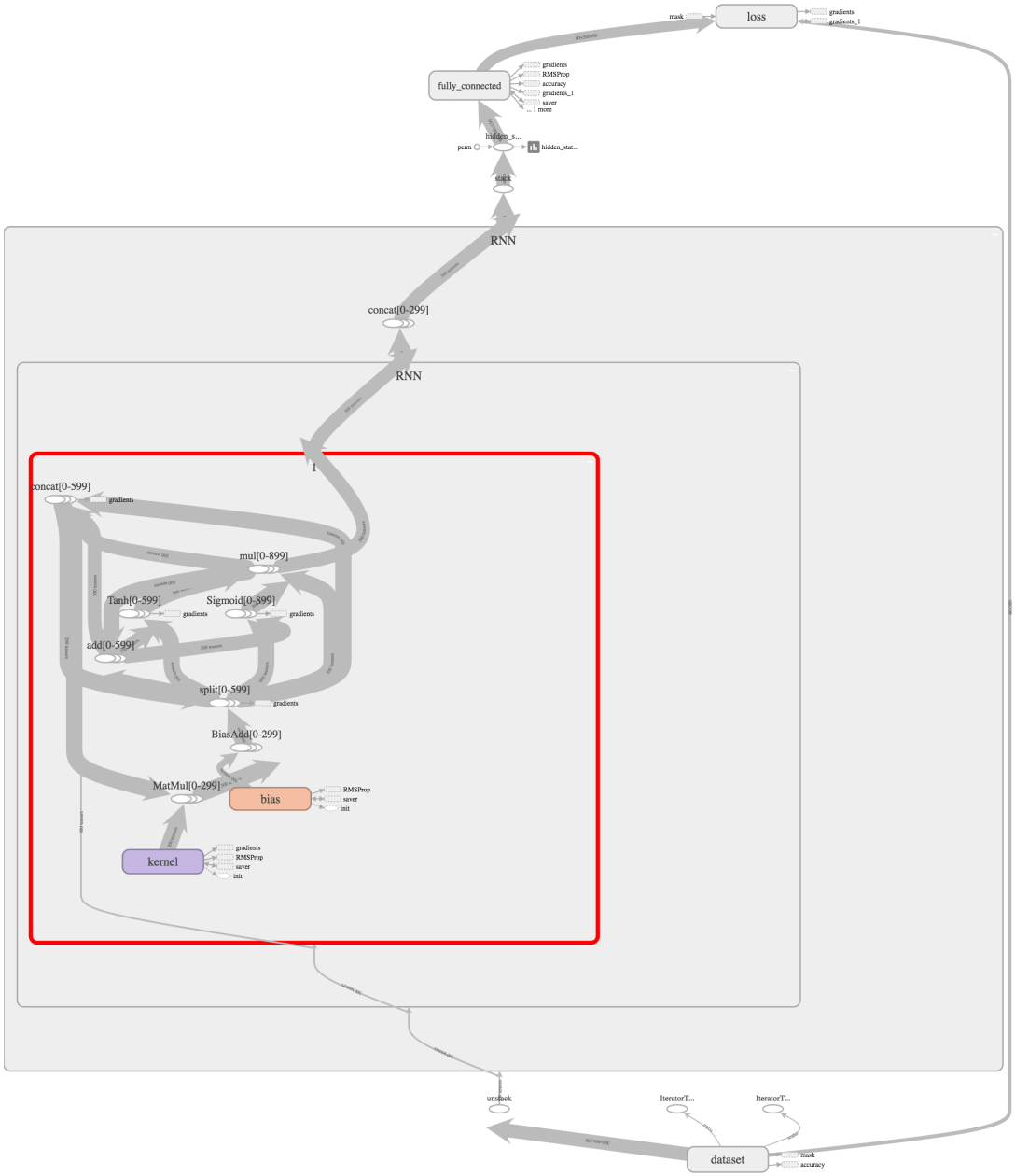

Figura 4.2 Grafo computacional de la arquitectura LSTM

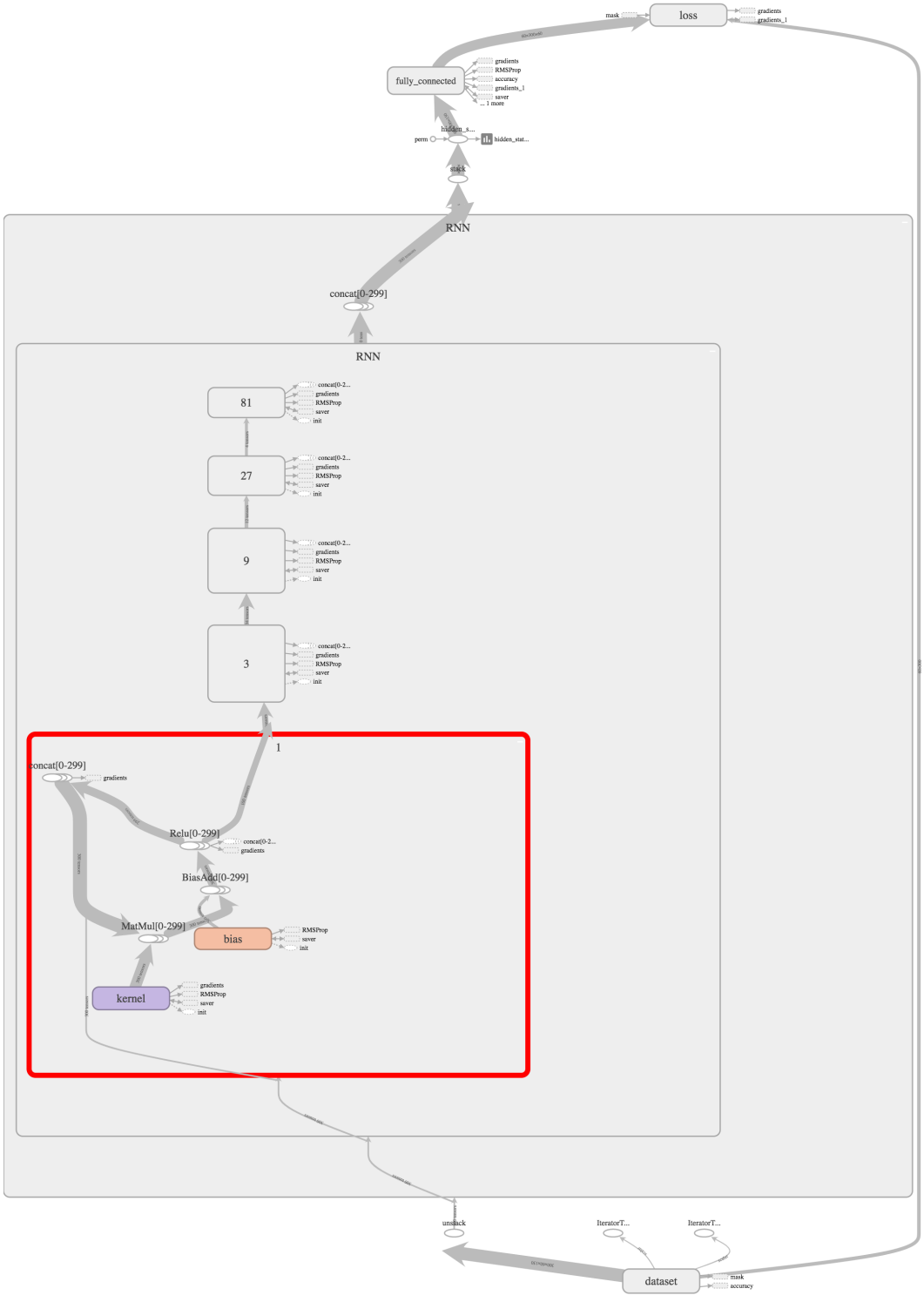

Figura 4.3 Grafo computacional de la arquitectura DRNN.

4.2 Resultados

En la iteración 100K de la **Figura 4.4**, D-RNN tiene 38,11% de clasificaciones correctas, LSTM tiene 43,89% y RNN tiene 17,24% de clasificación. Adicionalmente, para llegar a la iteración 100K, D-RNN demora 5h 16m 6s, LSTM demora 7h 23m 28s, y RNN 4h 46m 41s.

Los resultados analizan tres arquitecturas diferentes bajo el mismo esquema de optimización (i.e. learning rate, optimizador, tamaño del hidden layer, tamaño del batch de entrenamiento); y si recordamos el capítulo 1, podemos deducir que las funciones de error de los tres modelos son distintas. Por ende, los resultados son una comparación empírica, no rigurosa, entre los modelos.

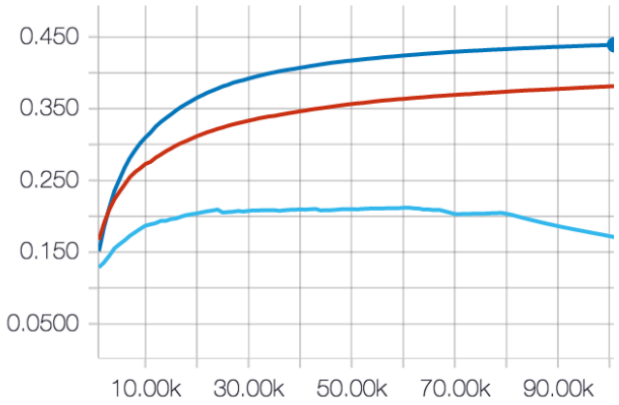

Figura 4.4 Porcentaje de clasificacioes correctas vs. Número de iteración: (Izq.), D-RNN (Rojo), LSTM (Azul), RNN (Celeste). El eje horizontal representa el número de iteraciones, el eje vertical el portentaje de clasificaciones correctas.

Para mitigar la invalidación de usar el mismo esquema de entrenamiento, se propone comparar el mejor modelo D-RNN encontrado, con los modelos RNN by LSTM encontrados por [4]. La comparación está en la Tabla 4.1.

Modelos/Modalidad	Cross View
RNN [4]	20,27%
D-RNN	48,10%
LSTM [4]	64,68%

Tabla 4.1 Porcentaje de clasificaciones correctas de tres modelos recurrentes: LSTM y RNN son evaluados en [4].

Con respecto al esquema de entrenamiento propuesto, en [4] RNN y LSTM aplican una transformación no-lineal a la capa de salida, D-RNN no; RNN y LSTM utilizan un batch size de tamaño 1000, D-RNN uno de tamaño 60; RNN y LSTM aplican *batch normalization* y *dropout*, D-RNN no. Todo lo demás es igual.

Empíricamente la transformación no-lineal en la última capa mejora la generalización del modelo. D-RNN no la implementa porque no es necesaria para garantizar a la red como aproximador universal.

El tamaño del batch tiene un impacto en la dirección del gradiente calculado, y por ende en la parametrización encontrada. Teóricamente, un tamaño de batch más grande produce una mejor representación de la función de error. Sin embargo, D-RNN no lo implementa porque requiere más memoria y aumenta el tiempo de entrenamiento.

“Dropout” y “Batch normalization” son soluciones empíricas para mejorar la capacidad de generalización de la red, D-RNN no la implementa porque no las podemos fundamentar teóricamente.

Comparando la parametrización, RNN utiliza 360150 parámetros, D-RNN utiliza 9150 parámetros, y LSTM utiliza 1441800 parámetros.

Con respecto a LSTM, D-RNN implementa un estado de 300 neuronas más, pero aun así requiere de menos parámetros. La clasificación de D-RNN es un 6% peor que la de LSTM. Con respecto a RNN, el estado recurrente es del mismo tamaño, y la clasificación es un 10% mejor en la iteración 21K.

Para validar si los parámetros impactan la generalización, se iteró una D-RNN con 6 unidades recurrentes, por un total de 27300 parámetros. Podemos

observar (Figura 4.5 - rosado), que con 1,89% de los parámetros de una LSTM y 11% más tiempo de entrenamiento, nos acercamos más al comportamiento.

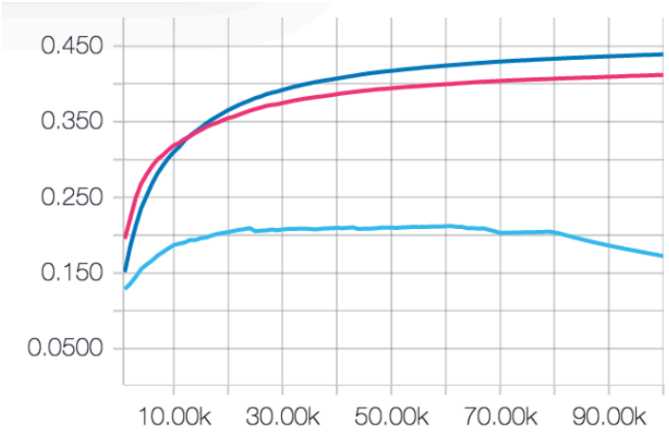

Figura 4.5 Porcentaje de clasificacioes correctas vs. Número de iteración: La D-RNN (en rosado) alcanza un 41,18% de clasificaciones correctas, la LSTM (43,89%) y la RNN(17,24%) son iguales a la Figura 1.

Adicionalmente, la Figura 4.6 revela, que a pesar de tener 48% de clasificaciones correctas, D-RNN clasifica bien bastantes de las salidas. Indagando un poco más, se descubre que D-RNN puede tardar la mitad de la secuencia en descifrar la actividad, y la otra mitad las clasifica bien. Esto implica que durante una parte de la secuencia D-RNN sabe lo que está sucediendo; y es la cantidad de instantes que no sabe lo que sucede, acumulados para todas las clases, los que causan una reducción en el porcentaje de clasificación.

Finalmente, se construyó una visualización de las secuencias procesadas por una DRNN, con 1143000 parámetros y 44% de clasificaciones correctas (**Figura 4.7**). Dentro de la visualización se observa que la red predice correctamente 80% de las actividades, reforzando la idea planteada por la **Figura 4.6**.

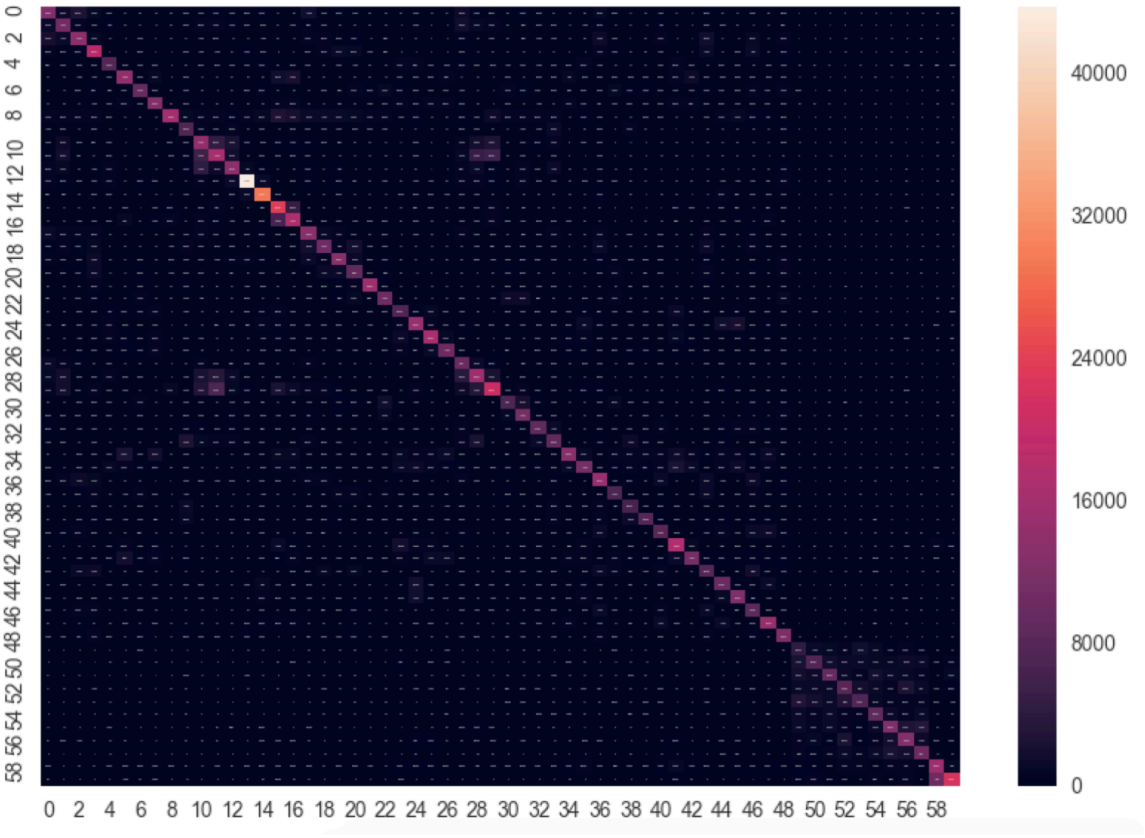

Figura 4.6 Matriz de confusión de D-RNN: El eje vertical representa la clase real y el eje horizontal la clase que la red predice. 80% de las actividades son clasificadas correctamente en algun punto de la secuencia.

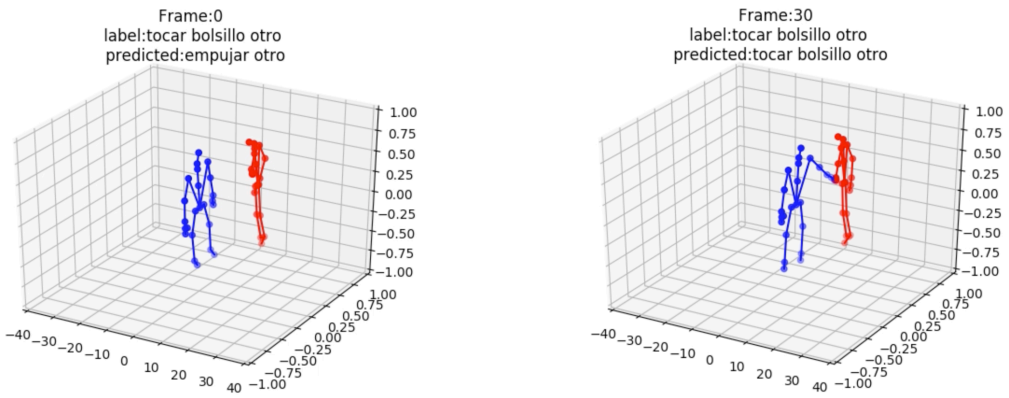

Figura 4.7 Dos frames de la actividad "tocar bolsillo de otra persona": *Label* es la etiqueta real de la actividad, *predicted* es la predicción de D-RNN.

CONCLUSIONES Y RECOMENDACIONES

La solución propuesta mejora la capacidad de generalización en un 17% con respecto a RNN, y empeora en un 6% con respecto a LSTM. Con respecto a memoria, D-RNN utiliza 0,63% de los parámetros que utiliza LSTM, y 2,5% de los parámetros de una RNN. En velocidad de convergencia, D-RNN tarda un 30% menos de tiempo que una LSTM y 10% más que una RNN. Consecuentemente, D-RNN hace un uso más eficiente de los parámetros, y es recomendable en arquitecturas que quieran minimizar consumo de memoria, a cambio de una pérdida en la generalización.

En conclusión, D-RNN es un modelo más simple que una LSTM pero con 6% más error de clasificación. Adicionalmente, logra reconocer correctamente 80% de las actividades presentadas; validandolo como una red neuronal con potencial práctico.

En trabajos futuros se recomienda utilizar más unidades recurrentes, para verificar si la arquitectura puede superar una LSTM en porcentaje de error. También se sugiere explorar el impacto empírico que las unidades recurrentes tienen en la derivada de la salida con respecto a todas las entradas, para confirmar que empíricamente nuestro modelo considera más entradas en su decisión. Finalmente, se sugiere comparer a D-RNN con otras arquitecturas que mitigan la inestabilidad de los gradientes.

BIBLIOGRAFÍA

[1] M. Nielsen, “Neural Networks and Deep Learning,” *Neural Networks and Deep Learning*. [Online]. Available: neuralnetworksanddeeplearning.com. [Accessed: 01-Aug-2017].

[2] I. Goodfellow, Y. Bengio, and A. Courville, *Deep learning*. Cambridge, Massachusetts: The MIT Press, 2016.

[3] A. Shahroudy, J. Liu, T.-T. Ng, and G. Wang, “NTU RGB+ D: A large scale dataset for 3D human activity analysis,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016, pp. 1010–1019.

[4] R. Zhao, H. Ali, and P. van der Smagt, “Two-Stream RNN/CNN for Action Recognition in 3D Videos,” *arXiv preprint arXiv:1703.09783*, 2017.

[5] R. Pascanu, T. Mikolov, and Y. Bengio, “On the difficulty of training recurrent neural networks,” in *International Conference on Machine Learning*, 2013, pp. 1310–1318.

[6] M. Abadi *et al.*, “Tensorflow: Large-scale machine learning on heterogeneous distributed systems,” *arXiv preprint arXiv:1603.04467*, 2016.

[7] C. Huyen, “CS 20: Tensorflow for Deep Learning Research,” Stanford, CA, USA, 12-Jan-2017.

[8] A. Graves, Supervised sequence labelling with recurrent neural networks. Heidelberg ; New York: Springer, 2012.

[9] C. Olah, “Understanding LSTM,” *colah’s blog*, 27-Aug-2015.

[10] Kenneth Tran, “Evaluation of Deep Learning Toolkits,” *github*, 06-Dec-2016. [Online]. Available: https://github.com/zer0n/deepframeworks. [Accessed: 25-Nov-2017].

[11] S. El Hihi and Y. Bengio, “Hierarchical recurrent neural networks for long-term dependencies,” in Advances in neural information processing systems, 1996, pp. 493–499.

[12] R. DiPietro, C. Rupprecht, N. Navab, and G. D. Hager, “Analyzing and Exploiting NARX Recurrent Neural Networks for Long-Term Dependencies.” 14-Jul-2017.

ANEXOS

A. DESMITIFICANDO TENSORFLOW

TensorFlow es un sistema de machine learning escalable, que opera en ambientes heterogéneos [6].

Escalable porque escala con la cantidad de recursos del sistema; y "opera en ambientes heterogéneos" porque puede ejecutar en diferentes dispositivos a la vez (e.g. CPU, GPU, TPU).

Los tres principios de diseño de TensorFlow son:

- Operadores primitivos dentro del grafo computacional: TensorFlow define todas las operaciones, desde multiplicación entre matrices hasta la actualización de los parámetros, como nodos dentro de un grafo computacional. TensorFlow luego distribuye los nodos entre todos los dispositivos disponibles; entre más dispositivos tenemos, menor es la carga sobre cada dispositivo y consecuentemente mayor es el *throughput* del sistema.
- Construcción y ejecución separados: El grafo computacional consta de *placeholders* para las entradas, *operations* para las operaciones, y *variables* para el estado del sistema. Todo grafo en TensorFlow es definido previo a su ejecución. Luego, TensorFlow aprovecha la definición completa del grafo para optimizar su ejecución.
- Abstracciones para diferentes dispositivos: Cada operador puede tener diferentes implementaciones para diferentes dispositivos (conocidos como *kernels*). Esto vuelve a TensorFlow capaz de operar en ambientes heterogéneos.

Los principios de diseño 1 y 2 son la guía principal al momento de construir una red neuronal: TensorFlow brinda los nodos para realizar las transformaciones que necesitamos, y la construcción y ejecución del grafo están separadas.

Los elementos principales para construir el grafo computacional son:

- Tensores: Son arreglos n-dimensionales donde los elementos comparten el mismo tipo de dato (e.g. int64, float32, string, uint8). Un tensor se caracteriza

por su grado (i.e. el número de dimensiones), su forma (i.e. el número de elementos en cada dimensión), y su tipo de dato.

- Operaciones: Son los nodos del grafo computacional. Representan las transformaciones de la red (e.g. multiplicación entre matrices, suma entre vectores,), y/o acciones que del sistema (e.g. guardar, cargar). Pueden recibir cero o más tensores como entrada, y su salida puede ser de cero o más tensores. Y su comportamiento puede cambiar en función de los parámetros que recibe.
- Operaciones con estado - Variables: Son un tipo de operaciones cuyo estado persiste en memoria. Cada operación de tipo variable tiene un buffer para almacenar su estado, y un identificador para leer y/o escribir al buffer. Este tipo de operaciones no tiene entrada.
- Operaciones con estado - Colas: TensorFlow implementa colas para coordinar la ejecución. Al igual que las variables, cada cola tiene un identificador para manipularla. Las colas implementan dos operaciones: encolar y desencolar. La operación encolar se bloquea si la cola está llena, y la operación desencolar se bloquea si no tenemos entradas disponibles. Los utilizaremos para separar el pre-procesamiento de los datos, y la ejecución del modelo.

En resumen, utilizaremos las colas para almacenar el dataset, las variables para almacenar los parámetros del modelo, las operaciones para representar las transformaciones, y los tensores serán las salidas y entradas de cada nodo.

Para explicar TensorFlow comenzaremos implementando una red neuronal, luego implementaremos una red neuronal recurrente, y finalmente implementaremos la solución propuesta en el capítulo 2 (i.e. D-RNN). En cada implementación se detalla la idea detrás del código. Los archivos explicados están en el github del proyecto.

Documentación - ANN.py

```
In [13]: import tensorflow as tf
```

Como mencionamos previamente, todo programa en TensorFlow se compone de dos secciones: El grafo computacional y la sesión de ejecución.

El grafo computacional representa el modelo como una DAG (*Directed acyclic graph*), donde los nodos son operaciones y los arcos representan tensores. Para el caso de las redes neuronales, las operaciones (nodos) corresponden a las transformaciones, y los arcos corresponden a los resultados de las transformaciones.

Antes de comenzar la implementación vamos a plantear dos ajustes.

Primero, la entrada no la vamos a definir como un vector de dos filas y una columna, como el modelo matemático, pero como una matriz de una fila y dos columnas. Este ajuste es porque TensorFlow indexa los vectores horizontalmente, y no verticalmente. Por ejemplo, el vector [[1,2,3]] tiene tres columnas y una fila (expresión en TensorFlow), no tres filas y una columna (expresión matemática).

Una forma de ajustar el modelo matemático a la indexación de TensorFlow es transponiendo las ecuaciones. De ahora en adelante, implementamos la forma transpuesta de cada ecuación.

$$\boldsymbol{h^T} = \phi(\boldsymbol{x^T} U^T + \boldsymbol{b^T})$$

Segundo, en vez de procesar una muestra a la vez (matemáticamente con un vector), procesamos varias muestras a la vez (matemáticamente una matriz de entrada donde las columnas representan muestras). Podemos entonces procesar el dataset con una sola operación entre matrices. Es decir:

$$H = [\boldsymbol{h_1}, \boldsymbol{h_2}, \ldots, \boldsymbol{h_n}] = \phi(U[\boldsymbol{x_1}, \boldsymbol{x_2}, \ldots, \boldsymbol{x_n}] + \boldsymbol{b}) \text{ s.t. } H \in R^{mxn}$$

Donde n es el número de muestras y m es el tamaño del hidden layer.

Teóricamente, b también debería transformarse en una matriz (mxn), donde todas las columnas son iguales. Sin embargo, TensorFlow automáticamente hace esta transformación cuando operamos la suma entre una matriz y un vector. A esta transformación se la conoce como broadcasting. Adicionalmente, la función de error la construimos con todas las muestras del dataset. Empiricamente, esto resulta muy costoso computacionalmente. Una solución es procesar una muestra del dataset y esperar que esta muestra aproxime bien a la

función de error. En la literatura, a la muestra se la conoce como *batch* y al número de muestras como batch size.

Las entradas de la red las instanciamos con un nodo tf.placeholder. tf.placeholder es un nodo cuyo valor no está definido hasta la ejecución. Es el nodo que se utiliza como entradas y etiquetas, ya que estos son provistos por el dataset.

In [ ]:
```
x = tf.placeholder(shape=(1,2),name='x',dtype=tf.float32)
y = tf.placeholder(shape=(1,2),name='y',dtype=tf.float32)
```

shape define la forma del tensor; *name* es el nombre que le vamos a dar a la operación; y dtype es el tipo de dato de los elementos del tensor. *name* no es necesario de definir, sin embargo, es el identificador que tendrá la operación (se recomienda definirlo).

Los parámetros de una red neuronal, usualmente son inicializados siguiendo una distribución normal (i.e. cada uno de sus elementos sigue una distribución normal). La inicialización impacta el punto de donde comenzamos a minimizar y la velocidad con la que convergeremos a una solución.

In [3]:
```
init_val_U  = tf.random_normal(shape=[2,3],name='init_U',dtype=tf.float32)
init_val_b = tf.random_normal(shape=[1,3],name='init_b',dtype=tf.float32)
```

Completamos el hidden layer definiendo las variables y las transformaciones:

- tf.Variable corresponde a los parámetros del modelo.
- tf.matmul corresponde a la transformación lineal entre la entrada y los parámetros.
- tf.sigmoid corresponde a la transformación no lineal ϕ.

In [ ]:
```
U = tf.Variable(initial_value=init_val_U,name='U',dtype=tf.float32)
b = tf.Variable(initial_value=init_val_b,name='b',dtype=tf.float32)
h = tf.sigmoid(tf.matmul(x,U)+b)
```

La capa de salida es una transformación afín. Nuevamente definimos los nodos de inicialización, los parámetros de la capa, y la transformación de salida.

$$\boldsymbol{o^T} = \boldsymbol{h^T} V^T + \boldsymbol{c^T}$$

```
In [5]: init_val_V = tf.random_normal(shape=[3,2],name='init_V',dtype=tf.float32)
        init_val_c = tf.random_normal(shape=[1,2],name='init_c',dtype=tf.float32)

        V = tf.Variable(initial_value=init_val_V,name='V',dtype=tf.float32)
        c = tf.Variable(initial_value=init_val_c,name='c',dtype=tf.float32)

        o = tf.sigmoid(tf.matmul(h,V)+c)
```

Ya tenemos la red neuronal completa. El siguiente paso es definir los nodos de entrenamiento: La función de error, el cálculo de los gradientes, y la aplicación de los gradientes.

TensorFlow implementa varias funciones de error (e.g. softmax_cross_entropy, mean_squared_error, log_loss). Por ahora utilizaremos mean_square_error.

$$\ell = \frac{1}{n}\sum_{i=1}^{n}(\boldsymbol{y}-\boldsymbol{o})^2$$

```
In [18]: loss = tf.losses.mean_squared_error(y,o)
```

Los nodos para calcular los gradientes y aplicar descenso del gradiente son creados por un optimizador. TensorFlow tiene implementaciones de varios optimizadores (e.g. Adagrad, Adam, Adadelta, RMSProp). Lo único que tenemos que hacer es instanciar el optimizador, y llamar el método minimize. Esto añade todos los nodos asociados al cálculo y aplicación de los gradientes.

```
In [ ]: learning_rate = 0.01
        train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
```

Con esto completamos la definición del grafo computacional. Ahora continuamos con la sesión de ejecución.

La ejecución de los nodos y la evaluación de los tensores ocurre dentro de un ambiente llamado session. Este ambiente prepara los recursos para evaluar el grafo computacional. Cuando ejecutamos un nodo, se evaluan todos los nodos necesarios para generar la salida del nodo en ejecución. Por ejemplo, si ejecutamos el nodo h, se ejecutarán los nodos tf.matmul, tf.add (por la suma) y tf.placeholder. Las entradas de los placeholders las pasamos como un diccionario, donde la clave corresponde al placeholder.

Las variables no se inicializan hasta hacerlo explicitamente. Para inicializar todas las variables tenemos que ejecutar un nodo tf.global_variables_initializer.

El objeto de tipo session lo denominamos sess, e implementa el método run para ejecutar el grafo. run recibe dos parámetros: la lista de nodos que deseamos correr y un diccionario. El diccionario se lo utiliza para indicar los valores que toman los nodos tf.placeholder.

A continuación probamos la red con una entrada y etiqueta de prueba.

```
In [ ]: dataset_x = [[1,2]] # vector de entrada de la red.
        dataset_y = [[1,0]] # etiqueta de salida de la red.
```

```
In [ ]: with tf.Session() as sess:
            init = tf.global_variables_initializer()
            sess.run(init)
            sess.run(train_step,feed_dict={x:dataset_x,y:dataset_y})
```

Con esto completamos la implementación de una red neuronal en TensorFlow. Ahora continuamos con la implementación de una red neuronal recurrente.

Documentación - RNN.py

```
In [1]: import tensorflow as tf
```

Como se menciona en el capítulo uno, la diferencia entre una red neuronal normal y una red neuronal recurrente es la unidad de recurrencia.

La unidad de recurrencia se construye añadiendo un parámetro recurrente al hidden layer. De esta forma convertimos la ecuación (2.3) en la ecuación (2.24).

$$\boldsymbol{h} = \phi(U\boldsymbol{x} + \boldsymbol{b}) \tag{2.3}$$

$$\boldsymbol{h}^{(t)} = \phi(W\boldsymbol{h}^{(t-1)} + U\boldsymbol{x}^{(t)} + \boldsymbol{b}) \tag{2.24}$$

La solución en TensorFlow parece de simplemente añadir una variable más (W). Pero ¿Cómo conectamos una salida en (t-1) con la entrada en (t)?

La respuesta es implementando todo el grafo que se forma al desplegar la red neuronal recurrente en el tiempo. Donde cada nodo recurrente consiste en ejecutar la misma operación pero con los parámetros h en (t-1) y x en (t).

Primero definimos las entradas y etiquetas de la red (x,y). A diferencia del modelo anterior, ahora las entradas tienen tres dimensiones. (tamaño de la secuencia, tamaño del batch, tamaño del vector de entrada).

In [2]:
```
x = tf.placeholder(shape=(3,1,2),name='x',dtype=tf.float32)
y = tf.placeholder(shape=(3,1,2),name='y',dtype=tf.float32)
hidden_size = 3 # Tamaño del hidden layer
```

El lector podría pensar que un ordenamiento más lógico para la entrada sería (tamaño del batch, tamaño de la secuencia, tamaño del vector de entrada). Sin embargo, la naturaleza secuencial del modelo obliga que el calculo de los estados recurrentes se haga en orden. Es decir, primero calculo el estado recurrente (h) en (t-1), luego (h) en (t), y luego (h) en (t+1).

Con esto en mente, que tal si calculo el estado recurrente (h) en (t-1), no con una secuencia, pero con un batch de secuencias. Es decir, uso la matriz (batch size, vector de entrada en (t)) como entrada para el estado recurrente (t). Entonces, la salida de cada estado recurrente es una matriz, donde cada columna es el estado recurrente en el instante (t) de diferentes secuencias.

Al igual que ANN.py, procesamos varias secuencias a la vez, la diferencia es que cada matriz (H) corresponde a un instante de tiempo. A continuación mostramos la ecuación:

$$H = \left[\boldsymbol{h}_1^{(t)}, \dots, \boldsymbol{h}_n^{(t)}\right] = \phi\left(W\left[\boldsymbol{h}_1^{(t-1)}, \dots, \boldsymbol{h}_n^{(t-1)}\right] + U\left[\boldsymbol{x}_1^{(t)}, \dots, \boldsymbol{x}_n^{(t)}\right] + \boldsymbol{b}\right)$$

$$\text{s.t.} \qquad H \in R^{mxn}$$

Donde n es el número de secuencias en el batch (denominado batch size) y m es la longitud de las secuencias.

Para crear la unidad recurrente (i.e. ecuación 2), el siguiente paso seria crear las variables {W, U, b} y la transformación no-lineal. TensorFlow implementa una clase que crea la unidad recurrente en una RNN con todos los parámetros necesarios.

La clase recibe como parámetro el tamaño de la capa recurrente y la función de activación. Una vez instanciada, facilita un método para ejecutar la unidad recurrente en el grafo computacional. Este método recibe como parámetros el vector de entrada en el instante (t) y el estado anterior de la red (i.e. h(t-1)).

```
In [3]: cell = tf.contrib.rnn.BasicRNNCell(hidden_size,activation=tf.nn.relu)
```

$$\boldsymbol{h}^{(t)} = \phi(W\boldsymbol{h}^{(t-1)} + U\boldsymbol{x}^{(t)} + \boldsymbol{b}) \qquad (2.24)$$

Como los tensores no se pueden iterar, tf.unstack es una operación que convierte el tensor de entrada en una lista. El parámetro axis es la dimensión que convertiremos en lista. El resultado es una lista de tensores con la forma de las dimensiones que quedaban, en este caso: (tamaño del batch, tamaño de la

```
In [4]: list_x = tf.unstack(x,axis=0)
```

entrada).

Procedemos a iterar la lista con un lazo for. En cada iteración, llamamos la unidad recurrente, con el tensor de entrada x_t y el estado anterior. La función cell() retorna una tupla que utilizaremos como entrada de la siguiente iteración. Finalmente, concatenamos la salida de cada instante (t) a una lista vacia output.

```
In [5]: output = []
        state = tf.zeros([1,hidden_size],tf.float32)
        for x_t in list_x:
            state, _ = cell(x_t,state)
            output.append(state)
```

La salida output es una lista. La convertimos a tensor utilizando tf.stack. Este método junta varios tensores con respecto a una dimensión. En nuestro caso, queremos juntar los tensores con respecto a la primera dimensión (i.e. axis=0).

```
In [6]: hidden_states = tf.stack(output,axis=0,name='stack_output')
```

Finalmente, la capa de salida es una transformación afín. Nuevamente definimos los nodos de inicialización, los parámetros de la capa, y la transformación de salida. Recordemos que utilizaremos la versión transpuesta.

$$\left(\boldsymbol{o}^{(t)}\right)^T = \left(\boldsymbol{h}^{(t)}\right)^T V^T + \boldsymbol{c}^T$$

```
In [7]: init_val_V = tf.random_normal(shape=[3,2],name='init_V',dtype=tf.float32)
        init_val_c = tf.random_normal(shape=[1,2],name='init_c',dtype=tf.float32)

        V = tf.Variable(initial_value=init_val_V,name='V',dtype=tf.float32)
        c = tf.Variable(initial_value=init_val_c,name='c',dtype=tf.float32)
```

La variable V es una matriz (i.e. un tensor de rango 2) y la entrada es un tensor de rango 3. Para multiplicarlos los tensores operandos deben tener el mismo rango. tf.reshape nos permite cambiar la forma de un tensor para convertirlo de un rango a otro.

Como la transformación afín de la salida es la misma para todos los instantes (t), de todas las secuencias (batch size), convertiremos el tensor output_tensor de forma (3,1,2) en forma (-1,2). La forma (-1,2) aplana la entrada mantienendo la forma de la última dimensión (i.e. el tamaño del vector de estados recurrentes).

Después de tf.reshape tendremos un tensor con n filas (representando cada instante de cada secuencia) y dos columnas.

```
In [8]: reshape_hidden_states = tf.reshape(hidden_states, [-1,3])
```

Ahora procedemos a hacer la multiplicación entre matrices:

```
In [9]: o = tf.matmul(reshape_hidden_states,V)+c
```

Para poder calcular el error entre la salida de la red y la etiqueta ambos tienen que tener la misma forma. Reformamos el tensor de salida de la red para calcular el error.

```
In [10]: o_reshape = tf.reshape(o,[3,1,2])
```

Finalmente, el último paso es crear los nodos de entrenamiento.

```
In [11]: loss = tf.losses.mean_squared_error(y,o_reshape)
```

```
In [12]: learning_rate = 0.01
         train_step = tf.train.GradientDescentOptimizer(learning_rate).minimize(loss)
```

Completamos el programa con una session de ejecución de prueba.

```
In [13]: dataset_x = [[[1,1]],[[2,2]],[[1,1]]] # secuencia de entrada de la red
         dataset_y = [[[1,0]],[[0,1]],[[0,1]]] # secuencia de salidas
```

```
In [14]: with tf.Session() as sess:
             init = tf.global_variables_initializer()
             sess.run(init)
             sess.run(train_step,feed_dict={x:dataset_x,y:dataset_y})
```

Documentación – DRNN.py

La única diferencia entre RNN y DRNN son las unidades recurrentes. RNN tiene una unidad recurrente, mientras que DRNN tiene tres. Adicionalmente, las conexiones entre las unidades recurrentes de DRNN son más complejas que las de RNN. Por este motivo, vamos a hacer uso de dos elementos de TensorFlow: *Scopes*, para crear nuevas variables cada vez que creamos una nueva unidad recurrente; y diccionarios, para almacenar los estados recurrentes anteriores y conectar correctamente las neuronas.

In [5]:
```
with tf.variable_scope('RNN', reuse=None):

    cell = {}
    out = {}
    outputs = []

    for key in net_arch:
        with tf.variable_scope('{}'.format(key)):
            cell[key] = tf.contrib.rnn.BasicRNNCell(hidden_size,activation=tf.nn.relu)

    state = {key:tf.zeros([batch_size,hidden_size],dtype=tf.float32) for key in net_arch}

    for count,batch in enumerate(list_x):
        for key in net_arch:
            if key==1:
                out[key], state[key] = cell[key](batch,state[key],'RNN/{}'.format(key))
            elif (count % key)==0:
                out[key], state[key] = cell[key](out[previous_key],state[key],'RNN/{}'.format(ke
            previous_key = key

        output = tf.concat([out[key] for key in net_arch],axis=1)
        outputs.append(output)
```

El resto del código es similar.

A continuación el lector puede observar los grafos computacionales generados por las tres arquitecturas. Se recomienda que los revise para que observe la conexión entre operaciones.

El código provisto en github es más complejo porque incluye: colas para manejar las entradas, almacenamiento y recuperación automática de modelos, manejo de ambientes, y almacenamiento de metadata. Si el lector desea aprender más al respecto se recomienda: tensorflow.org

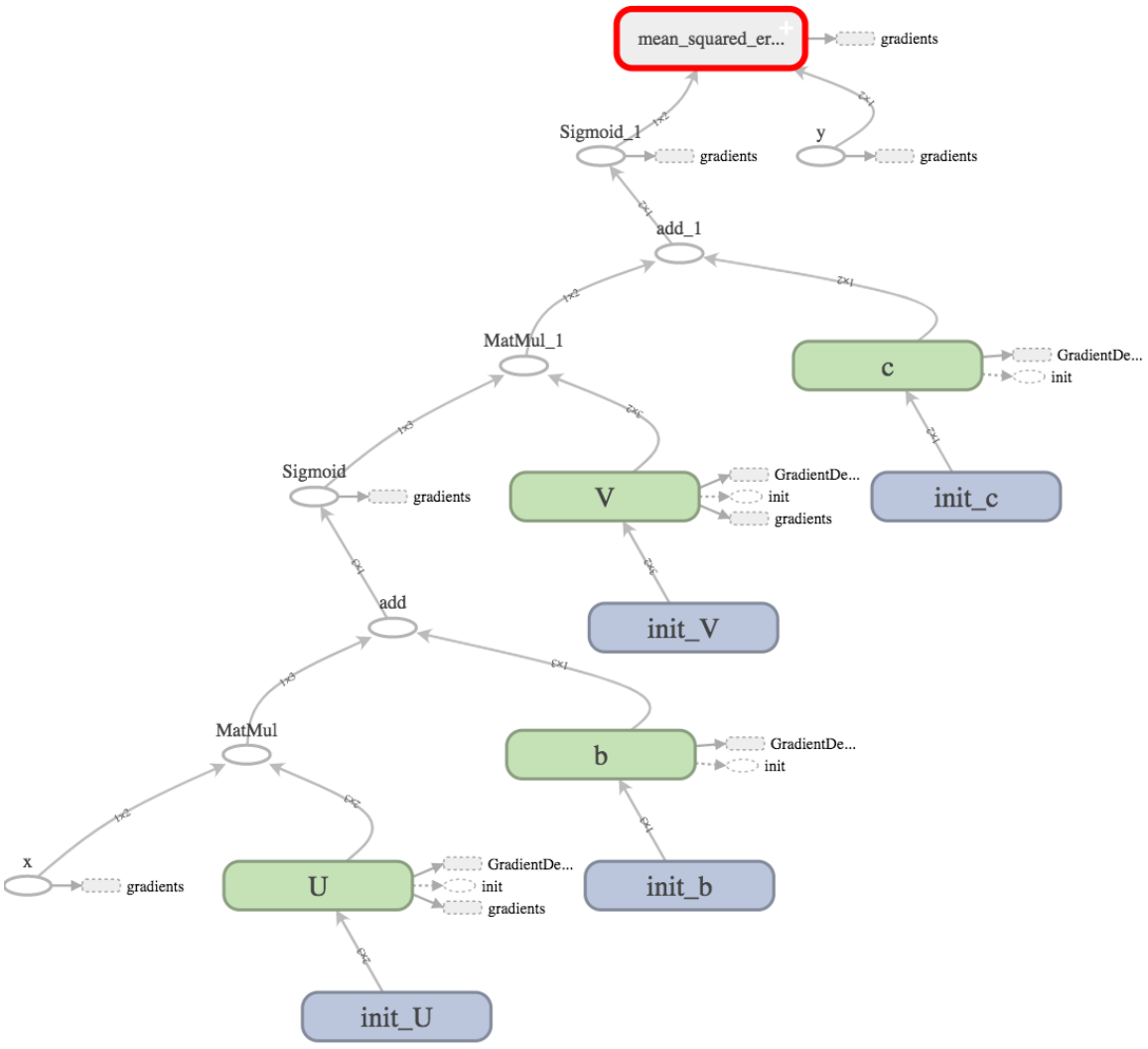

Figura A.1 Grafo computacional generado por ANN.py

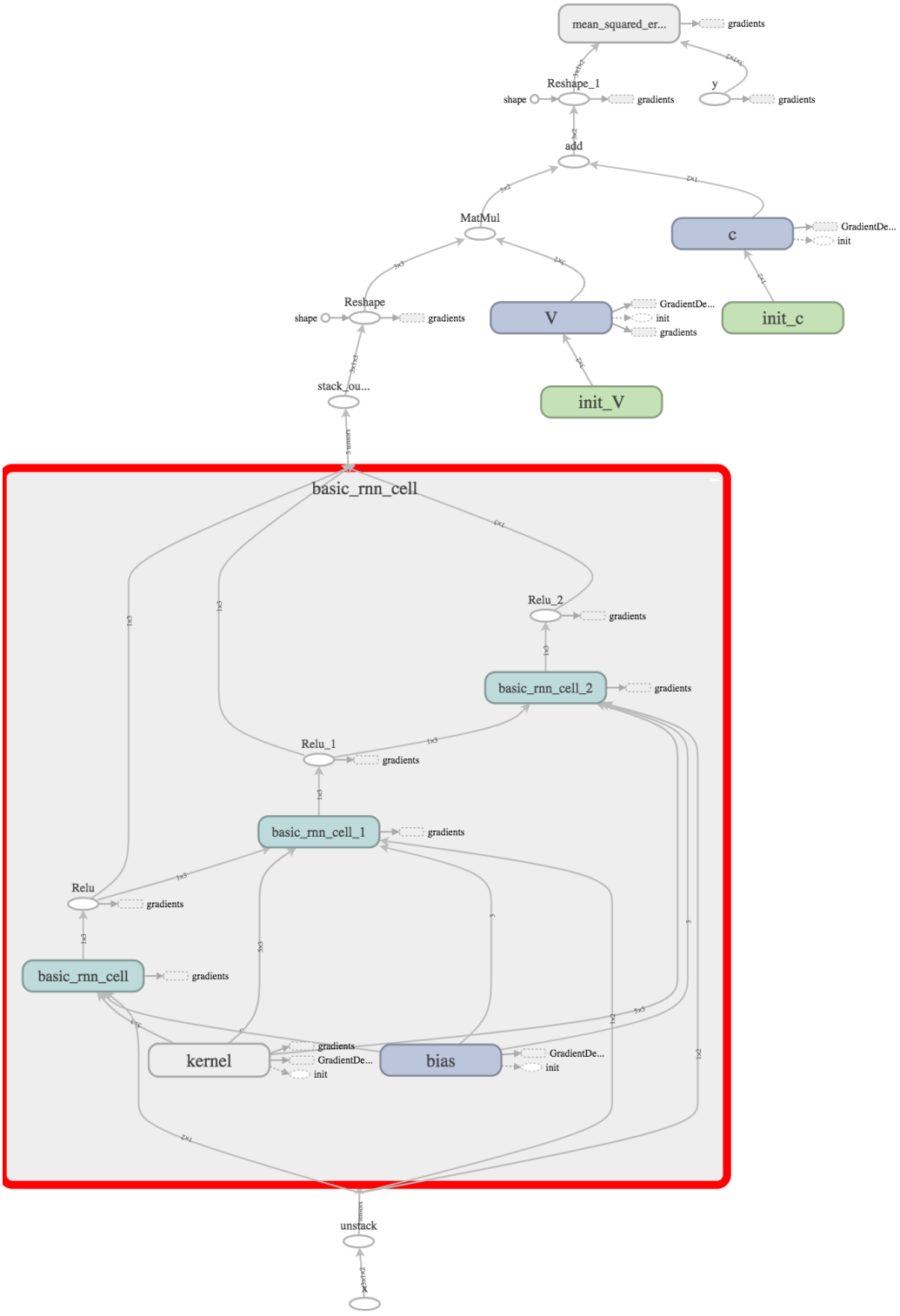

Figura A.2 Grafo computacional generado por RNN.py

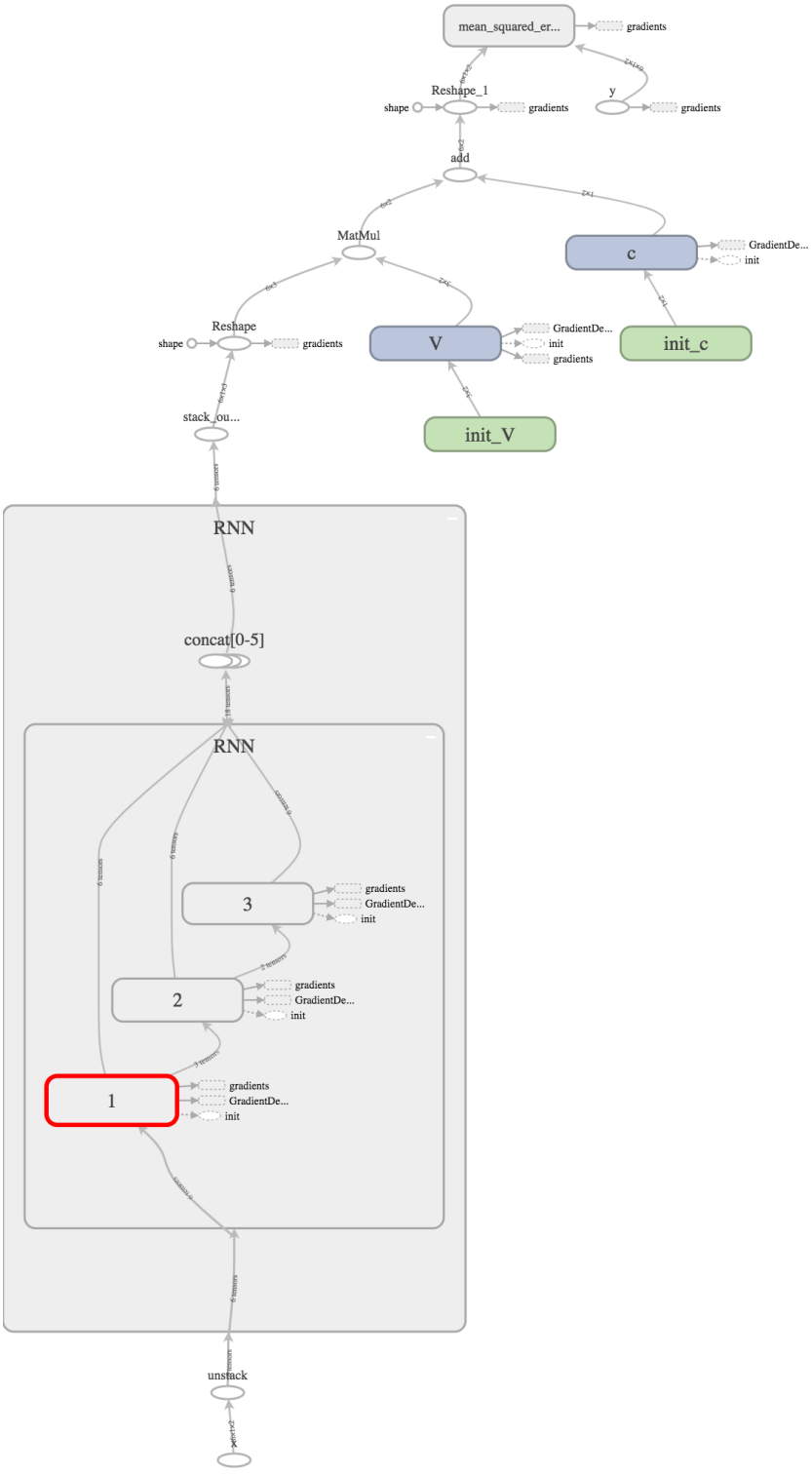

Figura A.3 Grafo computacional generado por DRNN.py