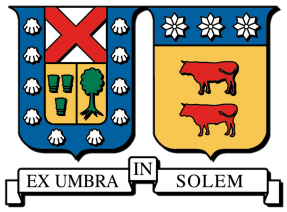
UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
SANTIAGO - CHILE



"EVALUACIÓN DE FRAMEWORKS DE SOFTWARE PARA LA IMPLEMENTACIÓN DE REDES NEURONALES ARTIFICIALES"

SEBASTIÁN ALEJANDRO VERGARA MIRANDA

MEMORIA DE TITULACIÓN PARA OPTAR AL TÍTULO DE INGENIERO CIVIL INFORMÁTICO

PROFESOR GUÍA: **RICARDO ÑANCULEF**
PROFESOR CORREFERENTE: **JUAN CARLOS CALVO**

JULIO - 2018

Agradecimientos

A mis padres y familiares, por su permanente apoyo y preocupación durante este proceso. Por acompañarme tanto en mis momentos de luz, como de oscuridad.

A mis amigos, por los buenos momentos compartidos. Por tener el tiempo, la disposición y la paciencia de ayudarme cuando lo necesité.

Al profesor Ricardo Ñanculef, por guiarme y proporcionarme los recursos necesarios para el desarrollo de esta memoria.

A Forest Agostinelli, Dan Ciresan, Matthieu Courbariaux y Chen-Yu Lee por tener la amabilidad de responder mis consultas en relación a sus publicaciones.

Resumen

Una red neuronal artificial puede ser definida como un paradigma de aprendizaje automático en que una tarea se descompone en tareas más simples denominadas capas o niveles de procesamiento. En las últimas décadas, las redes neuronales artificiales han permitido la obtención de importantes avances en problemas de visión computacional, procesamiento de lenguaje natural, análisis de sentimiento, reconstrucción de datos, entre otros. La creciente popularidad de las redes neuronales ha llevado a la creación de una amplia variedad de frameworks de software para su implementación. Sin embargo, el costo computacional asociado al entrenamiento de redes neuronales es, por lo general, bastante alto, por lo se que hace necesario contar con hardware que ayude a solventar este problema. Actualmente, la GPU es entendida como el recurso computacional más apropiado para trabajar con redes neuronales de manera eficiente. No obstante, el gran número de frameworks disponibles lleva a que los usuarios no puedan escoger con claridad la herramienta que mejor se ajusta a sus necesidades. En esta memoria, se evalúa el rendimiento de tres frameworks de software, Theano, TensorFlow y CNTK en la implementación de redes feedforward, redes convolucionales y redes recurrentes, tanto en términos temporales como espaciales. Para acceder a las funcionalidades de estos frameworks, se trabaja con el wrapper Keras, por lo que también se evalúa su facilidad de uso y su flexibilidad. A partir de los experimentos realizados, se determina que Theano presenta un excelente rendimiento en la implementación de redes feedforward y redes recurrentes, a la vez que TensorFlow logra un rendimiento sobresaliente sobre redes convolucionales y CNTK obtiene buenos rendimientos sobre los tres tipos de arquitecturas estudiadas.

Palabras Clave: Deep learning, red neuronal artificial, feedforward, convolucional, recurrente, LSTM, GPU, framework, Keras, Theano, TensorFlow, CNTK.

Abstract

An artificial neural network can be defined as a paradigm of automatic learning in which a task is decomposed into simpler tasks called layers or processing levels. In recent decades, artificial neural networks have allowed the achievement of important advances in computer vision, natural language processing, sentiment analysis and data reconstruction problems, among others. The growing popularity of neural networks has led to the creation of a wide variety of software frameworks for their implementation. However, the computational cost arising in neural network training is, in general, quite high, so it is necessary to have hardware to help solve this problem. Currently, the GPU is understood as the most appropriate computational resource to work with neural networks efficiently. However, the large number of available frameworks means that users can not clearly choose the tool that best suits their needs. In this report, the performance of three software frameworks, Theano, TensorFlow and CNTK, is evaluated in the implementation of feedforward networks, convolutional networks and recurrent networks, both in temporal and spatial terms. To access the functionalities of these frameworks, The Keras wrapper is used, so its ease of use and flexibility are also evaluated. From the experiments carried out, it is determined that Theano presents an excellent performance in the implementation of feedforward networks and recurrent networks, while TensorFlow achieves an outstanding performance over convolutional networks and CNTK obtains good performances on the three types of architectures studied.

Keywords: Deep learning, artificial neural network, feedforward, convolutional, recurrent, LSTM, GPU, framework, Keras, Theano, TensorFlow, CNTK.

Glosario

- **Adadelta**: Método de optimización análogo a RMSprop, donde la tasa de aprendizaje es reemplazada por el RMSE acumulado de los parámetros de la red hasta el epoch previo.
- **Adam**: Método de optimización que actualiza cada parámetro de una red neuronal con una tasa de aprendizaje en particular, la cual es adaptada en forma independiente de las demás.
- **Agrupamiento**: Parámetro que limita la conectividad entre los filtros de una capa convolucional y los inputs recibidos por esta, permitiendo así reducir la utilización de los recursos disponibles.
- **Backward pass**: Proceso de actualización de los parámetros de una red neuronal, desde la última capa de dicha red hasta la primera.
- **Bias**: Parámetro constante que es sumado al resultado de una operación lineal aplicada sobre otros parámetros de la red.
- **BN**: Siglas de Batch Normalization. Procedimiento que normaliza todas las activaciones de una capa, ya sea por filas o por columnas, manteniendo el promedio cercano a 0 y la desviación estándar cercana a 1. Este procedimiento es ejecutado para cada mini-batch por separado.
- **Capa**: Conjunto de neuronas.
- **CNN**: Iniciales de Convolutional Neural Network (red neuronal convolucional, en español).
- **CUDA**: Siglas de Compute Unified Device Architecture. Plataforma y modelo de programación creado por NVIDIA para el desarrollo de aplicaciones de compu-

tación paralela en GPU.

- **cuDNN**: Extensión de CUDA que provee primitivas para la implementación de redes neuronales.
- **Dropout**: Técnica de regularización que desactiva cierta fracción de las neuronas de una capa.
- **Epoch**: Ejecución de un forward pass más un backward pass sobre cada uno de los ejemplos del conjunto de entrenamiento.
- **Fase de entrenamiento**: Construcción de un modelo predictivo a partir de un conjunto de ejemplos ya conocidos.
- **Fase diferencial específica**: Variación en la diferencia de fase entre la componente horizontal polarizada y vertical polarizada de un pulso dentro de un rango determinado.
- **Fase de prueba**: Evaluación del modelo construido en la fase de entrenamiento sobre ejemplos desconocidos.
- **Flujos de ejecución**: Conjunto de instrucciones idénticas aplicadas sobre datos diferentes.
- **Forward pass**: Propagación de un ejemplo a través de una red neuronal y generación de una respuesta para dicho ejemplo.
- **Función de activación**: Transformación no lineal aplicada sobre el resultado de una operación lineal. La salida de una función de activación es denominada activación.
- **Función de pérdida**: Expresión que permite cuantificar que tan bien aprende una red neuronal, por medio de la comparación de las respuestas generadas por la misma en contraste con las respuestas correctas. Mientras menor es su valor, mejor es el aprendizaje de la red.
- **GCN**: Siglas de Global Contrast Normalization. Técnica de pre-procesamiento en la que, opcionalmente, se resta el promedio por columnas de un conjunto de datos. Luego, este es dividido por la desviación estándar por filas del mismo. En el caso de imágenes multicanal, se emplea el procedimiento anterior en cada canal por separado.

- **GPU**: Siglas de Graphics Processing Unit. Dispositivo diseñado para el procesamiento de contenido gráfico.
- **Hard sigmoid**: Variante de la función de activación sigmoidal, dada por:

$$f(x) = max(0, min(1, \frac{x+1}{2}))\,. \tag{0.1}$$

- **Hiper parámetro**: Variable asociada a una red neuronal cuyo valor no puede ser aprendido a partir del entrenamiento de la misma y que puede ser fijado manualmente o a través de heurísticas.
- **HMDIR-II**: Siglas de *How Much Did It Rain? II*, problema de time series forecasting.
- **Inicialización de parámetros**: Criterio para determinar cuales deben ser los valores iniciales de los parámetros de una red neuronal.
- **Iteración**: Ejecución de un forward pass más un backward pass sobre cada ejemplo de un mini-batch.
- **Leaky ReLU**: Variante de la función de activación ReLU, dada por:

$$f(x) = \begin{cases} \alpha x & \text{si } x < 0, \alpha \geq 0 \\ x & \text{si } x \geq 0 \end{cases} \tag{0.2}$$

- **LRN**: Siglas de Local Response Normalization. Procedimiento que normaliza la salida de una capa convolucional en caso de que la función de activación sea ReLU.
- **Método de optimización**: Algoritmo que determina los parámetros óptimos de una red neuronal, esto es, aquellos que permiten minimizar la función de perdida.
- **Mini-batch**: Subconjunto de ejemplos de entrenamiento que son procesados en cada iteración.
- **Momentum**: Hiper parámetro opcional usado en conjunto con SGD para evitar que el proceso de optimización de la función de pérdida se estanque en un óptimo local. Su valor puede situarse entre 0 y 1.

- **Neurona**: Unidad básica para la construcción de una red neuronal que aplica una transformación lineal sobre los inputs que recibe.
- **Outlier**: Dato que posee uno o más atributos cuyos valores son muy distantes respecto a la mayoría de las observaciones restantes.
- **Overfitting**: En español sobreajuste. Fenómeno que se presenta cuando un modelo presenta excelentes resultados sobre los ejemplos de entrenamiento, pero un pobre rendimiento sobre ejemplos de prueba.
- **Parámetro**: Valor asociado a una red neuronal que es determinado a partir del entrenamiento de la misma.
- **Percentil**: Valor de un atributo determinado que es menor para cierto porcentaje de los datos de una muestra. Por ejemplo, si se consideran una muestra de personas, el atributo *edad* y que el valor del percentil 30 para dicho atributo es 25 [año], quiere decir que el 30 % de las personas tienen una edad menor a 25 [año], mientras que el 75 % tiene una edad superior a los 25 [año].
- **Peso**: Parámetro de una red neuronal que cuantifica la importancia de la relación entre dos neuronas. Mientras mayor es su valor, mayor es la importancia de la conexión.
- **Pre-procesamiento**: Transformación que sitúa en una misma escala tanto a los ejemplos utilizados para el entrenamiento como para la evaluación de una red neuronal, buscando así que su habilidad de aprender mejore.
- **Reflectividad**: Medida de la fracción de radiación reflejada sobre una superficie, expresada como el cociente entre la energía de la radiación reflejada y la cantidad total de energía incidente en dicha superficie.
- **Reflectividad diferencial**: Diferencia entre la reflectividad horizontal y la reflectividad vertical en una superficie.
- **Regularización**: Aplicación de una o más restricciones sobre los parámetros de un modelo durante la fase de entrenamiento para evitar que este sufra de overfitting.
- **ReLU**: Siglas de la función de activación Rectified Linear Unit, dada por:

$$f(x) = max(0, x) \tag{0.3}$$

- **RMSE**: Siglas de Root Mean Square Error. Medición de la diferencia existente entre los valores predichos por un modelo y los valores reales. Matemáticamente, está dado por:

$$RMSE = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2} \tag{0.4}$$

  Donde n es el tamaño de la muestra de datos, y_i es el valor real asociado al dato i-ésimo e $\hat{y}_i$ es el valor predicho para el mismo dato.

- **RMSprop**: Método de optimización que cuenta con una tasa de aprendizaje adaptativa, la cual es dividida por el promedio de los gradientes de la función de pérdida elevados al cuadrado.

- **SGD**: Siglas de Stochastic Gradient Descent. Método de optimización caracterizado por una tasa de aprendizaje global, una función de perdida, un tamaño de mini-batch y el uso opcional de momentum y weight decay.

- **Sigmoidal**: Función de activación dada por:

$$f(x) = \frac{1}{1 + e^x} \tag{0.5}$$

- **Tanh**: Función de activación tangente hiperbólica, dada por:

$$f(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{0.6}$$

- **Tasa de aprendizaje**: Hiper parámetro que determina la medida en que debe actualizarse un determinado parámetro de una red neuronal durante la fase de entrenamiento. Posee un valor entre 0 y 1, de tal manera que mientras mayor es su valor, mayor es el cambio que sufren los parámetros.

- **Time series forecasting**: Problema en que se desea predecir el valor futuro de una variable a partir de valores pasados de la misma.

- **Weight decay**: Técnica de regularización usada para evitar que los parámetros de una red neuronal crezcan demasiado. Posee un valor entre 0 y 1, de tal forma que mientras mayor es su valor, más se limita el crecimiento de los pesos.
- **ZCA whitening**: Técnica de pre-procesamiento de un conjunto de ejemplos que lleva a que la matriz de covarianza asociada a este sea la matriz identidad.

Índice

1 Introducción **1**

1.1 Definición del Problema 2

1.2 Objetivos 3

1.2.1 Objetivo Principal 3

1.2.2 Objetivos específicos 3

1.3 Estructura del documento 4

2 Antecedentes **5**

2.1 ¿Como funciona una red neuronal? 5

2.2 Breve revisión de los tipos de redes que se estudiarán 7

2.2.1 Red feedforward 7

2.2.2 Red convolucional 10

2.2.3 Red recurrente 17

2.3 ¿Qué es el deep learning? 23

2.4 Entrenamiento de redes neuronales en GPU 25

2.5 ¿Cómo se procesan los datos en una GPU? 26

2.6 Breve revisión de frameworks a estudiar 31

2.6.1 Theano 31

2.6.2 TensorFlow 31

2.6.3 CNTK 32

2.6.4 Keras 32

3 Estado del Arte **34**

4 Diseño de Experimentos y Resultados **66**

4.1 Hardware utilizado 66

4.2 Frameworks . . . 66
4.3 Datasets . . . 67
4.4 Métricas de evaluación . . . 71
4.5 Experimentos preliminares . . . 72
4.6 Modelos implementados y resultados . . . 73
4.6.1 Redes feedforward . . . 73
4.6.1.1 CIFAR-10 . . . 73
4.6.1.1.1 Eficiencia en base a número de capas ocultas 75
4.6.1.1.2 Eficiencia en base al tamaño de mini-batch . 77
4.6.1.1.3 Consumo de memoria GPU . . . 79
4.6.1.2 SVHN . . . 80
4.6.1.2.1 Eficiencia en base a número de capas ocultas 80
4.6.1.2.2 Eficiencia en base al tamaño de mini-batch . 83
4.6.1.2.3 Consumo de memoria GPU . . . 85
4.6.1.3 MNIST . . . 86
4.6.1.3.1 Eficiencia en base a número de capas ocultas 86
4.6.1.3.2 Eficiencia en base al tamaño de mini-batch . 88
4.6.1.3.3 Consumo de memoria GPU . . . 90
4.6.2 Redes convolucionales . . . 90
4.6.2.1 MNIST . . . 92
4.6.2.1.1 Arquitectura maestra . . . 92
4.6.2.1.2 Arquitectura básica . . . 93
4.6.2.1.3 Variación de la cantidad de ejemplos de entrenamiento . . . 95
4.6.2.1.4 Variación de la cantidad de capas convolucionales . . . 99
4.6.2.1.5 Variación de la cantidad de filtros convolucionales . . . 104
4.6.2.1.6 Variación de tamaños de filtros convolucionales . . . 109
4.6.2.2 CIFAR-10 . . . 114
4.6.2.2.1 Arquitectura maestra . . . 114
4.6.2.2.2 Arquitectura básica . . . 116

4.6.2.2.3 Variación de cantidad de ejemplos de entrenamiento . . . 118
4.6.2.2.4 Variación de la cantidad de capas convolucionales . . . 122
4.6.2.2.5 Variación en la cantidad de filtros convolucionales . . . 127
4.6.2.2.6 Variación de tamaños de filtros convolucionales . . . 132
4.6.2.3 SVHN . . . 137
4.6.2.3.1 Arquitectura maestra . . . 137
4.6.2.3.2 Arquitectura básica . . . 140
4.6.2.3.3 Variación de cantidad de ejemplos de entrenamiento . . . 141
4.6.2.3.4 Variación de la cantidad de capas convolucionales . . . 145
4.6.2.3.5 Variación de la cantidad de filtros convolucionales . . . 150
4.6.2.3.6 Variación de tamaños de filtros convolucionales . . . 155
4.6.2.4 NORB . . . 160
4.6.2.4.1 Arquitectura maestra . . . 160
4.6.2.4.2 Arquitectura básica . . . 162
4.6.2.4.3 Variación de cantidad de ejemplos de entrenamiento . . . 163
4.6.2.4.4 Variación de la cantidad de capas convolucionales . . . 166
4.6.2.4.5 Variación de la cantidad de filtros convolucionales . . . 171
4.6.2.4.6 Variación de tamaños de filtros convolucionales . . . 176
4.6.3 Redes recurrentes . . . 181
4.6.3.1 IMDB Movie reviews . . . 181
4.6.3.1.1 Variación de la cantidad de capas LSTM . . . 183
4.6.3.1.2 Variación de la cantidad de celdas LSTM . . . 187

4.6.3.2 How Much Did It Rain? II 192
4.6.3.2.1 Variación de la cantidad de capas LSTM . . 193
4.6.3.2.2 Variación de la cantidad de celdas LSTM . . 197

5 Conclusiones **203**

Referencias Bibliográficas **208**

Anexos **214**
A Módulo inception 214

Índice de Figuras

2.1 Estructura básica de un perceptrón. 6
2.2 Estructura básica de una red feedforward. 8
2.3 Aplicación de un filtro de tamaño 3 x 3 x 1 sobre un input de dimensiones 7 x 7 x 1, dando como resultado un feature map de tamaño 5 x 5 x 1. Por simplicidad, se utiliza la función identidad como función de activación. Fuente: http://www.diegocalvo.es/red-neuronal-convolucional-cnn/ 11
2.4 Aplicación de una capa de max pooling con filtro de tamaño 2 x 2 y stride 2 sobre un input de dimensiones 4 x 4 x 1 que produce un output de tamaño 2 x 2 x 1. Fuente: http://www.diegocalvo.es/red-neuronal-convolucional-cnn/ 15
2.5 Red convolucional compuesta por una capa convolucional de cuatro filtros, una capa de pooling, una capa convolucional de ocho filtros, otra capa de pooling, cuatro capas totalmente conectadas de ocho, nueve, nueve y nueve neuronas, respectivamente, y finalmente, la capa de salida, integrada por cuatro neuronas (clases). Fuente: http://www.diegocalvo.es/red-neuronal-convolucional-cnn/ 16
2.6 Diagrama que representa la estructura básica de una red neuronal recurrente. 17
2.7 Diagrama que muestra la estructura ”desenrollada”de la red recurrente de la figura 2.6 para la secuencia (x_1, x_2, x_3, x_4). 19
2.8 Estructura de una red LSTM estándar. La celda de la red es denotada por A. x_t es el input recibido en el tiempo t y h_t es la salida generada en el tiempo t. C_t representa el estado de la celda en el tiempo t. Fuente: [8]. . 21
2.9 Simbología utilizada en la figura 2.8. Fuente: [8] 21

2.10 Arquitectura ejemplo de una GPU compatible con el modelo CUDA. Cada cuadrado celeste corresponde a un SP. Cada conjunto de 8 SP's da origen a un SM. 2 SM's forman un bloque. Fuente: [13]. 28
2.11 Ejecución de un programa en una GPU que opera bajo el modelo CUDA. Fuente: [13]. 29
2.12 Estructura interna de un grid. Fuente: [13]. 30

3.1 Estructura de ResNet-50. Imagen adaptada de: https://medium.com/@siddharthdas_32104/cnns-architectures-lenet-alexnet-vgg-googlenet-resnet-and-more-666091488df5 62

4.1 Incremento de la profundidad de una red feedforward y su impacto en el tiempo total de entrenamiento. En cada caso, la red resultante ha sido entrenada sobre los descriptores GIST del dataset CIFAR-10. 76
4.2 Speed-up relativo entre los tiempos de entrenamiento obtenidos por cada par de frameworks al incrementar la profundidad de una red feedforward entrenada sobre los descriptores GIST del dataset CIFAR-10. 77
4.3 Variación del tamaño de mini-batch utilizado en el entrenamiento de FNN-6 y su impacto en el tiempo total de entrenamiento. En cada caso, la red es entrenada sobre los descriptores GIST del dataset CIFAR-10. 78
4.4 Speed-up relativo entre los tiempos de entrenamiento de FNN-6 obtenidos por cada par de frameworks al variar el tamaño de mini-batch. En cada caso, FNN-6 es entrenada sobre los descriptores GIST del dataset CIFAR-10. 79
4.5 Incremento de la profundidad de una red feedforward y su impacto en el tiempo total de entrenamiento. En cada caso, la red resultante ha sido entrenada sobre la representación combinada de los descriptores HOG y los histogramas de tonos del dataset SVHN. 82
4.6 Speed-up relativo entre los tiempos de entrenamiento obtenidos por cada par de frameworks al incrementar la profundidad de una red feedforward entrenada sobre la representación combinada de los descriptores HOG y los histogramas de tonos del dataset SVHN. 83

4.7 Variación del tamaño de mini-batch utilizado en el entrenamiento de FNN-6 y su impacto en el tiempo total de entrenamiento. En cada caso, la red es entrenada sobre la representación combinada de los descriptores HOG y los histogramas de tonos del dataset SVHN. 84
4.8 Speed-up relativo entre los tiempos de entrenamiento de FNN-6 obtenidos por cada par de frameworks al variar el tamaño de mini-batch. En cada caso, FNN-6 es entrenada sobre la representación combinada de los descriptores HOG y los histogramas de tonos del dataset SVHN. 85
4.9 Incremento de la profundidad de una red feedforward y su impacto en el tiempo total de entrenamiento. En cada caso, la red resultante es entrenada sobre la concatenación de píxeles del dataset MNIST. 87
4.10 Speed-up relativo entre los tiempos de entrenamiento obtenidos por cada par de frameworks al variar la profundidad de una red feedforward. En cada caso, la red resultante es entrenada sobre la concatenación de píxeles del dataset MNIST. 88
4.11 Variación del tamaño de mini-batch utilizado en el entrenamiento de FNN-6 y su impacto en el tiempo total de entrenamiento. En cada caso, FNN-6 es entrenada sobre la concatenación de píxeles del dataset MNIST. 89
4.12 Speed-up relativo entre los tiempos de entrenamiento de FNN-6 obtenidos por cada par de frameworks al variar el tamaño de mini-batch. En cada caso, FNN-6 es entrenada sobre la concatenación de píxeles del dataset MNIST. 90
4.13 Variación de la fracción de ejemplos provistos para el entrenamiento de CNN-Softmax y su impacto en el tiempo total de entrenamiento. Dataset: MNIST. 97
4.14 Variación de la fracción de ejemplos provistos para el entrenamiento de CNN-Softmax y su impacto en la cantidad promedio de ejemplos procesados. Dataset: MNIST. 98
4.15 Speed-up relativo entre los tiempos de entrenamiento de CNN-Softmax para cada par de frameworks en función de la fracción de ejemplos provistos. Dataset: MNIST. 99
4.16 Incremento de la cantidad de capas convolucionales en CNN-Softmax y su impacto en el tiempo total de entrenamiento. Dataset: MNIST. 101

4.17 Speed-up relativo entre los tiempos de entrenamiento de CNN-Softmax para cada par de frameworks en función del factor de aumento. Dataset: MNIST. 102
4.18 Incremento de la cantidad de capas convolucionales en CNN-Softmax y su impacto en el consumo de memoria GPU. Dataset: MNIST. 103
4.19 Uso relativo de memoria GPU de cada par de frameworks en el entrenamiento de CNN-Softmax como función del factor de aumento. Dataset: MNIST. 104
4.20 Variación de la cantidad de filtros convolucionales en CNN-Softmax y su impacto en el tiempo total de entrenamiento. Dataset: MNIST. 106
4.21 Speed-up relativo entre los tiempos de entrenamiento de CNN-Softmax para cada par de frameworks en función de la variación en la cantidad de filtros. Dataset: MNIST. 107
4.22 Variación de la cantidad de filtros convolucionales de CNN-Softmax y su impacto en el consumo de memoria GPU. Dataset: MNIST. 108
4.23 Uso relativo de memoria GPU entre cada par de frameworks en el entrenamiento de CNN-Softmax como función de la variación de la cantidad de filtros. Dataset: MNIST. 109
4.24 Variación de los tamaños de los filtros convolucionales de CNN-Softmax y su impacto en el tiempo total de entrenamiento. Dataset: MNIST. . . . 111
4.25 Speed-up relativo entre los tiempos de entrenamiento de CNN-Softmax para cada par de frameworks en función de la variación en la cantidad de filtros. Dataset: MNIST. 112
4.26 Variación de los tamaños de los filtros convolucionales de CNN-Softmax y su impacto en el consumo de memoria GPU. Dataset: MNIST. 113
4.27 Razón entre los consumos de memoria GPU para cada par de frameworks en el entrenamiento de CNN-Softmax como función de la variación en el tamaño de los filtros. Dataset: MNIST. 114
4.28 Variación de la fracción de ejemplos provistos para el entrenamiento de CNN + ReLU y su impacto en el tiempo total de entrenamiento. Dataset: CIFAR-10. 120
4.29 Variación de la fracción de ejemplos provistos para el entrenamiento de CNN + ReLU y su impacto en la cantidad promedio de ejemplos de entrenamiento procesados por segundo. Dataset: CIFAR-10. 121

4.30 Speed-up relativo entre los tiempos de entrenamiento de CNN + ReLU para cada par de frameworks en función de la fracción de ejemplos provistos. Dataset: CIFAR-10. 122
4.31 Incremento de la cantidad de capas convolucionales en CNN + ReLU y su impacto en el tiempo total de entrenamiento. Dataset: CIFAR-10. . . 124
4.32 Speed-up relativo entre los tiempos de entrenamiento de CNN + ReLU para cada par de frameworks en función del factor de aumento. Dataset: CIFAR-10. 125
4.33 Incremento de la cantidad de capas convolucionales en CNN + ReLU y su impacto en el consumo de memoria GPU. Dataset: CIFAR-10. 126
4.34 Uso relativo de memoria GPU entre cada par de frameworks en el entrenamiento de CNN + ReLU como función del factor de aumento. Dataset: CIFAR-10. 127
4.35 Variación de la cantidad de filtros convolucionales en CNN + ReLU y su impacto en el tiempo total de entrenamiento. Dataset: CIFAR-10. . . 129
4.36 Speed-up relativo entre los tiempos de entrenamiento de CNN + ReLU de cada par de frameworks en función de la variación en la cantidad de filtros. Dataset: CIFAR-10. 130
4.37 Variación de la cantidad de filtros convolucionales de CNN + ReLU y su impacto en el consumo de memoria GPU. Dataset: CIFAR-10. 131
4.38 Razón entre los consumos de memoria GPU para cada par de frameworks en el entrenamiento de CNN-Softmax como función de la variación en la cantidad de filtros. Dataset: CIFAR-10. 132
4.39 Variación de los tamaños de los filtros convolucionales de CNN + ReLU y su impacto en el tiempo total de entrenamiento. Dataset: CIFAR-10. . 134
4.40 Razón entre los tiempos de entrenamiento de CNN + ReLU para cada par de frameworks en función de la variación en el tamaño de los filtros. Dataset: CIFAR-10. 135
4.41 Variación de los tamaños de los filtros convolucionales de CNN + ReLU y su impacto en el consumo de memoria GPU. Dataset: CIFAR-10. . . . 136
4.42 Uso relativo de memoria GPU entre cada par de frameworks en el entrenamiento de CNN + ReLU como función de la variación del tamaño de los filtros. Dataset: CIFAR-10. 137
4.43 Estructura de WRN-16-8. 138

4.44 Variación de la fracción de ejemplos provistos para el entrenamiento de basic-SVHN y su impacto en el tiempo total de entrenamiento. Dataset: SVHN. 143
4.45 Variación de la fracción de ejemplos provistos para el entrenamiento de basic-SVHN y su impacto en la cantidad promedio de ejemplos procesados. Dataset: SVHN. 144
4.46 Speed-up relativo de tiempos de entrenamiento de basic-SVHN entre cada par de frameworks en función de la fracción de ejemplos provistos. Dataset: SVHN. 145
4.47 Incremento de la cantidad de capas convolucionales en basic-SVHN y su impacto en el tiempo total de entrenamiento. Dataset: SVHN. 147
4.48 Razón entre los tiempos de entrenamiento de basic-SVHN para cada par de frameworks en función del factor de aumento. Dataset: SVHN. 148
4.49 Incremento de la cantidad de capas convolucionales en basic-SVHN y su impacto en el consumo de memoria GPU. Dataset: SVHN. 149
4.50 Uso relativo de memoria GPU entre cada par de frameworks en el entrenamiento de basic-SVHN como función del factor de aumento. Dataset: SVHN. 150
4.51 Variación de la cantidad de filtros convolucionales en basic-SVHN y su impacto en el tiempo total de entrenamiento. Dataset: SVHN. 152
4.52 Razón entre los tiempos de entrenamiento de basic-SVHN para cada par de frameworks en función de la variación en la cantidad de filtros. Dataset: SVHN. 153
4.53 Variación de la cantidad de filtros convolucionales de basic-SVHN y su impacto en el consumo de memoria GPU. Dataset: SVHN. 154
4.54 Razón entre los consumos de memoria GPU para cada par de frameworks en el entrenamiento de basic-SVHN como función de la variación en la cantidad de filtros. Dataset: SVHN. 155
4.55 Variación de los tamaños de los filtros convolucionales de basic-SVHN y su impacto en el tiempo total de entrenamiento. Dataset: SVHN. . . . 157
4.56 Speed-up relativo entre los tiempos de entrenamiento de basic-SVHN de cada par de frameworks en función de la variación en los tamaños de los filtros. Dataset: SVHN. 158

4.57 Variación de los tamaños de los filtros convolucionales de basic-SVHN y su impacto en el consumo de memoria GPU. Dataset: SVHN. 159
4.58 Razón entre los consumos de memoria GPU para cada par de frameworks en el entrenamiento de basic-SVHN como función de la variación en los tamaños de los filtros. Dataset: SVHN. 160
4.59 Variación de la fracción de ejemplos provistos para el entrenamiento de MP-ReLU y su impacto en el tiempo total de entrenamiento. Dataset: NORB. 164
4.60 Variación de la fracción de ejemplos provistos para el entrenamiento de MP-ReLU y su impacto en la cantidad promedio de ejemplos procesados. Dataset: NORB. 165
4.61 Speed-up relativo entre los tiempos de entrenamiento de MP-ReLU para cada par de frameworks en función de la fracción de ejemplos provistos. Dataset: NORB. 166
4.62 Incremento de la cantidad de capas convolucionales en MP-ReLU y su impacto en el tiempo total de entrenamiento. Dataset: NORB. 168
4.63 Speed-up relativo entre los tiempos de entrenamiento de MP-ReLU para cada par de frameworks en función del factor de aumento 169
4.64 Incremento de la cantidad de capas convolucionales en MP-ReLU y su impacto en el consumo de memoria GPU. Dataset: NORB. 170
4.65 Uso relativo de memoria GPU entre cada par de frameworks en el entrenamiento de MP-ReLU como función del factor de aumento. Dataset: NORB. 171
4.66 Variación de la cantidad de filtros convolucionales en MP-ReLU y su impacto en el tiempo total de entrenamiento. Dataset: NORB. 173
4.67 Speed-up relativo entre los tiempos de entrenamiento de MP-ReLU de cada par de frameworks en función de la variación en la cantidad de filtros. Dataset: NORB. 174
4.68 Variación de la cantidad de filtros convolucionales de MP-ReLU y su impacto en el consumo de memoria GPU. Dataset: NORB. 175
4.69 Uso relativo de memoria GPU entre cada par de frameworks en el entrenamiento de MP-ReLU como función de la variación en la cantidad de filtros. Dataset: NORB. 176

4.70 Variación de los tamaños de los filtros convolucionales de MP-ReLU y su impacto en el tiempo total de entrenamiento. Dataset: NORB. 178
4.71 Speed-up relativo entre los tiempos de entrenamiento de MP-ReLU de cada par de frameworks en función de la variación en el tamaño de los filtros. Dataset: NORB. 179
4.72 Variación de los tamaños de los filtros convolucionales de MP-ReLU y su impacto en el consumo de memoria GPU. Dataset: NORB. 180
4.73 Razón entre los consumos de memoria GPU para cada par de frameworks en el entrenamiento de MP-ReLU como función de la variación de la cantidad de filtros. Dataset: NORB. 181
4.74 Aumento de la cantidad de capas LSTM en IMDB-LSTM y su impacto en el tiempo total de entrenamiento. Dataset: IMDB Movie Reviews. . . 184
4.75 Speed-up relativo entre los tiempos de entrenamiento de IMDB-LSTM de cada par de frameworks en función del número de capas LSTM en la red. Dataset: IMDB Movie Reviews. 185
4.76 Aumento de la cantidad de capas LSTM en IMDB-LSTM y su impacto en el consumo de memoria GPU. Dataset: IMDB Movie Reviews. . . . 186
4.77 Uso relativo de memoria GPU entre cada par de frameworks en el entrenamiento de IMDB-LSTM como función del número de capas LSTM en la red. Dataset: IMDB Movie Reviews. 187
4.78 Aumento de la cantidad de celdas LSTM en IMDB-LSTM y su impacto en el tiempo total de entrenamiento. Dataset: IMDB Movie Reviews. . . 189
4.79 Speed-up relativo entre los tiempos de entrenamiento de IMDB-LSTM de cada par de frameworks en función del número de celdas LSTM en la red. Dataset: IMDB Movie Reviews. 190
4.80 Aumento de la cantidad de celdas LSTM en IMDB-LSTM y su impacto en el consumo de memoria GPU. Dataset: IMDB Movie Reviews. . . . 191
4.81 Razón entre los consumos de memoria GPU de cada par de frameworks en el entrenamiento de IMDB-LSTM como función del número de celdas LSTM en la red. Dataset: IMDB Movie Reviews. 192
4.82 Aumento de la cantidad de capas LSTM en HMDIR-LSTM y su impacto en el tiempo total de entrenamiento. Dataset: HMDIR-II. 194

4.83 Speed-up relativo entre los tiempos de entrenamiento de HMDIR-LSTM de cada par de frameworks en función del número de capas LSTM en la red. Dataset: HMDIR-II. 195
4.84 Aumento de la cantidad de capas LSTM en HMDIR-LSTM y su impacto en el consumo de memoria GPU. Dataset: HMDIR-II. 196
4.85 Uso relativo de memoria GPU entre cada par de frameworks en el entrenamiento de HMDIR-LSTM como función del número de capas LSTM en la red. Dataset: HMDIR-II. 197
4.86 Aumento de la cantidad de celdas LSTM en HMDIR-LSTM y su impacto en el tiempo total de entrenamiento. Dataset: HMDIR-II. 199
4.87 Speed-up relativo entre los tiempos de entrenamiento de HMDIR-LSTM obtenidos por cada par de frameworks en función del número de celdas LSTM en la red. Dataset: HMDIR-II. 200
4.88 Aumento de la cantidad de celdas LSTM en HMDIR-LSTM y su impacto en el consumo de memoria GPU. Dataset: HMDIR-II. 201
4.89 Razón entre los consumos de memoria GPU de cada par de frameworks en el entrenamiento de HMDIR-LSTM como función del número de celdas LSTM en la red. Dataset: HMDIR-II. 202

A.1 Estructura de un módulo inception. 215

Índice de Tablas

2.1 Comparación entre las características de una CPU y una GPU. 26

3.1 Frameworks que consiguen los mejores resultados en cada una de las arquitecturas estudiadas en [19]. Sólo se informaron tiempos para batches de tamaño 64, 256, 16 y 64 para LeNet, AlexNet, la red LSTM y los autoencoders, respectivamente. Para los autoencoders, se indica entre paréntesis la cantidad de capas ocultas de cada uno. En todo los casos, el mejor rendimiento fue alcanzando en conjunto con las librerías CUDA 7.5 y cuDNN v3. Notación: (g): Gradient computation time, (f): Forward time. 40

3.2 Framework más rápido para cada tamaño de batch posible usado en GPU, caso LeNet. Notación: (g): Gradient Computation time, (f): Forward time, (g, f): Tanto gradient computation time como forward time. . 42

3.3 Framework más rápido para cada tamaño de batch posible usado en GPU, caso AlexNet. Notación: (g): Gradient Computation time, (f): Forward time, (g, f): Tanto gradient computation time como forward time. . 43

3.4 Frameworks que consiguen los mejores resultados en el entrenamiento de cada una de las arquitecturas convolucionales estudiadas en [7] en base al tiempo promedio de procesamiento por mini-batch. No se informaron tiempos específicos. Dataset utilizado: ImageNet. Notación: (b): Backward time, (f): Forward computation time, (g, f): Tanto gradient computation time como forward time. 52

3.5 Frameworks que consiguen los mejores resultados en cada una de las redes LSTM estudiadas en [7] en base a la cantidad de palabras procesadas por segundo (en miles de palabras). Se utilizaron mini-batches de tamaño 20. No se informó la cantidad especifica de palabras procesadas por cada framework. Dataset utilizado: Penn TreeBank. 53
3.6 Frameworks que consiguen los mejores resultados para cada tamaño posible de mini-batch utilizado en los experimentos realizados con la red recurrente para mapeo sequence-to-sequence en [7], de acuerdo al tiempo promedio de procesamiento por mini-batch. No se informaron tiempos específicos. Dataset utilizado: Frames de vídeos. Notación: (b): Backward time, (f): Forward computation time, (g, f): Tanto gradient computation time como forward time. 54
3.7 Frameworks que consiguen los mejores resultados en el entrenamiento de cada uno de los modelos convolucionales estudiados en [28], en base a la mediana de los tiempos empleados en las iteraciones del proceso de entrenamiento. Dataset utilizado: ImageNet. 59
3.8 Frameworks que consiguen los mejores resultados en cada una de las arquitecturas estudiadas en [31]. Sólo se informaron los tiempos específicos asociados a batches de tamaño 64, 16 y 128 para las redes feedforward, convolucionales y recurrentes, respectivamente. 65

4.1 Características de cada framework utilizado en la implementación de los experimentos. 67
4.2 Características principales de los datasets considerados para problemas de clasificación de imágenes. 69
4.3 Tiempo de entrenamiento promedio de las redes FNN-6 y DNN-W20. Datasets: CIFAR-10 y MNIST, respectivamente. En cada caso, el experimento fue repetido cinco veces. 72
4.4 Características de cada una de las redes feedforward entrenadas sobre los descriptores GIST de CIFAR-10. Una arquitectura denotada por FNN-i representa a una red feedforward integrada por *i* capas ocultas. 75
4.5 Características de cada una de las redes feedforward entrenadas sobre la representación combinada de los descriptores HOG y los histogramas de tonos del dataset SVHN. Una arquitectura denotada por FNN-i representa a una red feedforward integrada por *i* capas ocultas. 81

4.6 Características de cada una de las redes feedforward entrenadas sobre los píxeles en bruto de MNIST. Una arquitectura denotada por FNN-i representa a una red feedforward integrada por *i* capas ocultas. 86
4.7 Rendimiento sobre arquitectura DNN-W20. Dataset: MNIST. 93
4.8 Rendimiento sobre arquitectura CNN-Softmax. Dataset: MNIST. 95
4.9 Fracciones de ejemplos usadas para el entrenamiento de CNN-Softmax. Dataset: MNIST. 96
4.10 Aumento en la cantidad de capas convolucionales de CNN-Softmax, junto con el número total de parámetros del modelo resultante. Dataset: MNIST. 100
4.11 Cantidad de filtros y parámetros para cada variación estudiada de CNN-Softmax, donde C_i corresponde a la capa convolucional *i* de la red. Dataset: MNIST. 105
4.12 Disminución / incremento de los tamaños de los filtros de las capas convolucionales de CNN-Softmax, junto con la cantidad de parámetros resultantes. C_i representa la capa convolucional *i*. Dataset: MNIST. 110
4.13 Rendimiento sobre arquitectura All-CNN-C. Dataset: CIFAR-10. 116
4.14 Rendimiento sobre arquitectura CNN + ReLU. Dataset: CIFAR-10. . . . 118
4.15 Fracciones de ejemplos usadas para el entrenamiento de CNN + ReLU. Dataset: CIFAR-10. 119
4.16 Aumento de la cantidad de capas convolucionales en CNN + ReLU, junto con el número total de parámetros del modelo resultante. Dataset: CIFAR-10. 123
4.17 Cantidad de filtros y parámetros para cada variación estudiada de CNN + ReLU, donde C_i corresponde a la capa convolucional *i* del modelo. Dataset: CIFAR-10. 128
4.18 Disminución / incremento en el tamaño de los filtros de las capas convolucionales de CNN + ReLU, junto con la cantidad de parámetros resultantes. C_i representa la capa convolucional *i*. Dataset: CIFAR-10. . . . 133
4.19 Rendimiento sobre arquitectura WRN-16-8. Dataset: SVHN. 139
4.20 Rendimiento sobre arquitectura basic-SVHN. Dataset: SVHN. 141
4.21 fracciones de ejemplos usadas para el entrenamiento de basic-SVHN. Dataset: SVHN. 142

4.22 Aumento en la cantidad de capas convolucionales de basic-SVHN, junto con el número total de parámetros del modelo resultante. Dataset: SVHN. 146
4.23 Cantidad de filtros y parámetros para cada variación de basic-SVHN, donde C_i corresponde a la capa convolucional i del modelo. Dataset: SVHN. 151
4.24 Disminución / incremento en el tamaño de los filtros de las capas convolucionales de basic-SVHN, junto con la cantidad de parámetros resultantes. C_i representa la capa convolucional i. Dataset: SVHN. 156
4.25 Rendimiento sobre arquitectura AP-ReLU. Dataset: NORB. 162
4.26 Rendimiento sobre arquitectura MP-ReLU. Dataset: NORB. 163
4.27 Fracciones de ejemplos usadas para el entrenamiento de MP-ReLU. Dataset: NORB. 163
4.28 Aumento en la cantidad de capas convolucionales en MP-ReLU, junto con el número total de parámetros de los modelos resultantes. Dataset: NORB. 167
4.29 Cantidad de filtros y parámetros para cada variación estudiada de MP-ReLU, donde C_i corresponde a la capa convolucional i de la red. Dataset: NORB. 172
4.30 Disminución / incremento en el tamaño de los filtros de las capas convolucionales de MP-ReLU, junto con la cantidad de parámetros resultantes. C_i representa la capa convolucional i. Dataset: NORB. 177
4.31 Variación del número de capas LSTM en IMDB-LSTM, junto a la cantidad total de parámetros de la arquitectura resultante. Dataset: IMDB Movie Reviews. 183
4.32 Variación del número de celdas LSTM en IMDB-LSTM, junto a la cantidad total de parámetros de la arquitectura resultante. Dataset: IMDB Movie Reviews. 188
4.33 Variación del número de capas LSTM en HMDIR-LSTM, junto a la cantidad total de parámetros de la arquitectura resultante. Dataset: HMDIR-II. 193
4.34 Variación del número de celdas LSTM en HMDIR-LSTM, junto a la cantidad total de parámetros de la arquitectura resultante. Dataset: HMDIR-II. 198

Capítulo 1

Introducción

Una red neuronal artificial [1] corresponde a un esquema de cómputo constituido por una gran cantidad de nodos interconectados que son capaces de procesar datos. Utilizado fundamentalmente para resolver problemas de aprendizaje automático, este esquema corresponde a una abstracción del funcionamiento del cerebro humano, si bien recoge aspectos muy básicos acerca del mismo. En este contexto, los nodos son denominados neuronas.

Las redes neuronales son útiles para:

- Descubrir reglas o patrones en base a un conjunto de datos de entrada.
- Trabajar con grandes volúmenes de datos y/o cantidad de atributos asociados a éstos.
- Trabajar con variables cuyas relaciones entre sí se desconocen o se conocen poco.
- Describir relaciones difíciles de aproximar con métodos convencionales que se ven limitados por supuestos fuertes como normalidad, linealidad, independencia entre variables, etc.

Existen diferentes tipos de redes: Redes feedforward, redes convolucionales, redes recurrentes, autoencoders, máquinas de Boltzmann, entre otras. Las redes convolucionales han permitido avances más que significativos en problemas de visión computacional, como clasificación de imágenes o reconocimiento de objetos. Las redes recurrentes han permitido importantes avances en problemas de procesamiento de lenguaje natural,

como speech recognition, y en problemas de procesamiento de texto, como machine translation y análisis de sentimiento. Los autoencoders han sido capaces de ofrecer mejoras en problemas de reconstrucción y compresión de datos.

Por las razones antes expuestas, el uso de redes neuronales artificiales constituye un tópico de alto interés tanto en el ámbito científico como industrial, lo que ha llevado al desarrollo de una gran variedad de frameworks de software para la implementación de estas estructuras.

Sin embargo, para que una red neuronal artificial sea capaz, ya sea de predecir un valor, o de clasificar un dato, debe ser entrenada a partir de un conjunto de datos. El proceso de entrenar una red neuronal tiene asociado un alto costo computacional, principalmente en términos temporales, lo que lleva a que sea inevitable contar con los recursos más apropiados para minimizar el costo.

Actualmente, la GPU es entendida como un recurso fundamental en la implementación eficiente de una red neuronal, al ser un dispositivo especializado en el manejo de aplicaciones que implican el uso intensivo de operaciones matriciales y de computación paralela. Esto ha llevado a que los frameworks antes mencionados deban enfocarse en hacer un uso óptimo de dicho recurso. No obstante, los frameworks pueden llegar a presentar rendimientos muy disímiles entre sí, por lo que resulta necesario emplear estudios comparativos que permitan evaluar el rendimiento de cada uno y así poder determinar bajo que circunstancias es o no conveniente trabajar con un framework determinado.

1.1. Definición del Problema

Theano, TensorFlow y CNTK son tres de los frameworks más populares usados en la actualidad para implementar redes neuronales artificiales, debido a que entregan la posibilidad de utilizar computación basada en GPU's para acelerar el entrenamiento de modo transparente para el usuario/desarrollador. Actualmente, es posible acceder a las funcionalidades que estas plataformas ofrecen por medio de una interfaz única, llamada Keras. Ante este escenario, surge en el usuario la necesidad de determinar cual es la librería que se adapta mejor a determinadas necesidades, particularmente a diferentes clases de redes neuronales artificiales.

1.2. Objetivos

1.2.1. Objetivo Principal

- Evaluar el framework Keras en términos de flexibilidad (capacidad para adoptar diferentes arquitecturas y algoritmos de aprendizaje) y eficiencia computacional (tanto temporal como espacial) en la implementación de redes neuronales profundas usando Theano, TensorFlow y CNTK como backends.

1.2.2. Objetivos específicos

- Seleccionar modelos de red neuronal que se encuentren entre los mejores resultados obtenidos actualmente en tareas de visión computacional (clasificación de imágenes) y predicción de secuencias y que reflejen las elecciones arquitecturales más comunes en la literatura: redes feedforward, redes convolucionales y redes recurrentes.
- Medir los tiempos y los consumos de memoria GPU empleados por cada framework en el entrenamiento de cada uno de los modelos seleccionados.
- Caracterizar la dependencia del tiempo de entrenamiento y del consumo de memoria GPU empleado por cada framework con respecto a los parámetros arquitecturales más importantes:
    - Redes feedforward: Número de capas ocultas.
    - Redes convolucionales: Número de capas ocultas, número de filtros y tamaño de filtros.
    - Redes recurrentes: Número de capas ocultas y número de celdas.
- Caracterizar la dependencia del tiempo de entrenamiento de redes convolucionales empleado por cada framework con respecto al número de datos utilizados.

1.3. Estructura del documento

Esta memoria está organizada como se detalla a continuación.

En el capítulo 2 se establece el marco teórico de la investigación. Se detalla el proceso de aprendizaje de una red neuronal y se revisan brevemente las características de los tipos de redes que se estudiarán. Además, se define el concepto de deep learning, el rol de la GPU en el proceso de entrenamiento de una red e información acerca de los frameworks a estudiar.

En el capítulo 3 se revisan algunas de las publicaciones científicas recientes que guardan relación con el tema tratado en este documento.

En el capítulo 4 se detallan las características de los experimentos realizados junto mostrar y analizar los resultados que de ellos se derivan. Para ello, se precisa el hardware utilizado, se mencionan algunas características técnicas de los frameworks, se describen los datasets con que se trabajará, se definen las métricas consideradas para evaluar el rendimiento de los frameworks y las características de las arquitecturas implementadas.

Capítulo 2

Antecedentes

2.1. ¿Como funciona una red neuronal?

Las redes neuronales derivan originalmente del modelo de neurona artificial introducido en [2], en donde un nodo individual está conectado a cada uno de los atributos de un ejemplo determinado, los que pasan a ser denominados neuronas de entrada, denotadas por x_1, x_2, x_3, ..., x_I. Los inputs x_1, x_2, x_3, ..., x_I y pesos w_1, w_2, w_3, ..., w_I corresponden a números reales y pueden ser positivos o negativos. Así, la neurona artificial está conformada por pesos, un proceso de sumado, una función de activación y un umbral. Todos los inputs son individualmente ”pesados”, sumados y evaluados en la función de activación. Existen muchas funciones de activación, pero la que se acostumbra a utilizar para este modelo es la función escalón. Dicha función típicamente entregará como salida 1 si el input que recibe es mayor que cierto umbral, mientras que en caso contrario su salida será 0. En el primer caso, se dice que la neurona se activa, mientras que en el segundo caso la neurona se mantiene apagada.

Para el aprendizaje de la neurona artificial, datos pertenecientes a un conjunto de entrenamiento son entregados a este uno tras otro y los pesos son modificados de acuerdo a la siguiente ecuación, que se cumple para todo ejemplo:

$$w_i = w_i + \eta * (y_i - \hat{y}_i) * x_i \,, \tag{2.1}$$

donde η, denominada tasa de aprendizaje, es un parámetro que permite controlar

que tanto influye el error en la actualización de los pesos y puede interpretarse como una medida de que tan rápido aprende el modelo, pues mientras mayor es esta tasa, mayor es el cambio que experimentan los pesos. w_i es el peso i-ésimo de la neurona artificial, x_i es el atributo i-ésimo de un determinado ejemplo, y_i es la salida correcta que el perceptrón debiese conocer para dicho ejemplo e $\hat{y}_i$ es la salida generada por la neurona artificial. Cada coeficiente de peso refleja el grado de importancia que tiene el atributo respectivo en la determinación de la respuesta. En la figura 2.1 se muestra la estructura recién descrita para un perceptrón que recibe ejemplos definidos por tres atributos.

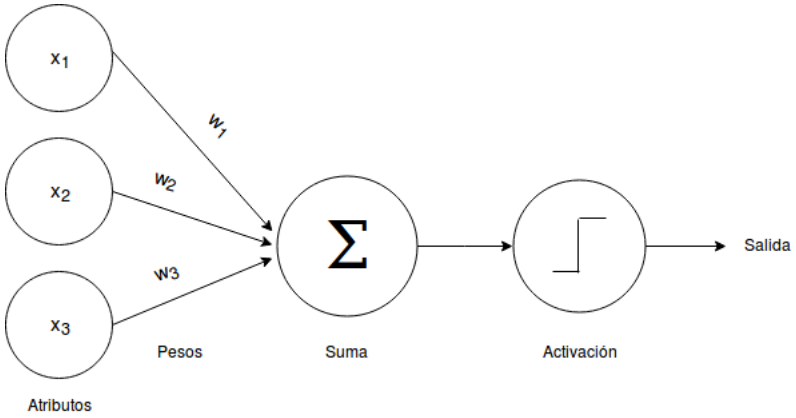


Figura 2.1: Estructura básica de un perceptrón.

Cuando un paso entero a través de todos los vectores de entrenamiento de entrada se completa sin error, la neurona artificial ha aprendido. En este punto, si se entrega un vector de entrada x que pertenece al conjunto de entrenamiento al perceptrón, este emitirá el valor correcto. Si por el contrario, x no está en el conjunto de entrenamiento, se espera que la neurona entregue una respuesta similar a la generada para otros vectores de entrenamiento parecidos a x.

¿Qué está haciendo matemáticamente la neurona? Está sumando todos los atributos de un ejemplo en forma ponderada y luego le asocia una de dos categorías posibles: En una, se agrupan los que hacen que la neurona se active y en la otra, los que no lo logran. Si se considera un problema en que los ejemplos están definidos por dos atributos. Gráficamente, esto es equivalente a dibujar la curva dada por:

$$w_1 x_1 + w_2 x_2 = t\,, \tag{2.2}$$

donde t es el umbral y x_j es el atributo j-ésimo de un ejemplo cualquiera. Los puntos que se sitúan en un lado de la curva caen en una categoría, mientras que los puntos al otro lado de la línea caen en la otra categoría.

Sin embargo, existen ciertas limitaciones asociadas al modelo de neurona. No todo conjunto de ejemplos puede ser separado por una recta como la descrita previamente. Aquellos que sí lo son, se denominan linealmente separables. En caso de no serlo, el aprendizaje nunca alcanzará un punto en el que todos los ejemplos son clasificados de forma correcta. El caso más famoso de la incapacidad de los perceptrones con conjuntos de vectores linealmente no separables es el problema booleano conocido como XOR. Es en este punto donde aparecen los perceptrones multicapa, también conocidos como redes neuronales multicapa o redes neuronales feedforward.

2.2. Breve revisión de los tipos de redes que se estudiarán

En esta memoria, se estudiarán tres tipos de arquitecturas: Redes feedforward, redes convolucionales y redes recurrentes.

2.2.1. Red feedforward

Tal como se señaló antes, una red feedforward corresponde a la expansión del concepto de neurona artificial. Este modelo puede estar integrado por múltiples neuronas organizadas en capas. Cada capa contiene un subconjunto del total de neuronas que componen a la red y asume un rol particular en el proceso de aprendizaje de la misma. Además, las conexiones existentes entre neuronas no forman ciclos ni tampoco existen loops. Básicamente, es posible identificar tres tipos de capas:

- Capa de entrada: Permite la ”presentación” de cada ejemplo a ser procesado por la red.
- Capa oculta: Lleva a cabo el procesamiento de los datos, los que son entregados a esta por medio de la capa de entrada, o bien, recibidos desde otra capa oculta.

- Capa de salida: Entrega una respuesta en base al procesamiento de los datos llevado a cabo por la(s) capa(s) oculta(s).

En esta red, la información se mueve en una sola dirección, hacia adelante, de tal manera que cada ejemplo de entrada es procesado por la(s) capa(s) oculta(s) y finalmente entrega una respuesta por medio de la capa de salida. Cada neurona de una capa oculta está conectada con todas las neuronas de la capa siguiente.

En la figura 2.2 es posible apreciar la estructura general de una red feedforward que recibe ejemplos definidos por tres atributos como entrada y tiene una capa oculta con dos neuronas y una capa de salida con dos neuronas.

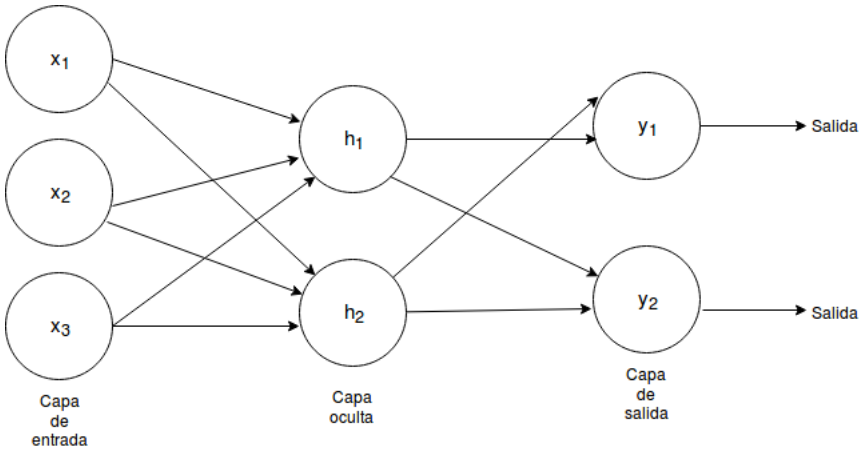


Figura 2.2: Estructura básica de una red feedforward.

Notar que pueden existir tantas neuronas como se deseen en cada una de estas capas

Ante esta arquitectura, surge la necesidad de disponer de un algoritmo de aprendizaje capaz de sintonizar no sólo los pesos existentes entre la capa oculta y la capa de salida, si no que también los pesos existentes entre la última capa oculta y la capa de entrada. Aquí es donde entra lo que se denomina Back Propagation, algoritmo encargado de ajustar los pesos de la red. En términos generales, Back Propagation:

- Propaga un ejemplo de entrenamiento a través de la red.
- Obtiene una respuesta para dicho ejemplo.
- Calcula el error en base a la respuesta correcta.
- Actualiza los pesos de la red.

Para sintonizar los pesos de las conexiones entre la capa oculta y la capa de entrada, es necesario conocer el error asociado a la capa oculta, pero sólo se conoce el error en la capa de salida, pues se conoce la salida correcta a partir del ejemplo de entrenamiento correspondiente y también se conoce el output producido por la red. De esta manera, lo que hace back propagation es tomar los errores asociados a la capa de salida y propagarlos proporcionalmente hacia la capa oculta. El error E_k de una determinada neurona k de la capa de salida está dado por:

$$E_k = (y_k - \hat{y_k}) * f'(\hat{y_k}) \,, \tag{2.3}$$

donde f es la función de activación escogida. Notar que la forma de determinar el valor de E_k corresponde sólo a una de las tantas elecciones posibles. Dicha expresión es denominada función de perdida.

De esta manera, el error asociado a una determinada neurona j de la capa oculta está dado por:

$$E_j = \sum_{k=1}^{O} W_{j,k} * E_k * f'(h_j) \,, \tag{2.4}$$

donde $W_{j,k}$ es el peso asociado a la conexión entre la j-ésima neurona de la capa oculta y la neurona k-ésima de la capa de salida, O es la cantidad de neuronas existentes en la capa de salida y h_j es la salida de la neurona j de la capa oculta.

Luego, se tiene que la actualización de cada peso asociado a una conexión entre la capa oculta y la capa de salida está dada por:

$$W_{j,k} = W_{j,k} + \eta * E_k * h_j \,. \tag{2.5}$$

Por otra parte, la actualización de cada conexión existente entre la capa de entrada y la capa oculta está dada por:

$$W_{i,j} = W_{i,j} + \eta * E_j * x_i \,, \tag{2.6}$$

donde $W_{i,j}$ es el peso asociado a la conexión entre la i-ésima neurona de la capa de

entrada y la neurona j-ésima de la capa oculta y x_i es el i-ésimo atributo.

En resumen, la secuencia de pasos a seguir para el entrenamiento de una red neuronal feedforward por medio de back propagation es:

- Inicializar los pesos de la red de acuerdo a algún esquema.
- Entregar un ejemplo perteneciente al conjunto de entrenamiento a la red neuronal y computar la salida correspondiente.
- A partir de la salida previamente generada, calcular el error asociado a cada neurona de cada una de las capas que componen la red.
- Calculados los errores, ajustar los pesos en base a las ecuaciones antes descritas, de tal manera que el error debiese decrecer.
- Los tres pasos anteriores se repiten para cada ejemplo del conjunto de entrenamiento y tantas veces como sea necesario hasta que los pesos presentan variaciones muy pequeñas entre una iteración y otra.
- De considerarse necesario, es posible agregar más capas ocultas a la red y repetir los cinco pasos anteriores.

2.2.2. Red convolucional

Las redes neuronales convolucionales [3] son utilizadas principalmente para resolver problemas de reconocimiento de imágenes y vídeos (visión computacional), implementación de sistemas recomendadores y procesamiento de lenguaje natural. Las redes neuronales convolucionales, así como las redes feedforward, están hechas de neuronas con pesos que deben ser aprendidos. Cada neurona recibe múltiples inputs, realiza una suma ponderada sobre éstos, aplica una función de activación sobre dicha suma y finalmente genera un output como respuesta. Muchas de las cosas que se explicaron previamente respecto a las redes feedforward también son válidas en este tipo de red.

Sin embargo, existen diferencias entre las redes convolucionales y las redes neuronales tradicionales. A diferencia de las redes feedforward, donde el input es un vector, las redes convolucionales reciben inputs multidimensionales, esto es, que poseen dos o más dimensiones. En el caso de problemas de visión computacional, los inputs pueden corresponder a imágenes de múltiples canales. Por lo general, una red convolucional

puede ser descrita como una combinación de capas convolucionales, capas de pooling, capas totalmente conectadas y una capa de salida.

Tal como se describió antes, cada capa de una red feedforward está compuesta por un conjunto de neuronas. Por otro lado, considerando una red convolucional que recibe imágenes tridimensionales como inputs, se tiene que cada capa convolucional está compuesta por un conjunto de filtros. Cada filtro puede ser entendido como un arreglo tridimensional de neuronas, cuya profundidad debe ser igual a la del input recibido. Luego, cada uno de estos filtros recorre el input completo aplicando productos punto elemento a elemento sobre un ”sub-arreglo” del input de la imagen que posee las mismas dimensiones del filtro, denominado campo receptivo, para luego sumar todos estas multiplicaciones. Esta operación es conocida como convolución. Posterior a la convolución, se aplica la función de activación correspondiente. El proceso descrito es repetido para cada uno de los filtros de la capa convolucional, de tal manera que el output generado por ésta capa corresponde a una salida tridimensional cuya profundidad coincide con la cantidad de filtros de la capa. Cada uno de los sub-arreglos bidimensionales que componen el output es denominado *feature map* (o *activation map)*. En la figura 2.3 se muestra un ejemplo de convolución.

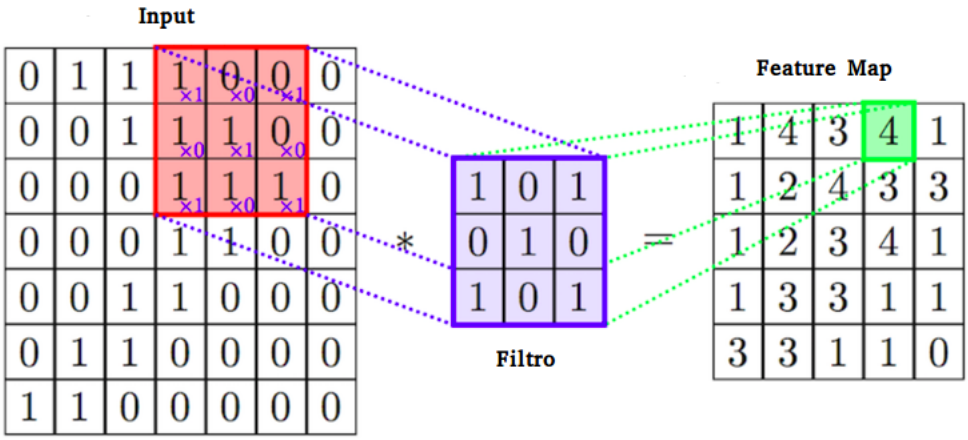


Figura 2.3: Aplicación de un filtro de tamaño 3 x 3 x 1 sobre un input de dimensiones 7 x 7 x 1, dando como resultado un feature map de tamaño 5 x 5 x 1. Por simplicidad, se utiliza la función identidad como función de activación. Fuente: http://www.diegocalvo.es/red-neuronal-convolucional-cnn/

Todos los filtros son inicializados aleatoriamente y se convierten en los parámetros

que deben ser aprendidos por la red en la etapa de entrenamiento de la misma. Conceptualmente, cada filtro tiene la función de aprender ciertas características a partir del input sobre el que es aplicado, de tal manera que a medida que son propagados por la red, cada capa convolucional aprende características más sofisticadas de los inputs recibidos en la primera capa, pasando de, por ejemplo, aprender curvas, vértices y contornos en las primeras capas a reconocer formas definidas y objetos específicos en las últimas. En el contexto de problemas de clasificación de imágenes, esto implica que cada capa está aprendiendo cierto subconjunto de características de cada una de las clases de imágenes del problema. Actualmente, las capas convolucionales suelen utilizar ReLU como función de activación.

En el pasado, funciones no lineales como tanh o sigmoide eran usadas, pero investigadores descubrieron que la función ReLU era computacionalmente más eficiente, al permitir que una red pudiera entrenarse con mayor rapidez sin afectar mayormente la precisión del modelo. La razón principal es que esta función es de gran utilidad para solventar el problema del desvanecimiento del gradiente [4], escenario en el cual resulta muy complicado aprender los parámetros de la red en las primeras capas (esto es, en aquellas más cercanas a la entrada). Este problema empeora al aumentar la cantidad de capas y tiene como causa principal precisamente la elección de una función de activación inapropiada. Específicamente, el problema que se presenta es que el gradiente con respecto a ciertos parámetros resulta ser extremadamente pequeño en las primeras capas, lo que en otras palabras quiere decir que aún cuando el valor de un parámetro presente una gran variación, esto no se verá reflejado en el output obtenido. Funciones de activación como tanh o sigmoide mapean el input recibido por la capa de tal forma que el output generado puede tomar valores dentro de un rango muy pequeño, como ocurre con la función sigmoide, que mapea inputs hacia el intervalo [0, 1]. De esta manera, aunque un input dado presente grandes cambios, el output correspondiente no presentará cambios mayores y en consecuencia el gradiente resultante será pequeño. La situación empeora en las capas posteriores, ya que la primera capa generará un output pequeño, la segunda capa generará un output aún más pequeño que el anterior y así sucesivamente. Luego, aunque los parámetros de las primeras capas presenten grandes variaciones, no se observarán grandes cambios en el output. Así, el fenómeno de desvanecimiento puede ser evitado si se escoge una función de activación que mapee los inputs recibidos hacia espacios de salida más grande, como ocurre con la función ReLU. En general, esta función ayuda a aumentar las propiedades no lineales de la redes.

Junto con determinar la cantidad y las dimensiones de los filtros de cada capa convolucional, también es necesario escoger el stride y el padding de los mismos [4].

El stride determina la cantidad de unidades que debe avanzar un determinado filtro en cada convolución realizada. Por ejemplo, si se cuenta con una imagen de dimensiones 7 x 7, un filtro de 3 x 3 y un stride de 1 x 1, implica que el filtro avanzará en una unidad en cada dirección por cada convolución que lleve a cabo. Luego, el stride de un filtro puede entenderse como la cantidad por la cual el filtro se desplaza. De esta forma, mientras mayor sea el stride utilizado, menores serán las dimensiones del output generado, lo que resulta útil cuando se requiere tratar con datos en un espacio de menor dimensionalidad. El stride es determinado de tal manera que las dimensiones del output resultante sean números enteros.

Sin embargo, la reducción de las dimensiones del input que está siendo procesado puede ser indeseable, más aún si se considera que una red convolucional se compone de varias capas convolucionales, llevando a que el efecto de reducción se vuelva permanente y se lleve a cabo más rápido de lo deseado. Con esto, una drástica reducción de las dimensiones del input en las primeras capas de la red lleva a que se pierda información fundamental acerca de este e impide que se puedan extraer características de bajo nivel. En ciertos casos, se requiere conservar las dimensiones del input luego de ser procesado por una capa convolucional. Es aquí donde entra en juego el concepto de padding, en este caso específico, zero padding. El zero padding consiste en que, para evitar que las dimensiones del input se reduzcan, éste es rodeado por bloques de ceros. Por ejemplo, si el input de la red es una imagen de dimensiones 32 x 32 x 3, y se aplica zero padding de 2 x 2, el input es rodeado por bloques de ceros de dimensiones 2 x 2. Así, se tendrá que el input, previo a la aplicación de los filtros, tendrá dimensiones 36 x 36 x 3, por lo que luego de que las convoluciones sean ejecutadas, se producirá un output de idénticas dimensiones a las del input original. En general, si se tiene un stride de 1 y se requiere que las dimensiones del input se conserven, entonces las dimensiones del zero padding deben estar dadas por:

$$Z = \frac{K - 1}{2}, \tag{2.7}$$

donde K es la dimensión de los filtros usados en la capa.

Tomando en cuenta todas las variables que entran en juego en una capa convolucio-

nal, se tiene que las dimensiones de salida del output de una capa convolucional están dados por:

$$O = \frac{I - K + 2P}{S} + 1\,, \tag{2.8}$$

donde O es la altura/largo del output, I es la altura/largo del input, K es el tamaño del filtro, P es el padding y S es el stride.

Pueden existir múltiples capas convolucionales consecutivas en una red, o bien, pueden existir capas convolucionales seguida de capas de pooling.

La función de una capa de pooling es reducir progresivamente el tamaño espacial (en alto y largo, pero no profundidad) de la representación generada por una determinada capa convolucional para disminuir la cantidad de parámetros y de cómputo en la red. Además, permite reducir la posibilidad de que el modelo se vea afectado por overfitting. Una capa de pooling está compuesta por un filtro. Típicamente, la capa de pooling opera sobre cada feature map de forma independiente. Cada capa de pooling está definida por el tamaño de la reducción a aplicar, es decir, el tamaño de su filtro, lo que se denomina pool size. Por ejemplo, si se tiene un input de tamaño 224 x 224 x 10 y se aplica un pooling con filtro de pool size 2 x 2 y stride 2, entonces el output generado posee dimensiones 112 x 112 x 10. El método de pooling más comúnmente utilizado es max pooling, en donde al aplicar filtros de tamaños n x n, se selecciona el valor máximo por cada subconjunto de n x n del input recibido. De forma análoga, también existe el average pooling, que en vez de escoger el valor máximo, toma el promedio de estos valores. Básicamente, una capa de pooling opera bajo el razonamiento de que una vez que se ha detectado una característica determinada a partir de los inputs, su ubicación específica no es lo más importante, si no su ubicación relativa con respecto a otras características. En la figura 2.4 se muestra un ejemplo de la aplicación de una capa de max pooling.

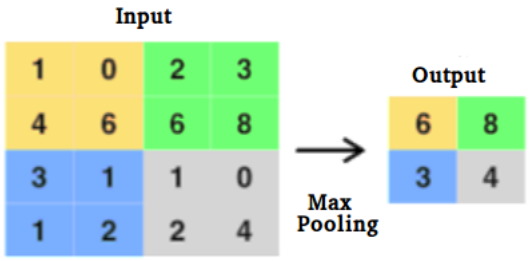


Figura 2.4: Aplicación de una capa de max pooling con filtro de tamaño 2 x 2 y stride 2 sobre un input de dimensiones 4 x 4 x 1 que produce un output de tamaño 2 x 2 x 1. Fuente: http://www.diegocalvo.es/red-neuronal-convolucional-cnn/

Luego de la sucesión de capas convolucionales y de pooling, siempre existen una o más capas totalmente conectadas, análogas a las capas ocultas dentro de una red feed-forward. En este contexto, dichas capas son denominadas como totalmente conectadas, ya que cada una de las neuronas que la integran está conectada a todos los feature maps generados por la capa anterior y a todas las neuronas de la siguiente capa totalmente conectada, si es que existe. Notar que la capa de salida debe ser de este tipo, y en el caso de problemas de clasificación, debe contener una cantidad de neuronas igual a la cantidad de clases del mismo. Al igual que en las redes feedforward destinadas a resolver este tipo de problemas, la función de activación más usada es ReLU, excepto para la capa de salida, que por lo general es softmax y entrega la probabilidad de que el input entregado a la red pertenezca a cada una de las clases del problema.

En la figura 2.5 es posible visualizar un ejemplo de red convolucional que integra todos los conceptos descritos hasta ahora.

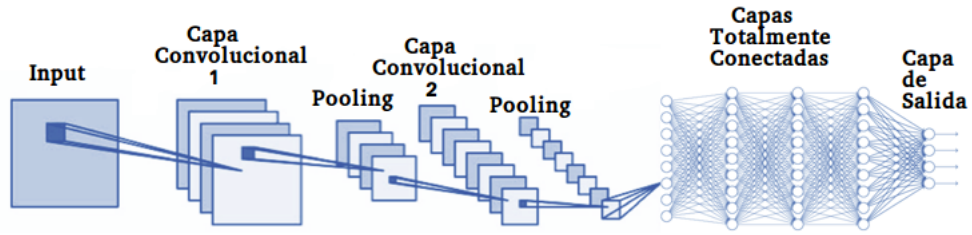


Figura 2.5: Red convolucional compuesta por una capa convolucional de cuatro filtros, una capa de pooling, una capa convolucional de ocho filtros, otra capa de pooling, cuatro capas totalmente conectadas de ocho, nueve, nueve y nueve neuronas, respectivamente, y finalmente, la capa de salida, integrada por cuatro neuronas (clases). Fuente: http://www.diegocalvo.es/red-neuronal-convolucional-cnn/

Ahora bien, no es trivial determinar de antemano la cantidad total de capas que debe tener la red, la cantidad de capas convolucionales, el tamaño de los filtros, el stride y el padding a utilizar. Más aún, no existen métodos convencionales para determinar los valores óptimos de cada uno de estos hiper parámetros, si no que son determinados en forma experimental. Esto se debe fundamentalmente a las características de los inputs a ser procesados por la red. En el caso de procesamiento de imágenes, estas pueden tener distintos tamaños, complejidades, entre otros.

Otra característica que distingue a una red convolucional es la existencia de compartición de parámetros, lo que se traduce en que todos los pesos de un filtro son idénticos. También, existe lo que se conoce como conectividad local, lo que está asociado al hecho de que cada filtro de cada capa está conectado sólo a un subconjunto del input recibido por esta última, a diferencia de las redes feedforward, en las que todas las neuronas de una capa están todas conectadas a las de la capa anterior y a las de la capa siguiente. Estos dos conceptos ayudan a reducir el número de parámetros de la red y hacen que su entrenamiento y evaluación puedan ser más eficientes.

Actualmente, las redes convolucionales son muy utilizadas en la industria [5]. Facebook utiliza este tipo de redes para sus algoritmos de etiquetado automático, Google para ofrecer la funcionalidad de búsqueda de imágenes, Amazon para su funcionalidad de recomendación de productos e Instagram para fundar los cimientos de su motor de búsqueda.

2.2.3. Red recurrente

Las redes neuronales recurrentes [6] fueron creadas en los años ’80, pero sólo recientemente han adquirido popularidad a partir de avances sobre los diseños de las redes y el incremento del poder de cómputo provisto por los computadores actuales. En las redes feedforward, se asume que todos los inputs y outputs son independientes entre sí, pero esta no es una buena idea para todos los problemas. Si se desea predecir la siguiente palabra de una oración, es fundamental conocer las palabras anteriores de la misma. Estas redes son denominadas recurrentes debido a que realizan la misma tarea para cada elemento de una secuencia, siendo el output dependiente de las computaciones previas. Se puede decir que las redes recurrentes poseen memoria que permite recordar la información que ha sido calculada hasta el momento. En teoría, las redes recurrentes pueden hacer uso de información en secuencias arbitrariamente largas, pero en la práctica se limitan a mirar hacia atrás sólo unos pocos pasos, debido al problema del gradiente desvaneciente. La figura 2.6 muestra la estructura básica de una red recurrente.

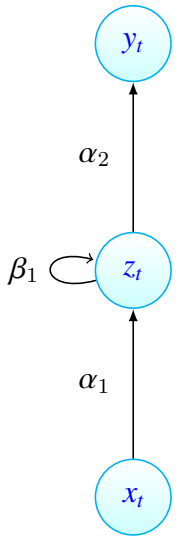


Figura 2.6: Diagrama que representa la estructura básica de una red neuronal recurrente.

Cada componente de la red mostrada en la figura es explicado a continuación:

- x_t es el input en el tiempo t. Por ejemplo, en el contexto de un problema de generación de modelos de lenguaje, donde el input de la red es una palabra determinada de un texto determinado, x_5 correspondería a la quinta palabra del texto.

- z_t es el estado oculto en el tiempo t. Es lo que corresponde a la memoria de la red, pues captura información acerca de lo que ha pasado en los pasos anteriores. z_t es calculado en base al estado oculto previo y el input recibido en el paso actual, esto es:

$$z_t = f(\alpha_1 x_t + \beta_1 z_{t-1}) \,. \tag{2.9}$$

  En general, la función f es una función no lineal como tanh o ReLU. Además, se asume que $z_{-1} = 0$.

- y_t es la salida generada en el tiempo t. Por ejemplo, si se requiere predecir la siguiente palabra de una secuencia, y_t será un vector de probabilidades que contiene la probabilidad de ocurrencia de cada palabra del vocabulario considerado. En este contexto, matemáticamente se tendría que:

$$y_t = softmax(\alpha_2 z_t) \,. \tag{2.10}$$

  Es decir, y_t es calculado solamente basado en la memoria disponible en el tiempo t.

Una red recurrente comparte los mismos parámetros (en este caso, α_1, α_2 y β_1) en cada paso. Esto es reflejo del hecho de que se están realizando las mismas tareas en cada paso, pero con inputs diferentes. Esto reduce en gran medida la cantidad de parámetros que deben ser aprendidos en la fase de entrenamiento.

En la figura [7] es posible apreciar lo que se conoce como la representación desenrollada (unfolded) de la red en los tiempos t = 1, 2, 3, 4.

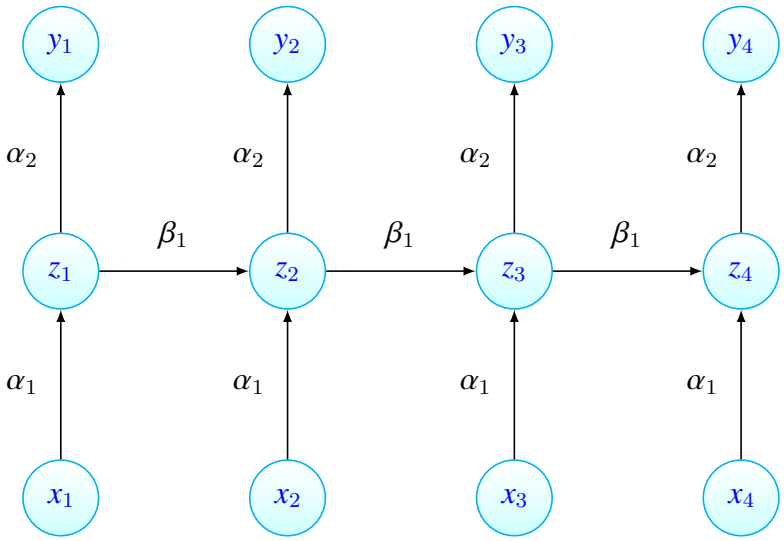


Figura 2.7: Diagrama que muestra la estructura "desenrollada"de la red recurrente de la figura 2.6 para la secuencia (x_1, x_2, x_3, x_4).

Las redes recurrentes han resultado ser exitosas en problemas de procesamiento de lenguaje natural. Dentro de esta categoría de problemas, destacan tres: language modeling, machine translation y speech recognition.

En los problemas de language modeling, se tiene que, dada una secuencia de palabras, se desea predecir la probabilidad de cada palabra dadas las anteriores. De esta manera, un modelo de lenguaje permite medir que tan probable es que una oración pueda darse. El modelo construido es de tipo generativo, pues permite generar texto nuevo en base a las probabilidades establecidas por el modelo de lenguaje. En problemas de esta clase, lo más frecuente es tratar con secuencias de palabras representadas como vectores binarios de ocurrencias de palabras en dicha secuencia. Por ejemplo, si se dispone de un vocabulario de cinco palabras, entonces para representar la sentencia en que se presentan la segunda, la cuarta y la quinta palabra del vocabulario, se utiliza el vector (0, 1, 0, 1, 1). La salida de la red representa las probabilidades de ocurrencia de cada una de ellas.

En problemas de machine translation, los inputs son secuencias de palabras en un determinado idioma. Como output, se desea producir la secuencia anterior de palabras, pero en otro idioma. Una diferencia fundamental con los problemas de language modeling es que el output puede generarse solamente luego de que se ha visto el input completo, esto es, todas las palabras de la secuencia, ya que para traducir la primera

palabra de la secuencia puede ser necesario obtener información acerca de la secuencia completa.

En problemas de speech recognition, se recibe como input una secuencia de señales provenientes de una onda sonora con el objetivo de predecir una secuencia de segmentos fonéticos junto a sus respectivas probabilidades.

Las redes recurrentes también pueden ser combinadas con redes convolucionales para la generación de descripciones de imágenes y vídeos.

El entrenamiento de una red recurrente es similar al de las redes descritas previamente. También se usa back propagation, pero con algunas diferencias fundamentales. Debido a que los parámetros son compartidos por todos los pasos de tiempo de la red, el gradiente respecto a cada output no depende sólo del paso de tiempo actual, sino que también de los pasos de tiempo previos. Por ejemplo, para calcular el gradiente en el paso de tiempo $t = 4$ se necesitaría ir tres pasos hacia atrás y sumar los gradientes. Este procedimiento es conocido como back propagation a través del tiempo (en inglés, Back Propagation Through Time (BPTT)).

Dentro de las extensiones existentes de redes recurrentes, es posible encontrar las redes Long Short Term Memory (LSTM) [8], que permiten solventar el problema del desvanecimiento del gradiente en la redes recurrentes habituales [4]. Además, son útiles para afrontar problemas en los que no sólo pueden existir dependencias entre elementos sucesivos de una determinada secuencia, sino entre elementos que pueden estar muy separados dentro de la misma, lo que se conoce como dependencias de largo plazo. Estas redes están especialmente diseñadas para evitar tal problema.

Las redes LSTM tienen una estructura similar a la de las redes recurrentes básicas, ya que también están integradas por un estado oculto y una salida que se repiten en el tiempo. Sin embargo, la mecánica tras el funcionamiento del estado oculto en una red LSTM es más compleja. El elemento clave dentro de esta clase de red es lo que se conoce como estado de la celda (*cell state*). La red LSTM puede modificar el estado de la celda añadiendo o modificando información por medio de estructuras llamadas compuertas (*gates*). La figura 2.8 muestra la estructura general de una red LSTM.

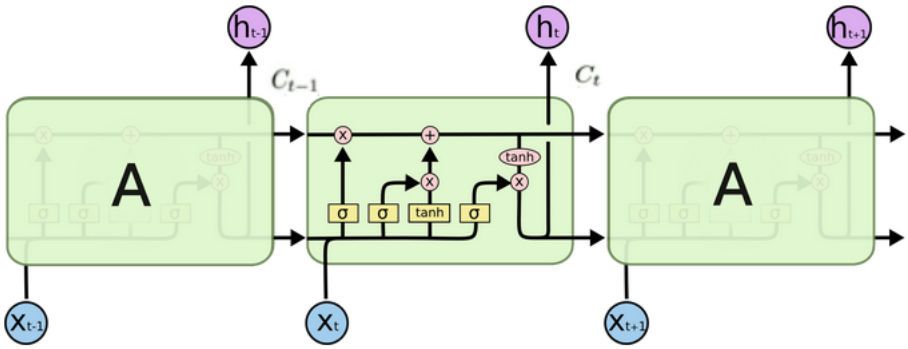


Figura 2.8: Estructura de una red LSTM estándar. La celda de la red es denotada por A. x_t es el input recibido en el tiempo t y h_t es la salida generada en el tiempo t. C_t representa el estado de la celda en el tiempo t. Fuente: [8].

En la figura 2.9 se detalla la simbología utilizada en la figura 2.8.

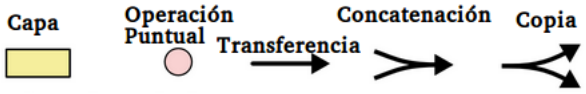


Figura 2.9: Simbología utilizada en la figura 2.8. Fuente: [8]

Cada compuerta de una celda está conformada por una capa con función de activación sigmoidal y una operación elemento-a-elemento de multiplicación. Como se mencionó previamente, la función sigmoidal produce números entre cero y uno, describiendo cuánto de cada componente debe dejarse pasar. Una red LSTM estándar tiene tres de estas compuertas.

Cuando un input ingresa a la red en un determinado tiempo *t*, el primer paso es decidir que información debe eliminarse del estado anterior de la celda, denotado por el vector C_{t-1}. Esta tarea es llevada a cabo por una capa con función de activación sigmoidal que se denomina compuerta de olvido (*forget gate layer*), denotada por la variable f_t. Dicha compuerta recibe a h_{t-1} y a x_t como parámetros y produce un número entre cero y uno para cada componente del estado de la celda. Si la función produce un cero, quiere decir que no debe almacenarse nada de la información actualmente contenida, mientras que un valor de uno implica que debe mantenerse toda la información disponible. Matemáticamente, la salida de la compuerta de olvido está dadq por:

$$f_t = \sigma(W_f \cdot (h_{t-1}, x_t) + b_f)\,. \tag{2.11}$$

Por ejemplo, si se tiene un modelo de lenguaje para predecir la siguiente palabra de una sentencia en base a todas las anteriores, el estado de la celda puede estar almacenando el género del sujeto de la sentencia, de manera que sea posible predecir los pronombres adecuados a futuro. Así, cuando en la sentencia se presente un nuevo sujeto, lo adecuado será olvidar el género del sujeto anterior.

El paso siguiente es decidir que parte de la nueva información recibida debe ser almacenada en el estado de la celda. Esta operación está basada en los dos componentes restantes de esta celda: Una capa con función de activación sigmoidal, que se denomina compuerta de entrada (*input gate layer*), denotada por la variable i_t , la cual decide que componentes deben ser actualizados, y una capa con función de activación tanh que crea un vector con los nuevos valores candidatos de la celda, $\tilde{C}_t$, que pueden ser añadidos a su estado. Matemáticamente:

$$i_t = \sigma(W_i \cdot (h_{t-1}, x_t) + b_i)\,. \tag{2.12}$$

$$\tilde{C}_t = tanh(W_c \cdot (h_{t-1}, x_t) + b_C)\,. \tag{2.13}$$

Luego de esto, es necesario actualizar el nuevo estado de la celda, C_t, a partir del estado previo de la celda, C_{t-1}. Las compuertas anteriores ya han tomado las decisiones acerca de que información retener y cual olvidar. Para llevar a cabo la actualización, se multiplica el estado viejo por f_t, descartando aquella información que decidió olvidarse. El resultado de la multiplicación anterior es después añadido al producto entre i_t y $\tilde{C}_t$. Conceptualmente, ésta última operación representa los nuevos valores candidatos escalados por cuánto se ha decidido actualizar cada valor del estado de la celda. Matemáticamente:

$$C_t = f_t \cdot C_{t-1} + i_t \cdot \tilde{C}_t\,. \tag{2.14}$$

Para terminar con el procesamiento del input x_t, es necesario determinar que es lo que irá en el output h_t. Este output estará basado en el estado de la celda, pero será una

versión ”filtrada”, en el sentido de que no corresponde al estado completo. Para ello, primero se aplica una capa con función de activación sigmoidal (la última compuerta de la celda) sobre el input actual y el output anterior. El resultado de esta operación, denotado por la variable o_t, permite decidir que es lo que irá en el output. En forma paralela, se aplica la función tanh sobre el estado actual de la celda, llevando a que cada uno de los elementos del estado tenga valores entre -1 y 1. Los resultados de las dos operaciones anteriores son multiplicados entre sí, permitiendo que en el output sólo vayan los componentes que se decidieron, lo que es denotado por h_t. En términos matemáticos, las ideas anteriores quedan expresadas como:

$$o_t = \sigma(W_o \cdot (h_{t-1}, x_t) + b_o)\,. \tag{2.15}$$

$$h_t = o_t \cdot tanh(C_t)\,. \tag{2.16}$$

Siguiendo el ejemplo de un modelo de lenguaje, si el input actual de la red es un sujeto y el input siguiente es un verbo, entonces puede ser importante que el output h_t almacene información acerca de si el sujeto es singular o plural para saber como debe ser conjugado el verbo que sigue a continuación.

2.3. ¿Qué es el deep learning?

El deep learning (en español, *aprendizaje profundo*) [9] es una rama del área de machine learning que ha permitido lograr importantes avances en problemas de visión computacional (como reconocimiento y localización de objetos) y procesamiento de secuencias (como reconocimiento de voz y procesamiento natural de lenguaje).

Una de las principales características del deep learning es la forma en que sus modelos aprenden a partir de los datos que reciben. En el área de machine learning tradicional, los atributos de los ejemplos recibidos por un modelo deben ser declarados explícitamente para que este sea capaz de generar una respuesta. Sin embargo, muchas veces no es trivial determinar de antemano cuáles son los mejores atributos para encontrar la respuesta deseada, o incluso, ni siquiera es posible extraerlos manualmente a partir de los datos en bruto, como ocurre precisamente en problemas que manejan imágenes

y secuencias. Es en este punto cuando los métodos tradicionales de machine learning empiezan a ser poco adecuados para resolver los problemas que se le plantean y el deep learning entra a jugar un rol fundamental. Con las técnicas de aprendizaje que el deep learning propone, no es necesario definir los atributos explícitamente, pues dichas técnicas permiten crear modelos capaces de aprender automáticamente cuáles son los atributos fundamentales para realizar buenas predicciones, permitiendo así disminuir los costos que esta tarea implica y enfocarse directamente en el diseño de los modelos. De acuerdo a [10], los algoritmos existentes en el área de deep learning buscan explotar la estructura desconocida presente en los datos recibidos por un modelo con el fin de descubrir buenas representaciones, lo que, por lo general, se lleva a cabo en múltiples niveles, esto es, en diferentes capas de la red, con atributos de alto nivel definidos a partir de atributos de bajo nivel. De acuerdo a esta misma fuente, el deep learning presenta un rendimiento destacado en problemas de áreas en que los datos son análogos, lo que hace referencia a que los datos no están estructurados, sino que corresponden a imágenes de píxeles de intensidad, documentos de texto y archivos de sonido.

Actualmente, cuando se habla de deep learning, se está haciendo referencia a redes neuronales artificiales de gran extensión. El término profundidad es una forma de caracterizar la cantidad de capas que ella tiene, de tal manera que mientras más capas tenga una red, mayor será su profundidad. Sin embargo, al aumentar la profundidad de una red, también lo hace la cantidad de parámetros que deben ser entrenados, llevando a que el proceso en su conjunto sea muy costoso. Afortunadamente, los permanentes avances de la tecnología han ayudado a sobrellevar estas dificultades. Actualmente, se dispone de computadores lo suficientemente poderosos como para entrenar redes neuronales artificiales profundas en forma eficiente, lo que también va de la mano con la gran abundancia de datos que se pueden encontrar hoy en día. Experimentalmente, se ha comprobado que la mayor parte de las redes neuronales que son implementadas para abordar problemas de visión computacional y procesamiento de secuencias continúan aumentando su rendimiento en términos de precisión a medida que sus profundidades aumentan y son alimentadas con más datos. Esto resulta diferente de muchas otras técnicas de machine learning que suelen alcanzar un peak en su performance, por mucho que se disponga de más datos para el entrenamiento de sus modelos. Sin embargo, es necesario precisar que los beneficios que actualmente se observan provienen del aprendizaje supervisado, esto es, problemas en que los ejemplos de entrenamiento contienen la respuesta que se desea predecir (es decir, datos etiquetados). En el ámbito del aprendizaje

no supervisado, los beneficios están aún en progreso a medida que el campo madura con la abundancia de datos no etiquetados.

2.4. Entrenamiento de redes neuronales en GPU

Tal como se señaló en la sección anterior, una de las bases más importantes para que la implementación de modelos sea eficiente es el trabajo en conjunto con uno o más computadores de alto rendimiento, lo que en este contexto se traduce en poder contar con máquinas que dispongan de uno o más dispositivos GPU [11]. De todas las tareas que deben ejecutarse para la obtención de un buen modelo, es la fase de entrenamiento la que requiere un uso intensivo del hardware disponible. Esto se debe a que en el proceso de entrenamiento de una red neuronal deben ejecutarse muchas operaciones sobre y entre matrices, como por ejemplo, multiplicaciones, operaciones que tienen asociada una complejidad temporal de orden $O(n^3)$, por lo que puede decirse que es un proceso computacionalmente costoso. Al aumentar la profundidad de la red, y en consecuencia, la cantidad de parámetros que esta posee, el número de multiplicaciones de matrices que deben llevarse a cabo también aumenta. Ante esto, es fundamental acelerar el proceso de cómputo. Una estrategia adecuada sería poder ejecutar la mayor cantidad posible de estas operaciones en forma paralela en vez de realizarlas en forma secuencial, como se hace en una CPU. Esto puede ser conseguido por medio de una o más GPU's.

Si bien las GPU's fueron inicialmente concebidas como dispositivos para permitir una adecuada visualización y manipulación de imágenes, vídeos y contenido multimedia, puede decirse que estos dispositivos tienen un punto en común con el entrenamiento de redes neuronales: El trabajo con matrices. Computacionalmente, las imágenes y los demás recursos mencionados son representados como matrices y para poder manipularlos (esto es, cuando se aplican rotaciones, se redimensionan y/o se distorsionan), es necesario llevar a cabo operaciones matemáticas sobre ellos. Luego, se puede decir que las GPU's están más especializadas para trabajar con operaciones y transformaciones sobre matrices, a diferencia de las CPU's que están principalmente diseñadas para operaciones de propósito general. Notar, sin embargo, que sólo las operaciones de tipo matricial son ejecutadas en GPU, mientras que las demás son computadas normalmente en CPU.

En la tabla 2.1 es posible observar las principales diferencias que existen entre una

GPU y una CPU [12].

Cuadro 2.1: Comparación entre las características de una CPU y una GPU.

CPU	**GPU**
Posee pocos núcleos	Posee cientos o miles de núcleos
Los núcleos son complejos en términos arquitecturales: Alta cantidad de registros por cada uno	Los núcleos son simples en términos arquitecturales: Baja cantidad de registros por cada uno
Procesador de propósito general	Procesador destinado a manejar tareas específicas
Ejecuta las instrucciones secuenciales de un programa	Ejecuta las instrucciones factibles de ser paralelizadas en un programa
Posee un mayor frecuencia de operación	Posee una menor frecuencia de operación

Además, el entrenamiento de un modelo en GPU puede diferir en varios ordenes de magnitud con respecto al de una CPU en términos de tiempo. Si el tiempo empleado para entrenar un modelo en CPU es del orden de varias horas, el entrenamiento del mismo modelo en GPU puede ser de no más de algunos minutos, como puede ocurrir en el caso del entrenamiento de redes neuronales convolucionales. No obstante es necesario considerar que los beneficios del cómputo paralelo de una GPU aumentan mientras mayor sea la posibilidad de paralelizar las operaciones que se deben ejecutar. Además, la diferencia en rendimiento con una CPU será posible de apreciar mientras más complejo sea el modelo en cuestión.

2.5. ¿Cómo se procesan los datos en una GPU?

En la sección anterior, se explicó el rol fundamental que cumplen las GPU’s en la aceleración del entrenamiento de redes neuronales. ¿Pero cómo operan internamente

estos dispositivos?

A grandes rasgos, el flujo seguido para el procesamiento de datos en GPU puede resumirse en los siguientes pasos:

- Copiar los datos necesarios desde memoria CPU hacia memoria GPU.
- Cargar las instrucciones que manipulan esos datos en GPU, manteniendo dichos datos en caché para ofrecer un mejor rendimiento.
- Copiar los resultados obtenidos desde la memoria de la GPU hacia la memoria de la CPU.

En la actualidad, existen dos modelos de programación de instrucciones para GPU: CUDA y OpenCL [13]. Estos modelos representan estándares de como se ejecuta un programa en esta clase de dispositivos. En este apartado, se describe el modelo CUDA, debido a que la arquitectura de la GPU que se utilizará para desarrollar los experimentos de esta memoria es compatible con el mencionado modelo.

CUDA, desarrollado por NVIDIA, opera bajo el modelo SIMD (*single instruction, multiple data*), el cual hace referencia a que una misma instrucción puede ser ejecutada de forma paralela sobre diferentes datos en diferentes unidades de procesamiento.

Una GPU que posee una arquitectura compatible con el modelo CUDA está conformada por múltiples multiprocesadores o *streaming multiprocessors* (SM). Cada SM está integrado por varios *streaming processors* (SP). Dos o más SM's dan origen a lo que se conoce como bloque y comparten la misma memoria caché y lógica de control. Cada SM recibe una cantidad elevada de flujos de ejecución. La figura 2.10 muestra una arquitectura de ejemplo como la que acaba de ser descrita. Es posible ver que la GPU mostrada está integrada por 8 bloques, 16 SM's y 128 SP's.

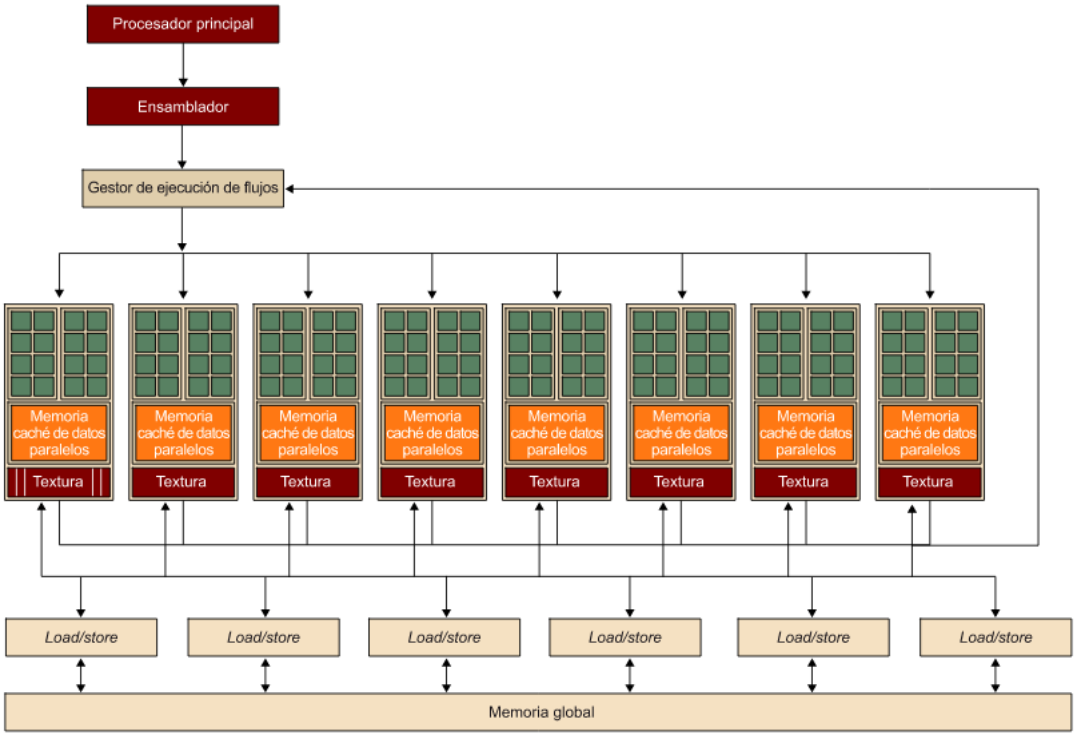


Figura 2.10: Arquitectura ejemplo de una GPU compatible con el modelo CUDA. Cada cuadrado celeste corresponde a un SP. Cada conjunto de 8 SP's da origen a un SM. 2 SM's forman un bloque. Fuente: [13].

Aquellos segmentos de un programa en que el paralelismo de datos es más bien escaso se ejecutan en CPU, mientras que aquellos segmentos que tienen asociados un alto nivel de paralelismo de datos son ejecutados en GPU. Es el compilador provisto por NVIDIA, llamado nvcc, el que se encarga de distribuir las instrucciones en cada dispositivo dependiendo del tipo que sean cuando el código del programa es compilado. El código que es procesado por la CPU es lenguaje ANSI C, al igual que el procesado por la GPU, pero este último está integrado por palabras claves que definen funciones que operan de forma paralela con los datos. Dichas funciones son denominadas kernels. Para sacar real provecho del paralelismo de datos, los kernels deben generar una cantidad de flujos de ejecución del orden de las decenas o centenas de miles de flujos.

La ejecución del programa empieza en la CPU. Cuando un kernel es invocado, la ejecución se traslada hacia la GPU y este genera un amplio conjunto de flujos de ejecución[1]. Tal conjunto de flujos es denominado *grid*. Un grid puede abarcar uno o más

[1]Por ejemplo, cada una de las sumas y productos ejecutados en el proceso de multiplicación de dos

bloques de la GPU. A su vez, cada bloque concentra uno o más flujos de ejecución. Todos los bloques de un mismo grid contienen la misma cantidad de flujos y están organizados de la misma forma. Se puede decir que un kernel ha finalizado cuando todos los flujos de ejecución generados por él en el grid que le corresponde lo han hecho, ante lo cual los flujos de un bloque deben sincronizarse previamente. Cuando el kernel ha finalizado, se retoma el control del programa en CPU hasta que otro kernel aparezca en el camino. La figura 2.11 muestra como se ejecuta un programa de estas características. La figura 2.12 resume la estructura de un grid.

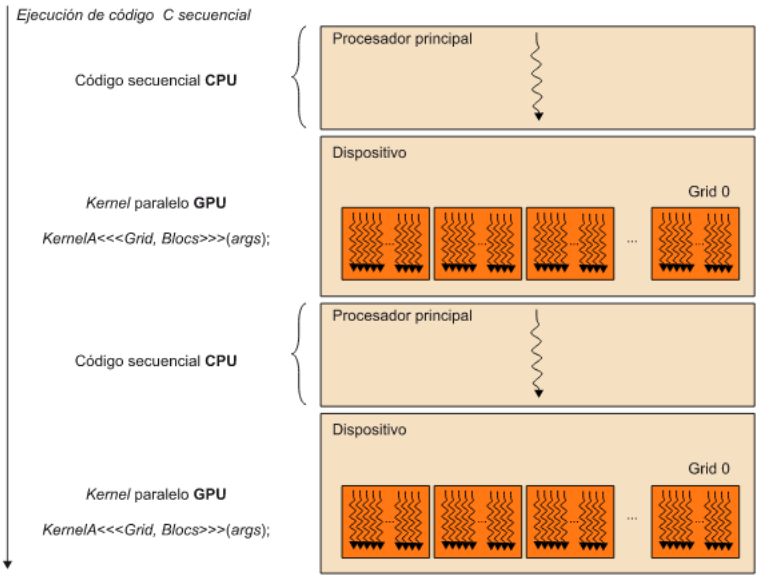


Figura 2.11: Ejecución de un programa en una GPU que opera bajo el modelo CUDA. Fuente: [13].

matrices corresponde a un flujo de ejecución diferente.

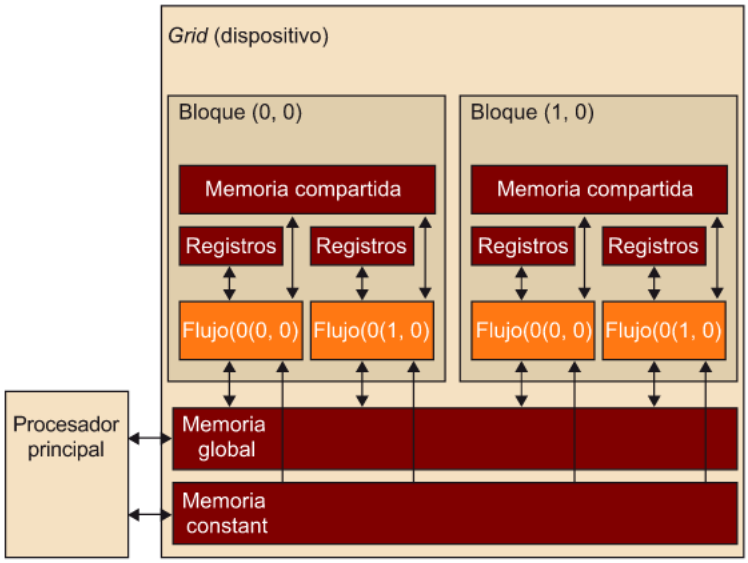


Figura 2.12: Estructura interna de un grid. Fuente: [13].

Considerar como ejemplo de una operación en GPU la multiplicación de dos matrices. Para que el kernel asociado a esta operación pueda ser ejecutado, el programa debe seguir los pasos listados a continuación:

- Asignar memoria en la GPU: Se debe reservar espacio en la memoria de la GPU tanto para las dos matrices que serán multiplicadas como para el resultado.
- Transferencia de datos: Se transfieren las matrices desde la CPU hacia la memoria previamente asignada en la GPU.
- Ejecutar el kernel: Llevar a cabo la multiplicación de las matrices en GPU.
- Transferencia del resultado: Copiar la matriz resultante de la multiplicación desde la GPU hacia la CPU y liberar la memoria correspondiente en la GPU, siempre que no sea necesario utilizarla nuevamente.

2.6. Breve revisión de frameworks a estudiar

2.6.1. Theano

Theano [14] es una librería de código abierto implementada en Python que permite definir, optimizar y evaluar expresiones matemáticas que involucran el manejo de arreglos multidimensionales. Theano fue desarrollado y es mantenido por el Laboratoire d'Informatique des Systèmes Adaptatifs de la Université de Montréal desde el año 2007. El framework opera bajo la licencia BSD.

Theano es un framework multiplataforma y sólo cuenta con dos interfaces, ambas disponibles en Python 2 y Python 3: Una propia y otra accesible a través de Keras. Sus funcionalidades pueden ser ejecutadas tanto en CPU como en una GPU individual y múltiples GPU's. Además, ofrece soporte limitado para el uso de paralelismo multinúcleo en CPU. Si bien inicialmente sólo era compatible con arquitecturas CUDA, actualmente también es compatible con arquitecturas OpenCL gracias al uso de su propio backend, libgpuarray. Permite la diferenciación automática de expresiones y la implementación de redes feedforward, redes convolucionales, redes recurrentes, máquinas de Boltzmann y redes de creencia profunda (DBN's). Está diseñando para ser altamente compatible con las estructuras de datos provistas por NumPy y SciPy.

2.6.2. TensorFlow

TensorFlow [15] es una librería de código abierto escrita en Python y C++ para la ejecución de operaciones de computación numérica, resolución de problemas de machine learning e implementación de redes neuronales artificiales. Fue originalmente desarrollado por ingenieros e investigadores del Google Brain Team en conjunto con la organización de investigación de Machine Inteligence de Google en el año 2015. Este framework opera bajo la licencia Apache 2.0. Es usado por empresas como Uber, Dropbox, Ebay, Intel, Coca-Cola y Twitter, entre otras. TensorFlow posee una estructura tal que permite al usuario ejecutar múltiples procesos en una o más CPU's o GPU's disponibles ya sea en un computador de escritorio, en un servidor o en un dispositivo móvil a través de su API. Actualmente sólo es compatible con arquitecturas CUDA y el soporte para OpenCL se encuentra en desarrollo. Está disponible para Linux, Mac

OS X, Windows y plataformas móviles, incluyendo Android y iOS. TensorFlow provee API's disponibles en múltiples lenguajes tanto para la construcción como la ejecución de modelos. Tales lenguajes son: Python (una propia y otra accesible a través de Keras), C, C++, Java y Go, si bien sólo se garantiza estabilidad en las API's de Python y C [16]. Al igual que Theano, permite la diferenciación automática de expresiones y la implementación de los mismos tipos de redes neuronales.

2.6.3. CNTK

CNTK (también denominado Microsoft Cognitive Toolkit) [17] es una herramienta de código abierto, implementada en C++ y que fue creada en el año 2016 por Microsoft Research para el trabajo con problemas de deep learning. Cuenta con una licencia del Massachusetts Institute of Technology (MIT) y es usado en productos como Skype, Cortana, Bing, Xbox, entre otros.

CNTK cuenta con interfaces tanto en lenguaje C++ como Python (una propia y otra accesible a través de Keras), o bien, puede ser empleado individualmente por medio de su propio lenguaje, BrainScript. Adicionalmente, es posible evaluar los modelos implementados con esta librería en los lenguajes C#, .NET y Java. Puede ser ejecutado en diversos entornos, ya sea en una CPU, en una GPU individual o paralelamente en múltiples GPU's y/o máquinas. Sin embargo, actualmente sólo es compatible con arquitecturas CUDA. Puede ser instalado en Windows, Linux y Mac OS X. Es factible trabajar con redes feedforward, redes convolucionales, redes recurrentes, redes recurrentes para mapeo sequence-to-sequence y redes LSTM, pero no con máquinas de Boltzmann ni DBN's. También se ofrece soporte para aprendizaje de refuerzo, redes adversarias generativas, aprendizaje supervisado y no supervisado y diferenciación automática de expresiones. Además, entrega interoperabilidad con NumPy.

2.6.4. Keras

Keras [18] es una librería de alto nivel y de código abierto escrita en Python para la implementación de redes neuronales artificiales y que puede usar a Theano, TensorFlow o CNTK como backends. Fue creada por Francois Chollet en el año 2015 y opera bajo la licencia del MIT.

Keras ofrece soporte para la diferenciación automática de expresiones y la implementación de redes feedforward, redes convolucionales, redes recurrentes y todas las arquitecturas que sus backends son capaces de implementar, así como también para la combinación de estas arquitecturas. Además, los modelos definidos pueden ser entrenados tanto en CPU como en una o múltiples GPU's. En general, es compatible sólo con arquitecturas CUDA, excepto si se usa Theano como backend, caso en el cual también resulta compatible con arquitecturas OpenCL. Este framework puede ser instalado en Linux, Mac OS X y Windows y ofrece interfaces para Python y R. Actualmente, es compatible con Python desde la versión 2.7 hasta la 3.5.

Capítulo 3

Estado del Arte

En marzo de 2016, Bahrampour et al. [19] realizaron un estudio comparativo de cinco frameworks para la implementación de métodos de deep learning: Caffe, Neon, TensorFlow, Theano y Torch. Para efectuar dicho estudio, los frameworks fueron comparados en base a tres aspectos:

- Extensibilidad: Capacidad del framework de incorporar diferentes tipos de arquitecturas de deep learning (por ejemplo, redes convolucionales, redes feedforward y/o redes recurrentes), uso de diferentes procedimientos de entrenamiento (por ejemplo, pre-entrenamiento no supervisado por capas y/o aprendizaje supervisado) y uso de diferentes algoritmos convolucionales.
- Utilización de hardware: Niveles de uso de CPU y GPU por parte del framework, buscando verificar que estos recursos estén siendo explotados en la mayor medida posible y que por tanto su uso sea eficaz.
- Rapidez: Qué tan rápido es el framework en el entrenamiento y evaluación de modelos. Para este criterio, se consideraron dos métricas: Forward Time (tiempo necesario para que un determinado mini-batch se propague por la red y genere el output correspondiente) y Gradient Computation Time (tiempo necesario para completar una iteración).

Así, los autores implementaron diversos tipos de arquitecturas considerando dos casos: Uso de CPU multithread y uso de GPU. En el caso de uso de CPU, se utilizó un dispositivo Intel Xeon E5-1650 v2 3.50 GHz de 6 núcleos y 12 threads por cada uno.

Para el caso de uso de GPU, se utilizó una tarjeta gráfica Nvidia Titan X, la cual cuenta con 3072 núcleos CUDA y 12 GB de memoria.

Los autores argumentan por qué escogieron tales frameworks y no otros. En primer lugar, aducen a que Caffe, Theano y Torch son los tres frameworks que cuentan con mayor interés por parte de la comunidad de desarrollo, en base a la cantidad de usuarios en grupos de Google asociados a dichos frameworks al momento de la elaboración de esta publicación (Caffe: 4220, Theano: 2827, Torch: 1874) y al número de contribuyentes para cada framework en su respectivo repositorio Github (Caffe: 172, Theano: 207, Torch: 77). Por otro lado, Neon fue escogido dado su buen rendimiento en el entrenamiento de redes convolucionales. Además, TensorFlow fue incluído dado al creciente interés de la comunidad en él.

Para la ejecución de pruebas en GPU, cada framework hizo uso de diferentes librerías como backends, las que son detalladas a continuación:

- Torch: Se experimentó con tres combinaciones: CUDA 7.5 + cuDNN v3, cunn y fbcunn. [1]
- Caffe: CUDA 7.5 + cuDNN v3.
- Theano: CUDA 7.5 + cuDNN v3, conv-fft [2] y cuconv. [3]
- TensorFlow: CUDA 7.0 + cuDNN v2.
- Neon: Backend propio.

En lo que respecta a las arquitecturas implementadas, se trabajó primeramente con el entrenamiento de stacked autoencoders y redes convolucionales, aplicando estas arquitecturas sobre los datasets MNIST e ImageNet con la finalidad de resolver problemas de clasificación multiclase en el caso de las redes convolucionales y realizar preentrenamiento no supervisado por capas y fine-tunning supervisado en el caso de los autoencoders. También se entrenó una red LSTM para llevar a cabo un proceso de análisis de sentimiento sobre el dataset provisto por el sitio web IMDB, buscando evaluar si una crítica determinada es positiva o negativa, lo que corresponde a un problema de clasificación binaria.

[1] cunn y fbcunn sólo fueron considerados para los casos de LeNet y AlexNet.
[2] Sólo para el caso de LeNet.
[3] Sólo para el caso de AlexNet.

Para el caso de redes convolucionales, se escogieron las siguientes arquitecturas:

- LeNet-5 [20]: Esta red está diseñada para recibir imágenes de 32 x 32 píxeles como inputs. En su versión original, está conformada por siete capas, que se definen a continuación:

  - La primera es una capa convolucional de 6 filtros de 5 x 5, cada uno, lo que como resultado genera 6 feature maps de tamaño 28 x 28, cada uno. Esta primera capa posee un total de 156 parámetros entrenables y 122.304 conexiones.

  - La segunda es una capa de pooling con pooling size de tamaño 2 x 2, que suma los valores del campo receptivo sobre el cuál es aplicado, luego multiplica dicha suma por un parámetro entrenable, le añade un bias y finalmente aplica la función sigmoidal sobre el resultado anterior. Así, su output corresponde a 6 feature maps de tamaño 14 x 14, cada uno. Esta capa posee 12 parámetros entrenables y 5.880 conexiones.

  - La tercera es una capa convolucional integrada por 16 filtros de tamaño 5 x 5, por lo que su output corresponde a 16 feature maps de tamaño 10 x 10, cada uno. Esta capa posee 1.516 parámetros entrenables y 151.600 conexiones.

  - La cuarta es una capa de pooling con pooling size de tamaño 2 x 2. Como output, genera 16 feature maps de tamaño 5 x 5, cada uno. Ésta cuarta capa posee 32 parámetros entrenables y 2.000 conexiones.

  - La quinta es una capa convolucional compuesta por 120 filtros de 5 x 5, cada uno. En consecuencia, el output generado consta de 120 feature maps de tamaño 1 x 1, cada uno. Existe una conexión total entre la cuarta y la quinta capa. Además, posee un total de 48.120 parámetros entrenables.

  - La sexta es una capa totalmente conectada a la quinta, compuesta por 84 neuronas. Posee 10.164 parámetros entrenables.

    Notar que desde la primera hasta la sexta capa, se usa la función de activación tanh, excepto en aquellas en que se especificó una función distinta.

  - La séptima y última capa (es decir, la capa de salida) está compuesta por 10 unidades RBF, una por cada clase. Está totalmente conectada a la capa

anterior.

Así, esta arquitectura está conformada por un total de 60.000 parámetros entrenables. No obstante, los autores realizaron algunos cambios en la red: tanh, utilizada como función de activación en las capas ocultas, es reemplazada por la función ReLU, mientras que en la capa de salida utilizaron la función de activación softmax en vez de unidades RBF, llevando a que el modelo resultante estuviese integrado por 60.850 parámetros entrenables.

- AlexNet [21]: Conformada por ocho capas con parámetros entrenables, siendo las cinco primeras de tipo convolucional y las tres últimas capas totalmente conectadas. En las siete primeras capas, se utiliza la función de activación ReLU, mientras que en la última, la función softmax. Además, se aplicó agrupamiento en tres de las capas convolucionales y se llevó a cabo regularización en las capas totalmente conectadas por medio de dropout. Tanto la primera como la segunda capa convolucional están seguidas por una capa de tipo LRN. Además, existen capas de max pooling después de cada capa LRN y a continuación de la quinta capa convolucional. Esta red recibe imágenes de dimensiones 224 x 224 x 3 como inputs. Las características de cada capa se especifican a continuación:

  - La primera capa convolucional posee 96 filtros de tamaño 11 x 11, cada uno, y stride de 4 píxeles.
  - La segunda capa convolucional tiene 256 filtros de tamaño 5 x 5, cada uno.
  - La tercera capa convolucional posee 384 filtros de tamaño 3 x 3, cada uno.
  - La cuarta capa convolucional posee 384 filtros de tamaño 3 x 3, cada uno.
  - La quinta capa convolucional tiene 256 filtros de tamaño 3 x 3, cada uno.
  - Tres capas totalmente conectadas consecutivas. Las dos primeras contienen 4096 neuronas, cada una, y la tercera (que es la capa de salida) posee 1000 neuronas.

    Esta arquitectura está integrada por 60 millones de parámetros. Sin embargo, no fue posible utilizar capas de tipo LRN ni agrupamiento en las pruebas realizadas con Neon y TensorFlow, tanto en CPU como en GPU, dadas las limitaciones ofrecidas por estos frameworks, incapaces de soportar tales características. Además, en el caso de utilización de GPU, las librerías cunn

y fbcunn no soportan agrupamiento. Luego, por transparencia, las pruebas realizadas con Caffe, Theano y Torch fueron llevadas a cabo tanto en presencia como en ausencia de agrupamiento y capas LRN.

Para el caso de autoencoders, se estudió, en primer lugar, un conjunto de tres autoencoders diferentes (de ahora en adelante, denotados por AE1, AE2 y AE3), cada uno compuesto por un encoder y un decoder con pesos compartidos, con el fin de realizar pre-entrenamiento no supervisado por capas, de tal manera que el output del encoder de un autoencoder determinado es el input del siguiente autoencoder. Para este procedimiento, se utilizó la función de activación sigmoidal y MSE como función de pérdida. En segundo lugar, se llevó a cabo fine-tunning supervisado sobre un stacked autoencoder (de ahora en adelante denotado como SE), conformado por los tres autoencoders anteriormente pre-entrenados. En esta segunda fase, se utilizó la función de activación softmax en la capa de salida y categorical cross-entropy como función de pérdida. Notar, sin embargo, que Neon y Caffe, al momento de desarrollarse esta publicación, no soportaban pesos compartidos, por lo que los parámetros de encoders y decoders resultaron ser diferentes en cada uno de los autoencoders. Se realizaron dos experimentos, variando la cantidad de capas ocultas usadas por el encoder de cada autoencoder. El encoder utilizado para AE1 tuvo una dimensión de 400 capas ocultas en el primer experimento y 800 en el segundo; el encoder de AE2 utilizó 200 capas ocultas en el primer experimento y 1000 en el segundo; el encoder de AE3 usó 100 capas ocultas en el primer experimento y 2000 en el segundo. No se indicó la cantidad de parámetros de los modelos resultantes. Notar que en el pre-entrenamiento de los autoencoders sólo se midió el gradient computation time.

Para el caso de la red LSTM, se utilizó una capa LSTM seguida por una capa de average pooling y a continuación la capa de salida, la cual utiliza regresión logística como función de activación [22]. No se indicó el número de parámetros de la arquitectura. Además, al momento de desarrollarse esta publicación, no fue posible trabajar con Caffe, dado a su incapacidad de soportar arquitecturas cíclicas, como es el caso de las redes LSTM. Por otra parte, Neon y TensorFlow tampoco fueron utilizados, al ser incapaces de procesar mini-batches con secuencias de tamaños diferentes. Así, sólo se compararon los rendimientos de Theano y Torch.

Junto con la comparación de diferentes frameworks y arquitecturas, los autores experimentaron de forma simultánea con diferentes tamaños de batch en el proceso de

entrenamiento de los modelos. Se usaron batches de tamaño 32, 64, 128, 256, 512 y 1024, excepto para la red LSTM, en la cual se usó un tamaño de batch fijo de 16, y para el caso de autoencoders, en que se trabajó únicamente con batches de tamaño 64. Además, en el caso de utilización de CPU, se estudió cada framework usando 1, 6 y 12 threads, excepto Neon y Caffe, para los que se trabajó solamente con 1 y 12 threads, respectivamente, debido a que al primero no soporta multithreading, mientras que el segundo sólo puede trabajar con una cantidad fija de threads, determinada en el momento de su instalación. También, en el caso de la red LSTM, cada framework trabajó sólo con 6 threads en CPU.

Una vez finalizado el proceso de experimentación, los autores concluyeron que Theano y Torch son los frameworks más extensibles, debido a que ambos ofrecen soporte para una gran cantidad de arquitecturas profundas, así cómo también para una gran cantidad de librerías. Respecto a la ejecución de pruebas usando CPU como recurso principal, resultó ser que, para el entrenamiento y testeo de cualquier tipo de arquitectura, Torch alcanzó el mejor rendimiento, seguido por Theano, siendo Neon el que obtuvo el peor rendimiento. Respecto a la evaluación de redes convolucionales en GPU, fue Torch quién consiguió los mejores resultados, seguido por Theano. Además, para el proceso de entrenamiento de las mismas, también por medio de GPU, se observó que Theano fue el más rápido en el entrenamiento de redes pequeñas, mientras que Torch fue el más rápido entrenando redes extensas, seguido de cerca por Neon. En lo que respecta al entrenamiento y evaluación de la red LSTM en GPU, fue Theano quién consiguió el mejor rendimiento. En la tabla 3.1 se muestra un resumen con los frameworks que consiguen los mejores resultados sobre cada una de las arquitecturas estudiadas, mostrando la mayor cantidad información posible para cada configuración. Notar que los autores sólo informaron los tiempos específicos obtenidos para LeNet, AlexNet, la red LSTM y los autoencoders con batches de tamaño 64, 256, 16 y 64, respectivamente.

Cuadro 3.1: Frameworks que consiguen los mejores resultados en cada una de las arquitecturas estudiadas en [19]. Sólo se informaron tiempos para batches de tamaño 64, 256, 16 y 64 para LeNet, AlexNet, la red LSTM y los autoencoders, respectivamente. Para los autoencoders, se indica entre paréntesis la cantidad de capas ocultas de cada uno. En todo los casos, el mejor rendimiento fue alcanzando en conjunto con las librerías CUDA 7.5 y cuDNN v3. Notación: (g): Gradient computation time, (f): Forward time.

Arquitectura	**Dataset**	**Mejor(es) framework(s)**	**Tiempo promedio de procesamiento por mini-batch [ms]**
LeNet-5	MNIST	(g, f) Theano (f) Torch	(g) 1,4 (f) 0,5
AlexNet	ImageNet	(g, f) Torch	(g) 286,3 (f) 92,5
LSTM	IMDB	(g, f) Theano	(g) 16,72 (f) 4,66
AE1 (400)	MNIST	(g) Theano (g) Torch	(g) 0,6
AE1 (800)	MNIST	(g) Caffe (g) Torch	(g) 0,8
AE2 (200)	MNIST	(g) Theano	(g) 0,4

AE2 (1000)	MNIST	(g) Torch	(g) 0,9
AE3 (100)	MNIST	(g) Theano	(g) 0,3
AE3 (2000)	MNIST	(g) Caffe	(g) 1,5
SE (400-200-100)	MNIST	(g, f) Theano	(g) 0,4 (f) 0,2
SE (800-1000-2000)	MNIST	(g) Theano (f) Torch	(g) 1,1 (f) 0,7

Para complementar el resumen anterior, las tablas 3.2 y 3.3 muestran, para cada tamaño de batch posible usado en GPU, el framemework más rápido en base a los tiempos medidos en los casos de LeNet y AlexNet, respectivamente.

Cuadro 3.2: Framework más rápido para cada tamaño de batch posible usado en GPU, caso LeNet. Notación: (g): Gradient Computation time, (f): Forward time, (g, f): Tanto gradient computation time como forward time.

Tamaño batch	**Mejor(es) framework(s)**
32	(g, f) Theano
64	(g, f) Theano (f) Torch
128	(g, f) Theano
256	(g) Neon (f) Theano
512	(g) Neon (f) Theano
1024	(g) Neon (f) Theano

Cuadro 3.3: Framework más rápido para cada tamaño de batch posible usado en GPU, caso AlexNet. Notación: (g): Gradient Computation time, (f): Forward time, (g, f): Tanto gradient computation time como forward time.

Tamaño batch	**Mejor(es) framework(s)**
32	(g, f) Torch (g) Neon
64	(g, f) Neon (f) Torch
128	(g, f) Neon (f) Torch
256	(g, f) Neon
512	(g, f) Neon

De manera complementaria, los autores destacaron a Torch por poseer extensa documentación acerca de sus características y funcionalidades. Por otro lado, también destacaron la alta flexibilidad de TensorFlow, pero al mismo tiempo criticaron su rendimiento en GPU, añadiendo que alcanza valores demasiado altos tanto de Forward Time como de Gradient Computation Time en comparación con los frameworks restantes. Además, fue quién alcanzó el nivel máximo de consumo de memoria RAM de la GPU. Sin embargo, es pertinente recordar que Caffe, Theano y Torch ofrecen soporte para trabajar con las librerías CUDA 7.5 y cuDNN v3, mientras que, hasta el momento de la publicación de este trabajo, TensorFlow no ofrecía soporte para tales versiones de las librerías mencionadas, por lo que las pruebas realizadas en conjunto con TensorFlow sólo pudieron llevarse a cabo utilizando CUDA 7.0 y cuDNN v2, por lo que el framework no fue evaluado en igualdad de condiciones.

A comienzos de mayo de 2016, Al-Rfou et al. [7], miembros del equipo de desarrollo de Theano, elaboraron un documento en el que dan cuenta de las principales caracterís-

ticas de este framework, mencionando sus principales ventajas, así como las debilidades que posee. También, aprovecharon esta instancia para realizar una comparación entre Theano, Torch y Tensorflow respecto a su rendimiento sobre diferentes modelos utilizados en la resolución de problemas de Machine Learning.

En cuanto a sus principales características, los autores destacan la posibilidad de representar expresiones matemáticas simbólicas por medio de grafos acíclicos dirigidos, donde los nodos pueden ser variables o bien la aplicación de cierta operación matemática. Cada nodo-variable está asociado un tipo de dato definido entre cuatro posibles, los que son:

- TensorType: Para arreglos n-dimensionales que se almacenan en memoria principal.
- CudaNdarrayType: para arreglos n-dimensionales que se almacenan en memoria GPU.
- GpuArrayType
- Sparse: Para matrices sparse que se almacenan en memoria principal.

El tipo de cada variable debe especificarse en el momento en que el grafo es creado. Además, se ofrece la posibilidad de clonar grafos, o parte de ellos, de tal forma que variables intermediarias en el grafo original pueden convertirse en inputs u outputs en el grafo nuevo y viceversa.

Se indica que otra funcionalidad importante ofrecida por Theano es la posibilidad de computar derivadas en forma eficiente (considerando tanto el tiempo transcurrido como el consumo de memoria necesario para llevar a cabo la operación) por medio de back propagation. Por otro lado, también ofrece funcionalidades especiales para facilitar el manejo de loops dentro del grafo. Dado a que Theano es, fundamentalmente, una interfaz de python, permite al usuario crear funciones propias que pueden ser aplicadas sobre variables presentes en el grafo. De igual forma, Theano permite que el usuario puede implementar funcionalidades en C++ o CUDA, en caso de que se lleve a cabo computación sobre GPU. Theano ofrece la posibilidad de realizar optimizaciones en el grafo construido, específicamente sobre los nodos que aplican operaciones matemáticas; entre ellas pueden encontrarse la eliminación de operaciones redundantes, eliminación de cálculos innecesarios, reemplazo de operaciones por otras más estables o por

otras más eficientes, ejecutar operaciones en GPU, cálculos in-place, uso de variables compartidas, entre otras funcionalidades, siendo posible deshabilitarlas si el usuario así lo requiere.

Por otro lado, los autores comparan las funcionalidades ofrecidas por Theano con otros framework. Se menciona su similitud con TensorFlow en relación al uso de grafos para la implementación de modelos y a la posibilidad de computar derivadas de manera eficiente, destacando que como plus ofrece la posibilidad de trabajar con computación distribuida. De igual forma, se destacan las posibilidades que ofrecen MXNet y Caffe para el cálculo de derivadas y se enfatiza en que el primero también permite computación distribuida. Además, se hace referencia al hecho de que Torch carece de métodos propios para el cómputo eficiente de derivadas, siendo necesario para ello trabajar en conjunto con paquetes como nn, cunn y Autograd. Por otro lado, destacan a Neon y Chainer, dado a que las operaciones que ofrecen están optimizadas para su uso en GPU, destacando el eficiente cómputo de derivadas y el buen rendimiento en la implementación de redes neuronales del primero, y el fácil manejo de loops y de estructuras condicionales del segundo.

A la fecha de publicación de este artículo, se realizaron importantes mejoras en el framework: Se facilitó el acceso a algoritmos convolucionales provistos por diferentes librerías. Se amplió la gama de GPU's soportadas, mejorando además las posibilidades de portabilidad entre estas. También, se aumentó la cantidad de operaciones que pueden ser ejecutadas en GPU. Por medio de la integración con la librería CNMeM, se volvió factible manejar de forma eficiente el uso de memoria de GPU, a través de técnicas asincrónicas de asignación y liberación de dicho recurso. Se optimizó el manejo de loops, logrando que su uso sea más estable, y disminuyendo los tiempos de compilación y ejecución de los grafos que incluyen tal estructura. Se añadió un nuevo backend para GPU, libgpuarray, el cuál ofrece soporte para una mayor variedad de tipos de datos y para el uso de múltiples GPU's, permitiendo así el uso de la técnica model parallelism, la cual divide el modelo en partes y las distribuye en los distintos dispositivos de procesamiento disponibles. Se posibilitó la integración con el paquete Platoon, permitiendo trabajar con data parallelism, técnica que divide los datos de entrada en múltiples partes y luego usa cada una de ellas en diferentes copias del modelo que están siendo entrenadas. Se habilitó el almacenamiento y la recuperación de modelos ya entrenados, evitando así que sea necesario re-computar los parámetros de dichos modelos en caso de que el

usuario requiera volver a hacer uso de ellos.

También se añadió el modulo d3viz para la visualización interactiva de grafos, permitiendo, por ejemplo, que el usuario pueda amplificar cierta zona del grafo, pueda obtener información acerca de aquellos nodos en donde el tiempo de cómputo es mayor al colorearlos de una forma determinada o conocer el tipo de dato asociado a cada uno de estos últimos. Además, se mejoró la prevención de errores, siendo posible detectar de antemano la utilización de tipos de datos equivocados, dimensiones de arreglos incompatibles y/o valores inesperados (como NaN o infinito). De igual manera, se mejoran los mensajes de error entregados, siendo más explicativos y detallados.

Una vez terminada la revisión de las características del framework, los autores contrastan el rendimiento de Theano con el de Torch y el de TensorFlow, ejecutando pruebas sobre redes convolucionales, redes recurrentes y redes recurrentes para mapeo sequence-to-sequence. Los experimentos fueron ejecutados en un equipo con CPU Intel Core i7-5930K de 6 núcleos y 12 threads por cada núcleo y en conjunto con un cluster de GPU's denominado NVIDIA Digits DevBox, conformado por 4 GPU's NVIDIA Titan X. Notar que todas las pruebas fueron ejecutadas utilizando sólo una GPU, excepto en aquella en que se trabajó con data parallelism. No se realizaron experimientos en CPU. Se usaron las librerías CUDA 7.5.17 y cuDNN v4 como backends de todos los frameworks en cada una de las pruebas. Notar que en este escenario TensorFlow sí pudo competir con los demás frameworks en igualdad de condiciones, a diferencia de lo ocurrido en [19].

Para el caso de redes convolucionales, se estudiaron cuatro arquitecturas sobre ImageNet:

- ALexNet: Los detalles de esta red ya han sido descritos previamente. Se usaron mini-batches de tamaño 128 para el proceso de entrenamiento.
- OverFeat [23]: Conformada por once capas, siendo cinco de ellas convolucionales y tres totalmente conectadas. A continuación de la primera, segunda y quinta capa convolucional se sitúa una capa de max pooling, todas con pooling size de 2 x 2 y stride de 2. Todas las capas de la red utilizan ReLU como función de activación, excepto por la capa de salida, la cual emplea la función softmax. Se usaron mini-batches de tamaño 128 para el entrenamiento de esta red. La arquitectura está conformada por 145 millones de parámetros. La estructura de cada capa se detalla a continuación:

- La primera capa convolucional está integrada por 96 filtros de tamaño 11 x 11 y stride de 4.
- La segunda capa convolucional está compuesta por 256 filtros de tamaño 5 x 5 y stride de 1.
- La tercera capa convolucional se encuentra conformada por 512 filtros de tamaño 3 x 3 y stride de 1.
- La cuarta y la quinta capa convolucional están integradas por 1024 filtros de tamaño 3 x 3 y stride de 1, cada una.
- La sexta es una capa totalmente conectada de 3072 neuronas. Se aplica dropout de 0,5.
- La séptima capa, también totalmente conectada, posee 4096 neuronas. Se aplica dropout de 0,5.
- La capa de salida, también totalmente conectada, contiene 1000 neuronas.

- OxfordNet (modelo A) [24]: Constituida por once capas, donde las primeras ocho son de tipo convolucional y las últimas tres son capas totalmente conectadas. Cada una de ellas hace uso de ReLU como función de activación, excepto por la capa de salida, que utiliza softmax. Todas las capas convolucionales tienen un stride fijo de 1. Luego de la primera, segunda, cuarta, sexta y octava capa convolucional se sitúa una capa de max pooling, todas con pooling size de 2 x 2 y stride de 2. Se reporta que este modelo posee un total de 133 millones de parámetros entrenables. Se usaron mini-batches de tamaño 64 para su entrenamiento. A continuación se listan las características de cada capa:
  - La primera capa convolucional está conformada por 64 filtros de tamaño 3 x 3.
  - La segunda capa convolucional está integrada por 128 filtros de tamaño 3 x 3.
  - La tercera y la cuarta capa convolucional están constituidas por 256 filtros de 3 x 3.
  - Entre la quinta y la octava capa convolucional existen 512 filtros de 3 x 3, por cada una.

- La novena y la décima capa, totalmente conectadas, contienen 4096 neuronas, cada una.
- La capa de salida, totalmente conectada, posee 1000 neuronas.

- GoogLeNet V1 [25]: Conformada por dos capas convolucionales, cuatro capas de max pooling, una capa de average pooling, dos capas totalmente conectadas y nueve módulos inception[4], de tal manera que existen 22 capas con parámetros entrenables en el modelo. Se reporta la existencia de un total de 6.797.700 parámetros entrenables. Para toda capa convolucional (incluyendo aquellas contenidas en los módulos inception) se utilizó ReLU como función de activación. Se usaron mini-batches de tamaño 128 para el entrenamiento del modelo. A continuación, se detalla la estructura de cada una de las capas, siendo estas listadas en el mismo orden de aparición dentro del modelo.
  - Capa convolucional de 64 filtros de tamaños 7 x 7 y stride de 2.
  - Capa de max pooling con pooling size de 3 x 3 y stride de 2.
  - Capa convolucional de 64 filtros de tamaños 1 x 1 y stride de 1.
  - Capa convolucional de 192 filtros de 3 x 3 y stride de 1.
  - Capa de max pooling con pooling size de 3 x 3 stride de 2.
  - Modulo inception constituido por:
    - Capa convolucional de 64 filtros de 1 x 1 y stride 1.
    - Capa convolucional de 96 filtros de 1 x 1 y stride 1.
    - Capa convolucional de 128 filtros de 3 x 3 y stride 1.
    - Capa convolucional de 16 filtros de 1 x 1 y stride 1.
    - Capa convolucional de 32 filtros de 5 x 5 y stride 1.
    - Capa de max pooling con pooling size de 3 x 3 y stride de 1.
    - Capa convolucional de 64 filtros de 1 x 1 y stride 1.
  - Modulo inception integrado por:

[4]Ver anexo A para más detalle sobre la estructura y el propósito de un módulo inception

    - Capa convolucional de 128 filtros de 1 x 1 y stride 1.
    - Capa convolucional de 128 filtros de 1 x 1 y stride 1.
    - Capa convolucional de 192 filtros de 3 x 3 y stride 1.
    - Capa convolucional de 32 filtros de 1 x 1 y stride 1.
    - Capa convolucional de 96 filtros de 5 x 5 y stride 1.
    - Capa de max pooling con pooling size de 3 x 3 y stride de 1.
    - Capa convolucional de 64 filtros de 1 x 1 y stride 1.
- Capa de max pooling con pooling size de 3 x 3 y stride de 2.
- Modulo inception conformado por:
    - Capa convolucional de 192 filtros de 1 x 1 y stride 1.
    - Capa convolucional de 96 filtros de 1 x 1 y stride 1.
    - Capa convolucional de 208 filtros de 3 x 3 y stride 1.
    - Capa convolucional de 16 filtros de 1 x 1 y stride 1.
    - Capa convolucional de 48 filtros de 5 x 5 y stride 1.
    - Capa de max pooling con pooling size de 3 x 3 y stride de 1.
    - Capa convolucional de 64 filtros de 1 x 1 y stride 1.
- Modulo inception conformado por:
    - Capa convolucional de 160 filtros de 1 x 1 y stride 1.
    - Capa convolucional de 112 filtros de 1 x 1 y stride 1.
    - Capa convolucional de 224 filtros de 3 x 3 y stride 1.
    - Capa convolucional de 24 filtros de 1 x 1 y stride 1.
    - Capa convolucional de 64 filtros de 5 x 5 y stride 1.
    - Capa de max pooling con pooling size de 3 x 3 y stride de 1.
    - Capa convolucional de 64 filtros de 1 x 1 y stride 1.
- Modulo inception integrado por:

    - Capa convolucional de 128 filtros de 1 x 1 y stride 1.
    - Capa convolucional de 128 filtros de 1 x 1 y stride 1.
    - Capa convolucional de 256 filtros de 3 x 3 y stride 1.
    - Capa convolucional de 24 filtros de 1 x 1 y stride 1.
    - Capa convolucional de 64 filtros de 5 x 5 y stride 1.
    - Capa de max pooling con pooling size de 3 x 3 y stride de 1.
    - Capa convolucional de 64 filtros de 1 x 1 y stride 1.
- Modulo inception constituido por:
    - Capa convolucional de 112 filtros de 1 x 1 y stride 1.
    - Capa convolucional de 144 filtros de 1 x 1 y stride 1.
    - Capa convolucional de 288 filtros de 3 x 3 y stride 1.
    - Capa convolucional de 32 filtros de 1 x 1 y stride 1.
    - Capa convolucional de 64 filtros de 5 x 5 y stride 1.
    - Capa de max pooling con pooling size de 3 x 3 y stride de 1.
    - Capa convolucional de 64 filtros de 1 x 1 y stride 1.
- Modulo inception conformado por:
    - Capa convolucional de 256 filtros de 1 x 1 y stride 1.
    - Capa convolucional de 160 filtros de 1 x 1 y stride 1.
    - Capa convolucional de 320 filtros de 3 x 3 y stride 1.
    - Capa convolucional de 32 filtros de 1 x 1 y stride 1.
    - Capa convolucional de 128 filtros de 5 x 5 y stride 1.
    - Capa de max pooling con pooling size de 3 x 3 y stride de 1.
    - Capa convolucional de 128 filtros de 1 x 1 y stride 1.
- Capa de max pooling con pooling size de 3 x 3 y stride de 2.
- Modulo inception integrado por:

    - Capa convolucional de 256 filtros de 1 x 1 y stride 1.
    - Capa convolucional de 160 filtros de 1 x 1 y stride 1.
    - Capa convolucional de 320 filtros de 3 x 3 y stride 1.
    - Capa convolucional de 32 filtros de 1 x 1 y stride 1.
    - Capa convolucional de 128 filtros de 5 x 5 y stride 1.
    - Capa de max pooling con pooling size de 3 x 3 y stride de 1.
    - Capa convolucional de 128 filtros de 1 x 1 y stride 1.
- Modulo inception constituido por:
    - Capa convolucional de 384 filtros de 1 x 1 y stride 1.
    - Capa convolucional de 192 filtros de 1 x 1 y stride 1.
    - Capa convolucional de 384 filtros de 3 x 3 y stride 1.
    - Capa convolucional de 48 filtros de 1 x 1 y stride 1.
    - Capa convolucional de 128 filtros de 5 x 5 y stride 1.
    - Capa de max pooling con pooling size de 3 x 3 y stride de 1.
    - Capa convolucional de 128 filtros de 1 x 1 y stride 1.
    - Capa de average pooling con pooling size de 7 x 7 y stride de 1.
- Capa con dropout de 0,4.
- Capa totalmente conectada y lineal de 1000 neuronas.
- Capa de salida, totalmente conectada, de 1000 neuronas.

Para el proceso de entrenamiento de cada uno de los modelos anteriores, se consideraron las métricas forward computation time y backward time promedio por mini-batch. Los resultados obtenidos indicaron que ambos tiempos son superiores para Theano respecto a Torch y TensorFlow, esto es, Theano resultó ser siempre más lento que los frameworks restantes. Aún así, los autores consideraron que las diferencias resultaron ser muy pequeñas entre los tiempos medidos para cada uno, lo que quiere decir que los rendimientos de los frameworks fueron de todas maneras comparables. En la tabla

3.4 se muestran los frameworks que consiguieron los mejores resultados en el entrenamiento de cada una las arquitecturas convolucionales estudiadas, adjuntando la mayor cantidad de información posible asociada a cada configuración.

Cuadro 3.4: Frameworks que consiguen los mejores resultados en el entrenamiento de cada una de las arquitecturas convolucionales estudiadas en [7] en base al tiempo promedio de procesamiento por mini-batch. No se informaron tiempos específicos. Dataset utilizado: ImageNet. Notación: (b): Backward time, (f): Forward computation time, (g, f): Tanto gradient computation time como forward time.

Arquitectura	**Mejor(es) framework(s)**	**Tamaño de mini-batch**
AlexNet	(b, f) Torch (f) TensorFlow	128
OverFeat	(b) Torch (f) TensorFlow	128
OxfordNet	(b) Torch (f) TensorFlow	64
GoogLeNet	(b, f) TensorFlow (f) Torch	128

Para el caso de redes neuronales recurrentes, se consideraron tres variantes de la arquitectura LSTM estándar, las que fueron aplicadas sobre el dataset Penn Treebank para el etiquetado de palabras por medio de POS tagging (pronombres, sustantivos, verbos, entre otros). Este dataset está compuesto por alrededor de 2.500 textos en inglés, cada uno con una cantidad variable de oraciones, almacenando así un total de cerca de un millón de palabras y un vocabulario de cardinalidad 49.000 [26]. Cómo métrica de evaluación en el entrenamiento, se consideró la cantidad de palabras procesadas por segundo (en miles de palabras), de tal manera que mientras más palabras por segundo procesa un framework, mejor es su rendimiento. Las variantes LSTM usadas fueron:

- Modelo pequeño: Capa única, 200 unidades ocultas. Recibe palabras de largo 20 como input. Posee 184.036 parámetros entrenables.
- Modelo mediano: Capa única, 600 unidades ocultas. Recibe palabras de largo 40 como input. Posee 1.560.036 parámetros entrenables.
- Modelo grande: Dos capas, 650 unidades ocultas cada una. Recibe palabras de largo 50 como input. Posee 5.228.636 parámetros.

Todos los modelos fueron regularizados con dropout en las conexiones no recurrentes. Se utilizaron mini-batches de tamaño 20 en el entrenamiento de cada uno de estos modelos.

Los resultados mostraron que TensorFlow fue quién mejor rindió en el modelo pequeño, seguido por Theano, siendo este último un poco más rápido que el primero en los modelos mediano y grande. Torch obtuvo el peor rendimiento en los tres modelos. En la tabla 3.5 se resumen los resultados obtenidos.

Cuadro 3.5: Frameworks que consiguen los mejores resultados en cada una de las redes LSTM estudiadas en [7] en base a la cantidad de palabras procesadas por segundo (en miles de palabras). Se utilizaron mini-batches de tamaño 20. No se informó la cantidad especifica de palabras procesadas por cada framework. Dataset utilizado: Penn TreeBank.

Modelo	**Número de capas**	**Número de celdas**	**Mejor(es) framework(s)**
Pequeño	1	200	TensorFlow
Mediano	1	600	Theano
Grande	2	650 (c/u)	Theano

En el caso de redes recurrentes para mapeo sequence-to-sequence, se consideró el modelo encoder-decoder descrito en [27]. Dicho modelo recibe como input frames de vídeos, generando como output una descripción en inglés. Cada frame es pre-procesado por GoogLeNet V1 (encoder) y convertido en un vector de 1024 atributos. Luego, cada secuencia de vectores es recibida por una red recurrente conformada por una capa LSTM (decoder; no se indica el número de unidades presentes en dicha capa ni la canti-

dad total de parámetros de la arquitectura). Nuevamente, se consideraron cómo métricas forward computation time y backward time. Además se consideraron tres tamaños de mini-batches para la fase de entrenamiento: 32, 64 y 128. No fue posible medir el rendimiento de Torch, dada la imposibilidad de implementar la arquitectura estudiada, por lo que sólo fue posible comparar a Theano con TensorFlow. Luego, Theano logró obtener un forward computation time levemente menor al de TensorFlow, pero los papeles se invirtieron respecto a backward time, siendo los tiempos totales muy parecidos. Se observó que Theano fue más rápido para mini-batches de tamaño pequeño, mientras que TensorFlow lo fue para batches de gran tamaño. Por último, se menciona que los beneficios de usar computación paralela aumentan a medida que se trabaja con mini-batches de tamaño cada vez más grande. En la tabla 3.6 se resumen los resultados obtenidos en éste experimento.

Cuadro 3.6: Frameworks que consiguen los mejores resultados para cada tamaño posible de mini-batch utilizado en los experimentos realizados con la red recurrente para mapeo sequence-to-sequence en [7], de acuerdo al tiempo promedio de procesamiento por mini-batch. No se informaron tiempos específicos. Dataset utilizado: Frames de vídeos. Notación: (b): Backward time, (f): Forward computation time, (g, f): Tanto gradient computation time como forward time.

Tamaño batch	**Mejor(es) framework(s)**
32	(b, f) Theano
64	(b, f) Theano (b) TensorFlow
128	(b) TensorFlow (f) Theano

En el último caso de la sesión de pruebas, se estudió el uso de data parallelism por medio de los mismos tres modelos LSTM evaluados previamente (pequeño, mediano y grande) y variando la cantidad de GPU's usadas, gracias al paquete Platoon. Los autores experimentaron con dos y cuatro GPU's. Junto con variar la cantidad de GPU's, tam-

bién se compararon los desempeños de las redes al sincronizar los resultados luego del procesamiento de cada mini-batch versus sincronizar cada cien mini-batches. Al igual que antes, se consideró la métrica de palabras procesadas por segundo. Notar que sólo se ejecutaron pruebas en Theano. Los resultados mostraron que existe una relación directamente proporcional entre la cantidad de GPU's y el número de palabras procesadas por segundo en cada uno de los tres modelos, observando una aceleración de entre 1,6 y 1,7 al usar dos GPU's y de alrededor de 3,2 para cuatro GPU's al sincronizar luego del procesamiento de cada mini-batch. Por otro lado, se observó una aceleración de 2 al usar dos GPU's y de entre 3,9 y 4 con cuatro GPU's al sincronizar luego del procesamiento de cien mini-batches, mejorando aún más el rendimiento de las redes.

Para finalizar, el equipo de desarrollo expuso las deficiencias que posee Theano. En primer lugar, se señaló que, debido a que el framework es una interfaz de Python, se ve afectado por las restricciones del lenguaje, siendo la ejecución concurrente de múltiples threads en CPU una las limitaciones más relevantes existente. Por otro lado, el tiempo requerido para optimizar un grafo comienza a crecer significativamente a medida que el tamaño de dicho grafo aumenta. Más aún, de acuerdo a los mismos autores, el tiempo de optimización varía supra linealmente respecto a la cantidad de nodos que posee el grafo. A futuro, se espera poder mejorar los tiempos de compilación del código generado internamente a partir del grafo implementado. Si bien Theano ofrece soporte para loops y estructuras condicionales, sólo considera los casos más generales de uso y por tanto carece de flexibilidad en este sentido, siendo prácticamente infactible el uso de loops recursivos, loops anidados o bien, modificar la cantidad de iteraciones preestablecida. Se critica el hecho del alto consumo de memoria que conlleva el entrenamiento de los modelos implementados, ocasionando que el poder de procesamiento que ofrece una GPU no pueda aprovecharse al máximo. Por último, se indica que para la compilación de las expresiones matemáticas definidas, el grafo debe estar construido por completo previamente, por lo que se plantea la necesidad de poder compilar en tiempo real, es decir, que inmediatamente después de que el usuario especifique una expresión, esta sea compilada. Tal procedimiento es empleado por Torch.

A fines de mayo de 2016, Abadi et al. [28], miembros del equipo de desarrollo de TensorFlow, estudiaron el rendimiento del framework en el proceso de entrenamiento de redes neuronales sobre problemas de clasificación de imágenes y modelos de lenguaje, buscando demostrar que el manejo de dichos problemas puede realizarse de forma

eficiente y escalable en TensorFlow.

Los autores destacaron las principales características que todo sistema computacional debe poseer para poder manejar problemas de machine learning de forma adecuada en TensorFlow: Debe permitir paralelizar tanto el entrenamiento como la evaluación de modelos, debe facilitar el uso de recursos que permitan acelerar estos procesos (como por ejemplo, disponer de una o más GPU's) y favorecer el rendimiento de los modelos en la fase de evaluación a medida que estos crecen y/o la cantidad de datos disponibles aumenta considerablemente. Por último, señalaron que el sistema debe ser extensible.

Luego mencionaron como, al igual que Theano, TensorFlow hace uso de grafos para representar los modelos creados por el usuario. Sin embargo, existen diferencias: En TensorFlow, los vértices del grafo representan operaciones matemáticas (denominados *operations*) y los arcos valores que circulan a través de este, ya sea como inputs o como outputs de una operación matemática. Tales valores son denominados *tensors*, los que pueden ser datos densos o sparse. Se indica que esta estructura favorece el procesamiento paralelo y distribuido de los modelos construidos. El cómputo paralelo puede ser administrado dentro del mismo framework al particionar el grafo original en sub-grafos, mientras que el cómputo distribuido posibilita que distintas tareas puedan ejecutarse en diferentes dispositivos (por ejemplo, se puede entrenar el modelo en una GPU y evaluarlo en una CPU) al asignar las operaciones realizadas en dichas tareas en forma explícita a dispositivos específicos. También, se hace referencia a la existencia de operaciones que permiten manejar con sencillez estructuras condicionales complejas, como loops anidados y procesos iterativos concurrentes, por ejemplo.

A continuación, los autores destacan las características que, a la fecha de publicación de este paper, posee TensorFlow. Por defecto, el framework permite la diferenciación automática de expresiones, llevando a que sea posible optimizar los parámetros de un modelo sin que el usuario deba diferenciar explícitamente dichas expresiones, facilitando así, por ejemplo, el entrenamiento de redes neuronales vía gradient back-propagation. Además, provee una amplia gama de métodos que permiten optimizar estos parámetros, dando la posibilidad de que los usuarios implementen sus propios métodos de optimización. Dada la alta dimensionalidad de los datos con que se acostumbra a trabajar en problemas de machine learning, TensorFlow hace uso de representaciones distribuidas de los mismos, que se caracterizan por poseer una menor dimensionalidad respecto a los datos originales, optimizando el proceso de entrenamiento del modelo en cuestión.

Con el propósito de que sea tolerante a fallas, el framework posibilita la creación y recuperación de checkpoints, de tal manera que es posible conservar los valores almacenados por variables del modelo en un instante determinado. Sin embargo, el equipo de desarrollo señala que la frecuencia de uso de estas funcionalidades debiese ser baja.

Si bien TensorFlow fue originalmente diseñado para entrenar modelos de manera asincrónica, el equipo de desarrollo ha comenzado a experimentar con métodos sincrónicos. El que el entrenamiento sea sincrónico o asincrónico determina la forma en que se sincronizan, valga la redundancia, los parámetros de un modelo cuando estos son computados de forma paralela. Más aún, los autores plantearon, a modo de hipótesis, la posibilidad de realizar esta tarea sincrónicamente en menos tiempo que por medio del método asincrónico, gracias a que actualmente es factible acelerar el proceso de entrenamiento vía ejecución en GPU. A su vez, indicaron las ventajas y desventajas de cada método: Por un lado, el uso de técnicas asincrónicas permite que sea posible actualizar una mayor cantidad de parámetros en menor tiempo respecto a métodos sincrónicos en presencia de unidades de trabajo (o en otras palabras, máquinas) más lentas que otras, pero esto lleva a que ciertos dispositivos deban trabajar con datos obsoletos. Por otra parte, el uso de métodos sincrónicos permite que las unidades de trabajo puedan trabajar con parámetros actualizados al momento de ejecutar una iteración de entrenamiento, pero se ven desfavorecidos en la presencia de unidades más lentas que otras. Para hacer frente a esta última problemática, los autores mencionan la implementación de *backup workers* en el framework, si bien no se entregan mayores detalles respecto a dicha implementación.

Se destaca que TensorFlow está implementando en el lenguaje de programación C++, buscando portabilidad y un rendimiento eficiente. El framework puede ser utilizado en diferentes sistemas operativos, cómo lo son Linux, Mac OS X, Android y iOS; también en diversas arquitecturas de CPU's y GPU's (para el último recurso, se mencionan NVIDIA Kepler, NVIDIA Maxwell y NVIDIA Pascal, permitiendo que los usuarios pueden pueden extender el funcionamiento de TensorFlow en otras arquitecturas). Se muestra, además, que el framework está organizado en capas desde el punto de vista arquitectónico y se detallan las funciones que cumple cada una de estas capas. Adicionalmente, se ofrece soporte para múltiples lenguajes en el lado del cliente, aunque sólo se mencionan a Python y C++. Además, se entregan herramientas al usuario que le permiten obtener feedback de su trabajo, como dashboards que muestran el

progreso en el entrenamiento de modelos, visualizadores gráficos que explican las conexiones existentes dentro de estos y funcionalidades para monitorear la ejecución de tareas distribuidas a través de múltiples dispositivos.

Para finalizar, se ejecutaron pruebas para evaluar el rendimiento de TensorFlow sobre problemas de clasificación de imágenes y language modeling. Parte de las pruebas fueron ejecutadas en un cluster compartido, mientras que otras se llevaron a cabo sobre una máquina individual. Así, en cada experimento se reportó la mediana de los tiempos empleados en las iteraciones de entrenamiento de cada modelo estudiado. En primer lugar, se estudió el tiempo necesario para el entrenamiento de cuatro modelos convolucionales diferentes sobre el dataset ImageNet: AlexNet, Overfeat, OxfordNet y GoogleNet (ya descritos previamente), comparando los resultados obtenidos con el rendimiento de Caffe, Neon y Torch en la ejecución de las mismas pruebas. Para este experimento, se usó una máquina con CPU Intel Core i7-5930K 3.5GHz de 6 núcleos y una GPU Nvidia Titan X; sólo se realizaron experimentos en GPU. TensorFlow consigue el mejor de todos los tiempos sobre AlexNet, compartiendo el primer lugar con Torch. Sin embargo, en los tres modelos restantes, es Neon quien consigue los mejores resultados, aunque los tiempos de TensorFlow son comparables con los de Torch, siendo los de este último ligeramente menores (según los autores existe, en promedio, una diferencia del 6 % entre los tiempos de los dos frameworks). La similitud de rendimientos es atribuida a que tanto TensorFlow como Torch ocuparon la misma versión de cuDNN como backend (no se especifica que versión). Además, se justifica el buen rendimiento de Neon con el hecho de que el framework hace uso de kernels convolucionales optimizados. Ante esto, se considera como una posibilidad futura tener en cuenta tal funcionalidad en TensorFlow. A partir de este punto en adelante, sólo se hicieron experimentos en este framework. En la tabla 3.7 se resumen los resultados obtenidos, adjuntando la mayor cantidad de información disponible.

Cuadro 3.7: Frameworks que consiguen los mejores resultados en el entrenamiento de cada uno de los modelos convolucionales estudiados en [28], en base a la mediana de los tiempos empleados en las iteraciones del proceso de entrenamiento. Dataset utilizado: ImageNet.

Arquitectura	**Mejor(es) framework(s)**	**Mediana de tiempos de iteraciones de entrenamiento [ms]**
AlexNet	Torch, TensorFlow	81
OverFeat	Neon	211
OxfordNet	Neon	320
GoogleNet	Neon	270

Para el análisis del uso de comunicación síncrona, se estudiaron cinco modelos nulos de diferentes tamaños: Un modelo denominado *scalar* (el más ”liviano”, en términos de cantidad de parámetros), dos modelos *sparse* de tamaños 1 GB y 16 GB y dos modelos densos de tamaños 100 MB y 1 GB. Para cada uno de estos modelos, se mostró que, refutando la hipótesis planteada por los autores, el entrenamiento con métodos sincrónicos es perjudicial en términos de tiempo, en el sentido de que la mediana de los tiempos de las iteraciones de entrenamiento crece al aumentar el número de unidades de trabajo. Por otro lado, la cantidad de mini-batches procesados por segundo aumenta linealmente a medida que también lo hace la cantidad de unidades paralelas que trabajan en este procedimiento, independiente del tipo de comunicación utilizado.

Continuando con problemas de clasificación de imágenes, se entrenó el modelo Inception-v3 provisto por Google sobre ImageNet, dada su alta precisión (78.8 %) en la competencia ILSVRC 2012. Este modelo corresponde a una versión modificada de la red GoogLeNet y presenta los siguientes cambios con respecto a esta última arquitectura:

- Aquellas capas convolucionales que aplicaban filtros de 5 x 5 en los módulos inception han sido reemplazadas por dos capas convolucionales consecutivas con

filtros de 3 x 3 y un máximo de 128 de estos. Además, se aplica BN posterior a las convoluciones (modificaciones introducidas en Inception-v1 [29]).

- Se factorizaron las convoluciones de 7 x 7 en tres convoluciones de 3 x 3 (modificación introducida en Inception-v2 [30]).
- Se integró la operación BN auxiliary (modificación introducida en Inception-v3 [30]).

El número total de parámetros entrenables de esta arquitectura no fue especificado.

Al comparar el uso de computación sincrónica con computación asincrónica en el entrenamiento del modelo, se tiene que la cantidad de imágenes procesadas por segundo aumenta a medida que también lo hace el número de unidades de trabajo paralelas, para cada caso, aunque dicho aumento es más rápido para el cómputo asincrónico. Sin embargo, aún cuando el total de imágenes procesadas por segundo aumenta, se tiene que la contribución marginal de cada unidad de trabajo nueva que se agrega comienza a ser cada más insignificante respecto al total, lo que se manifiesta en un aumento de la mediana de los tiempos de las iteraciones empleadas en el entrenamiento del modelo, para cada uno de los dispositivos. Esta idea es válida tanto en el contexto de computación sincrónica cómo asincrónica. Notar que cada unidad trabajó con un cluster de GPU's por separado. Así, el paper no consiguió demostrar su hipótesis respecto a la posibilidad de tener un rendimiento más alto en el entrenamiento del modelo por medio de computación sincrónica a pesar del uso de GPU's.

Para el problema de modelos de lenguaje, el principal objetivo propuesto fue: A partir de una secuencia de palabras, predecir la palabra que viene a continuación. Luego, se procedió a entrenar una red neuronal de dos capas LSTM con 512 celdas, cada una, sobre un dataset integrado por alrededor de un billón de palabras. Así, el modelo está integrado por 0,82 billones de parámetros. Dada la alta dimensionalidad de los datos, fue necesario reducir la cantidad de atributos con que se trabajó. Cada palabra perteneciente al vocabulario del modelo representa una clase del mismo. Así, se definieron 40.000 clases diferentes. Al entrenar el modelo, se observó que la cantidad de palabras procesadas por segundo aumentó con el número de unidades de trabajo que se utilizaron, dada la posibilidad de realizar computación paralela. De igual forma, se observó que al disminuir la dimensionalidad de los datos, la cantidad de palabras procesadas por segundo aumentó aún más, a pesar de que el número de unidades de trabajo se mantuvo

constante en cada experimento. Este ocurre debido a que, al tener menor dimensionalidad, es necesario trasladar menos datos a cada una de las unidades de trabajo.

En septiembre de 2016, Shi et al. [31] compararon cuatro frameworks diseñados para el trabajo con problemas de deep learning: Caffe, CNTK, TensorFlow y Torch. Los autores resaltaron las principales características y/o rendimientos más destacados alcanzados por dichos frameworks hasta la fecha de publicación del artículo:

- Caffe: Se reporta que esta herramienta es capaz de procesar 40 millones de imágenes al día al trabajar con una GPU NVIDIA K40 o una GPU Titan. Más aún, al usar cuDNN como backend, el procesamiento se acelera en un factor de 1,3 en la primera GPU mencionada.
- CNTK: Experimentos demostraron que al trabajar en un ambiente de múltiples GPU's, alcanza el mejor rendimiento sobre una red feedforward de cuatro capas ocultas. Más aún, resulta ser 1,5 veces más rápido que Caffe, TensorFlow, Theano y Torch. Como desventaja, se menciona la incapacidad de soportar la operación de concatenación en ciertas redes convolucionales, como GoogLeNet, por ejemplo.
- Tensorflow: Se destaca su flexibilidad, portabilidad y eficiencia en general.
- Torch: Se destaca el ofrecimiento de estructuras de datos y operaciones útiles para trabajar con problemas de machine learning.

A su vez, el paper se enfoca en estudiar el rendimiento sobre tres tipos de arquitecturas: Redes feedforward, redes convolucionales y redes recurrentes. Para cada tipo, se consideró una red pequeña y una grande, con el fin de realizar comparaciones entre ellas. Para las redes feedforward, se consideró una de cinco capas (FCN-5) y otra de ocho capas (FCN-8); si bien no se entregan detalles respecto a la cantidad de neuronas por capa ni las funciones de activación usadas, se indica que las arquitecturas están integradas por 55 y 58 millones de parámetros, respectivamente.

En el caso convolucional, se trabajó con dos modelos, AlexNet (ya descrito previamente) y ResNet-50 [32], cuya estructura se especifica en la figura 3.1. Notar que todas las capas convolucionales utilizan stride 1, excepto por la primera, en que se usa stride 2. Por otra parte, las capas de pooling consideran un pooling size de 3 x 3 y de 7 x 7 respectivamente; el stride es de 2 en ambas.

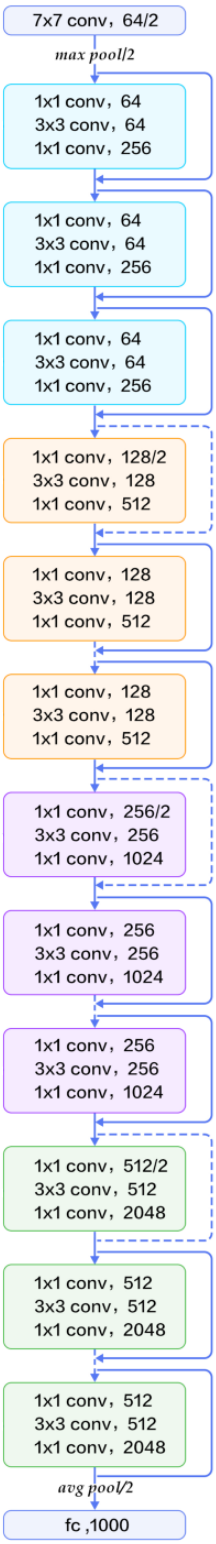


Figura 3.1: Estructura de ResNet-50. Imagen adaptada de: https://medium.com/@siddharthdas_32104/cnns-architectures-lenet-alexnet-vgg-googlenet-resnet-and-more-666091488df5

El modelo está conformado por 25,6 millones de parámetros. Además, todas las capas convolucionales hacen uso de ReLU como función de activación. Se aplica BN posterior a cada convolución.

En el caso recurrente, se consideraron dos redes LSTM, LSTM-32 y LSTM-64, de dos capas, cada una, y que reciben como inputs secuencias de largo 32 y 64, respectivamente. Ambas arquitecturas están conformadas por 13 millones de parámetros.

Desafortunadamente, no se mencionan los datasets que se usaron para realizar los experimentos (aparentemente, se utilizó ImageNet en el caso convolucional).

Respecto a hardware, se usaron dos tipos de CPU multinúcleo: una Intel i7-3820 3.60 GHz de cuatro núcleos, en conjunto con 64 GB de memoria RAM, y dos servidores equipados con una CPU Intel Xeon E5-2630 v3 2.40 GHz de ocho núcleos, cada uno, y un total de 128 GB de memoria RAM, de tal manera que cada framework fue testeado con un número variable de threads en cada caso: 1, 2, 4 y 8 threads en el caso de la CPU Intel i7 y 1, 2, 4, 8, 16 y 32 threads en el caso de los servidores. Por otro lado, también se trabajó con tres tipos diferentes de GPU's: Una NVIDIA GTX 980 con arquitectura Maxwell y 4 GB de memoria, una TESLA K80 con arquitectura Kepler y 12 GB de memoria y una GTX 1080 con arquitectura Pascal y 8 GB de memoria; las dos primeras utilizaron CUDA 7.5, mientras que la tercera usó CUDA 8.0. Notar que para el caso CPU, todos los frameworks trabajaron en conjunto con la librería OpenBLASv0.2.18, excepto por TensorFlow, que utilizó Eigen 3.2.8. Para el caso GPU, en cambio, todos los frameworks trabajaron en conjunto con la librería cuDNN v4.

Notar que se utilizaron mini-batches de tamaño 64 para las redes feedforward, de tamaño 16 para las redes convolucionales y de tamaño 128 para las redes recurrentes en las pruebas realizadas sobre CPU, mientras que para el caso de utilización de GPU, se experimentó con batches de tamaños 32, 64, 128, 256, 512 y 1024 en el caso de las redes feedforward, de tamaños 16, 32, 64 y 128 en el caso de las redes convolucionales y de tamaños 64, 128, 256 y 512 en el caso de las redes recurrentes. Como métrica de evaluación, se consideró el tiempo promedio de procesamiento de cada mini-batch.

Considerar que no fue posible evaluar las redes recurrentes en Caffe, dada la imposibilidad de implementarlas en este framework. La evaluación de ResNet-50 en CNTK fue llevada a cabo únicamente en GPU, debido a la incapacidad de hacer BN en CPU. Además, se descartó el estudio de redes convolucionales en Tensorflow junto con la

GPU GTX 1080, dado a problemas de soporte ofrecidos por cuDNN v4.

En cuánto a las pruebas ejecutadas sobre CPU's, Torch fue quien obtuvo los mejores resultados en el entrenamiento de redes feedforward, presentando su mejor performance al utilizar cuatro threads, superando incluso a los demás frameworks en los casos en que estos hicieron uso de una cantidad superior. Caffe y CNTK presentaron resultados levemente peores respecto a Torch, mientras que TensorFlow consiguió el peor rendimiento, siendo comparable con los demás sólo cuando utilizó los servidores con 32 threads. Por otro lado, para el entrenamiento del modelo AlexNet, Caffe obtuvo el mejor rendimiento por medio de la CPU de 4 núcleos con 4 threads, mientras que TensorFlow alcanzó su mejor performance a través de los servidores y con 16 threads. Para el modelo RestNet-50, Torch presentó los mejores resultados en los dos tipos de CPU usadas. Para redes recurrentes, CNTK resultó ser aproximadamente dos veces más rápido que Torch, aunque TensorFlow logró superar su rendimiento cuando usó los servidores con 32 threads. En general, se observó que, al aumentar el número de threads, se redujo la contribución marginal de cada uno en términos de la reducción del tiempo de entrenamiento total de cada modelo.

En cuanto a las pruebas ejecutadas sobre GPU's, Caffe y CNTK consiguen resultados similares y mejores en comparación con TensorFlow y Torch sobre las redes feedforward; más aún, se señala que TensorFlow tiene un pobre rendimiento sobre este tipo de red, siendo alrededor de dos veces más lento que Caffe y CNTK. Sobre los modelos convolucionales, Caffe consigue los mejores resultados sobre las GPU's GTX 980 y K80 para AlexNet independiente del tamaño de mini-batch considerado, mientras que CNTK, TensorFlow y Torch alcanzan resultados comparables entre sí sobre el modelo ResNet-50, e incluso se señala que CNTK consigue resultados levemente mejores que Caffe por medio de las GPU's GTX 980 y K80; además, se comenta que TensorFlow y Torch logran gestionar de mejor forma el uso de memoria de GPU, a diferencia de los frameworks restantes, que debieron enfrentar errores de ejecución al trabajar con inputs demasiado grandes. Sobre los modelos recurrentes, CNTK obtuvo los mejores resultados para cualquier tipo de configuración. En lo que respecta al uso de técnicas de paralelización en CPU y GPU, fue este último recurso el que permitió un rendimiento ampliamente superior, tal y cómo esperaban los autores.

En la tabla 3.8 se resumen los resultados obtenidos en [31], mostrando la mayor cantidad de información posible por cada configuración. Notar que los autores informaron

los tiempos específicos de cada configuración sólo para batches de tamaño 64, 16 y 128 para las redes feedforward, convolucionales y recurrentes, respectivamente.

Cuadro 3.8: Frameworks que consiguen los mejores resultados en cada una de las arquitecturas estudiadas en [31]. Sólo se informaron los tiempos específicos asociados a batches de tamaño 64, 16 y 128 para las redes feedforward, convolucionales y recurrentes, respectivamente.

Arquitectura	**Mejor(es) framework(s)**	**Recurso(s)**	**Librería(s)**	**Tiempo promedio por mini-batch [s]**
FCN-5	Caffe, CNTK	GPU NVIDIA GTX 1080	CUDA 8.0 + cuDNN v4	0,033
FCN-8	CNTK			0,037
AlexNet	Caffe			0,026
ResNet-50	Torch			0,188
LSTM-32	CNTK			0,062
LSTM-64	CNTK			0,122

Capítulo 4

Diseño de Experimentos y Resultados

4.1. Hardware utilizado

Todos los experimentos fueron desarrollados en un servidor individual con sistema operativo Ubuntu 15.10, placa madre MSI X99A GAMING 7, procesador Intel Core i7-5820K CPU @ 3.30 GHz, 31 GiB de memoria RAM y una tarjeta gráfica NVIDIA GeForce GTX TITAN X de 3072 núcleos CUDA y 12 [GB] de memoria.

4.2. Frameworks

En la tabla 4.1 se muestran las características de cada uno de los frameworks utilizados en la implementación de las redes.

Cuadro 4.1: Características de cada framework utilizado en la implementación de los experimentos.

Framework	Versión	Backend(s)
Keras	2.0.8	Theano | TensorFlow | CNTK
Theano	0.9.0	CUDA 8.0 + cuDNN v6
TensorFlow	1.3	CUDA 8.0 + cuDNN v6
CNTK	2.1	CUDA 8.0 + cuDNN v6

Todos los experimentos han sido implementados en Python 2.7.

4.3. Datasets

En esta memoria se han estudiado modelos diseñados para resolver problemas de clasificación de imágenes, análisis de sentimiento y time series forecasting. Para el primer escenario, se han considerado los siguientes datasets:

- MNIST [33]: Se trata de una colección de 70.000 imágenes de 28 x 28 x 1 píxeles correspondientes a dígitos manuscritos (es decir, números entre 0 y 9). En su versión tradicional, la colección se encuentra separada en dos subconjuntos: Uno de entrenamiento de 60.000 imágenes y otro de test de 10.000 imágenes. La tarea consiste en construir un modelo que sea capaz de identificar correctamente el dígito representado en la imagen.
- CIFAR-10 [34]: Consiste en un conjunto de 60.000 imágenes RGB de 32 x 32 x 3 píxeles que contiene 10 clases de objetos y 6000 ejemplos por clase. La versión utilizada viene separada en 50.000 ejemplos de entrenamiento y 10.000 ejemplos de prueba. El conjunto de pruebas fue obtenido seleccionando 1000 imágenes aleatorias de cada clase, mientras que los datos restantes fueron ordenados aleatoriamente. Las clases son mutuamente excluyentes y corresponden a las siguientes categorías: Gatos, perros, ranas, caballos, pájaros, ciervos, aviones, automóviles, camiones y barcos.

- Street View House Numbers (SVHN) [35]: Contiene imágenes naturales de dígitos de direcciones obtenidos desde Google Street View en formato RGB de 32 x 32 x 3 píxeles. El dataset contiene más de 600.000 imágenes de entrenamiento y 26.032 imágenes de test. Para facilitar la realización de experimentos, el dataset de entrenamiento usualmente es dividido en un conjunto pequeño de 73.257 imágenes y un conjunto "extra" de 531.131 imágenes. En esta memoria sólo se trabajará con modelos que hagan uso de la versión pequeña.
- NORB [36]: Contiene imágenes de tamaño 108 x 108 x 2 píxeles de 50 juguetes que pertenecen a seis categorías genéricas: Animales de cuatro patas, figuras humanas, aviones, camiones, autos y una sexta categoría para imágenes sin objetos. Dichos objetos fueron fotografiados por dos cámaras bajo 6 condiciones de iluminación, 9 ángulos de elevación (desde 30 a 70 grados, con aumentos de 5 grados) y 18 ángulos azimutales (desde 0 a 340 grados, con aumentos de 20 grados). El conjunto de entrenamiento está compuesto por 10 folds. Cada fold contiene 29.160 imágenes de 2 canales (6 categorías, 5 ejemplos, 6 tipos de iluminación, 9 tipos de elevación y 18 variaciones azimutales). Sin embargo, en la literatura es común utilizar sólo los dos primeros folds. Los modelos escogidos para este dataset cumplen con tal requisito, de tal manera que el conjunto de entrenamiento utilizado contiene un total de 58.320 imágenes. Por otro lado, el conjunto de pruebas está integrado por 2 folds, sumando un total de 58.320 ejemplos.

En la tabla 4.2 se resumen las características de los datasets descritos anteriormente.

Cuadro 4.2: Características principales de los datasets considerados para problemas de clasificación de imágenes.

Dataset	**Ejemplos de entrenamiento**	**Ejemplos de prueba**	**Dimensiones de las imágenes (filas x columnas x canales)**	**Clases**
MNIST	60.000	10.000	28 x 28 x 1	10
CIFAR-10	50.000	10.000	32 x 32 x 3	10
SVHN	73.257	26.032	32 x 32 x 3	10
NORB	58.320	58.320	108 x 108 x 2	6

Para el caso de análisis de sentimiento, se ha escogido el dataset *IMDB Movie Reviews* [37], integrado por reseñas de películas extraídas desde www.imdb.com. Cada una de estas reseñas tiene asociada una polaridad, ya sea positiva (+1) o negativa (-1). Luego, el problema asociado consiste en, dada una reseña, determinar su polaridad, lo que en otras palabras quiere decir que se está tratando con un problema de clasificación binaria. Tanto el conjunto de entrenamiento como el de prueba están integrados por 25.000 reseñas, cada uno. Cada reseña está definida por una secuencia de tamaño variable de palabras.

Finalmente, para el caso de time series forecasting, se ha considerado el problema *How Much Did It Rain? II* [38] propuesto por la plataforma Kaggle [1] en 2015. El objetivo de dicho problema consiste en predecir la cantidad total de lluvia que ha caído en una locación en específico al finalizar una hora determinada del día, a partir de un número variable de mediciones realizadas dentro de esa misma hora. Por ejemplo, se puede requerir la cantidad total de lluvia caída entre las 13:00 y las 14:00 hrs. a partir de las mediciones llevadas a cabo a las 13:07, 13:26, 13:38 y 13:51 hrs. Cada medición es expresada en [mm] y está definida en base a ciertos indicadores entregados por uno de los múltiples radares polarimétricos provistos por el Servicio Meteorológico Nacional

[1] https://www.kaggle.com/c/how-much-did-it-rain-ii

de Estados Unidos. Los ejemplos de entrenamiento corresponden a mediciones realizadas por radares sobre pluviómetros localizados en el medio oeste de Estados Unidos durante la temporada de crecimiento del maíz, es decir, entre abril y agosto de 2014, 20 días por cada mes, de tal manera que el conjunto de entrenamiento está integrado por 13.765.201 registros. El conjunto de prueba ha sido conformado en forma análoga al de entrenamiento, pero a partir de las mediciones realizadas entre los 10/11 días restantes de cada mes, quedando conformado por 8.022.756 registros. Sin embargo, es necesario precisar que los valores de las mediciones reales no han sido incluidos en el conjunto de prueba, por lo que el error de predicción no es calculado en los experimentos asociados a este dataset. Ambos conjuntos de datos han sido aleatorizados de tal manera que no están ordenados ni por locación ni por hora del día. Además, dichos atributos han sido eliminados en todos los registros. Cada ejemplo está compuesto por un total de 22 atributos (o indicadores) que son definidos a continuación:

- **Id**: Un número único asociado al conjunto de observaciones realizadas sobre un pluviómetro dentro de una hora. Este atributo no es considerado para predecir.
- **minutes_past**: Cantidad de minutos transcurridos entre el inicio de una hora y el instante en que se realiza la observación. Por ejemplo, si se está considerando el rango horario entre las 13:00 y las 14:00 hrs. y el valor del atributo es 25 [min], quiere decir que la medición se llevó a cabo a las 13:25 hrs.
- **radardist_km**: Distancia, medida en [km], entre el pluviómetro y el radar.
- **Ref**: Reflectividad del radar en [km].
- **Ref_5x5_10th**: Valor del percentil 10 para la reflectividad en un vecindario de 5 x 5 alrededor del pluviómetro.
- **Ref_5x5_50th**: Ídem al anterior, pero para el percentil 50.
- **Ref_5x5_90th**: Ídem al anterior, pero para el percentil 90.
- **RefComposite**: Reflectividad vertical máxima sobre el pluviómetro, medida en [dBZ].
- **RefComposite_5x5_10th**: Valor del percentil 10 para la reflectividad vertical máxima en un vecindario de 5 x 5 alrededor del pluviómetro.
- **RefComposite_5x5_50th**: Ídem al anterior, pero para el percentil 50.

- **RefComposite_5x5_90th**: Ídem al anterior, pero para el percentil 90.
- **RhoHV**: Coeficiente de correlación, sin unidad de medida.
- **RhoHV_5x5_10th**: Valor del percentil 10 para el coeficiente de correlación en un vecindario de 5 x 5 alrededor del pluviómetro.
- **RhoHV_5x5_50th**: Ídem al anterior, pero para el percentil 50.
- **RhoHV_5x5_90th**: Ídem al anterior, pero para el percentil 90.
- **Zdr**: Reflectividad diferencial, medida en [dB].
- **Zdr_5x5_10th**: Valor del percentil 10 para la reflectividad diferencial en un vecindario de 5 x 5 alrededor del pluviómetro.
- **Zdr_5x5_50th**: Ídem al anterior, pero para el percentil 50.
- **Zdr_5x5_90th**: Ídem al anterior, pero para el percentil 90.
- **Kdp**: Fase diferencial específica, medida en $[\frac{deg}{km}]$.
- **Kdp_5x5_10th**: Valor del percentil 10 para la fase diferencial específica en un vecindario de 5 x 5 alrededor del pluviómetro.
- **Kdp_5x5_50th**: Ídem al anterior, pero para el percentil 50.
- **Kdp_5x5_90th**: Ídem al anterior, pero para el percentil 90.

Notar que un registro puede sufrir de la ausencia de uno o más valores de atributos. La forma de abordar esta situación depende de cada usuario.

4.4. Métricas de evaluación

Se han considerado las siguientes métricas para evaluar el rendimiento de cada framework:

- Tiempo total de entrenamiento en GPU, medido en [s].
- Speed-up relativo, esto es, la razón entre los tiempos totales de entrenamiento de cada par de frameworks.
- Consumo de memoria GPU en entrenamiento, medido en [MiB].

- Uso relativo de memoria, esto es, la razón entre los consumos de memoria GPU de cada par de frameworks.
- Error de entrenamiento, entendido como la fracción de ejemplos que son mal clasificados sobre el total de ejemplos del conjunto de entrenamiento (sólo en problemas de clasificación).
- Error de prueba, análogo al error de entrenamiento (sólo en problemas de clasificación).

4.5. Experimentos preliminares

Con el fin de analizar la sensibilidad de cada framework en términos temporales con respecto a la repetición de un experimento determinado, se ejecutaron cinco instancias de entrenamiento tanto de la red feedforward FNN-6 (estudiada con mayor detalle en las secciones 4.6.1.1.1 y 4.6.1.1.2; dataset: CIFAR-10) como de la red convolucional DNN-W20 (estudiada con mayor detalle en la sección 4.6.2.1.1; dataset: MNIST), buscando así determinar el tiempo promedio junto con la varianza en cada caso. Los resultados obtenidos pueden ser observados en la tabla 4.3.

Cuadro 4.3: Tiempo de entrenamiento promedio de las redes FNN-6 y DNN-W20. Datasets: CIFAR-10 y MNIST, respectivamente. En cada caso, el experimento fue repetido cinco veces.

Red	**Tiempo promedio Theano [s]**	**Tiempo promedio TensorFlow [s]**	**Tiempo promedio CNTK [s]**
FNN-6	99,96 ± 0,38	389,06 ± 2,78	232,62 ± 0,98
DNN-W20	2041,43 ± 6,28	1351,48 ± 30,41	967,89 ± 0,12

De acuerdo a dicha tabla, las varianzas no resultan ser significativas, esto es, por más repeticiones que se hagan de un mismo experimento, las relaciones de orden entre frameworks no presentarán mayores cambios. En base a esta evidencia, se ha optado

por concentrar el esfuerzo en hacer un estudio en amplitud, implementando la mayor variedad posible de arquitecturas y datasets, siendo cada experimento ejecutado sólo una vez, descartando así la repetición de muchos experimentos sobre modelos o problemas específicos.

4.6. Modelos implementados y resultados

En esta sección, se especifican los modelos que han sido implementados para cada dataset, junto con los hiper parámetros de entrenamiento y las métricas pertinentes a cada uno. El código asociado a los experimentos puede ser encontrado en https://github.com/Seba93/ExperimentosMemoria.

4.6.1. Redes feedforward

Para el análisis de redes feedforward, se han considerado los datasets MNIST, CIFAR-10 y SVHN. Para que tanto las imágenes de entrenamiento como de prueba puedan ser procesadas adecuadamente por esta clase de red, es necesario transformarlas en vectores. Con esto en mente y con el objetivo de trabajar sobre representaciones de buenos atributos y de baja dimensionalidad, cada dataset es pre-procesado con un extractor de características diferente. Los modelos presentados en las siguientes subsecciones son de elaboración propia, pero, al menos para el caso de CIFAR-10, no se presentan diferencias de precisión de clasificación de prueba mayores al 5 % con respecto al modelo no convolucional state-of-the-art, que es de un 78,62 % [39].

4.6.1.1. CIFAR-10

Se extraen los descriptores GIST [40] de las imágenes, quedando cada una de ellas definida como un vector de 960 atributos. En términos generales, los descriptores GIST pueden ser entendidos como una aproximación de la distribución de orientaciones y escalas dentro de una imagen RGB a partir de ciertas propiedades espaciales en las que se centra el algoritmo. Dichas propiedades son:

- Naturalidad: Permite distinguir entre estructuras cuyos contornos corresponden principalmente a trazos bien definidos (como líneas rectas o curvas) y estructuras cuyos contornos siguen un patrón no convencional. Por ejemplo, la imagen de un perro, de un caballo o de un gato posee un alto grado de naturalidad, mientras que la de un camión, de un barco o de un avión posee un bajo grado de naturalidad.
- Apertura: Facilita diferenciar entre imágenes que contienen una baja cantidad de objetos ”distractores” (mayor apertura) e imágenes que poseen una alto número de elementos ajenos al objeto que se desea clasificar (menor apertura). Por ejemplo, probablemente la imagen de un avión tendrá un mayor grado de apertura en comparación a la de un automóvil.
- Aspereza: Establece una separación entre imágenes donde la superficie del objeto en cuestión posee texturas en su mayoría irregulares (alta aspereza) e imágenes en que las texturas son más bien lisas. Por ejemplo, la imagen de un pájaro tendrá un mayor nivel de aspereza respecto a la de un barco.
- Expansión: Favorece la distinción entre imágenes donde el entorno que rodea al objeto es más bien amplio e imágenes capturadas en torno a escenarios pequeños. Luego, es más probable que la imagen de un caballo o de un pájaro tenga un nivel de expansión más alto que la imagen de un perro o de un gato.
- Robustez: Permite diferenciar entre imágenes en donde los márgenes del escenario en que se sitúa el objeto pueden ser determinados con facilidad (baja robustez) e imágenes en que no lo es (alta robustez). Por ejemplo, es altamente probable que la imagen de un avión o de un barco tenga asociado un alto nivel de robustez y que la de un automóvil o de un camión tenga un nivel bajo de robustez.

  Matemáticamente, la obtención de los descriptores está dada por la aplicación de una cierta cantidad n de filtros de Gabor sobre una imagen, generando n feature maps como salida, cada uno de igual tamaño que la imagen original. A continuación, cada feature map es divido en una malla de 4 x 4, esto es, en 16 regiones distintas dentro de él. Los valores agrupados en cada región son promediados, de tal manera que los promedios resultantes son concatenados con los de los demás feature maps, generando así el vector final que representa a la imagen. Notar que este proceso es efectuado de manera independiente por cada uno de los tres canales de la misma. Así, el vector resultante queda definido por $n \cdot 16 \cdot 3$ atributos.

En esta memoria, se trabajó con $n = 20$.

4.6.1.1.1. Eficiencia en base a número de capas ocultas

En esta sección, se estudia el impacto de variar el número de capas ocultas entre 1 y 6. Para toda arquitectura, la cantidad de neuronas en la capa oculta inicial, H_0, está dada por:

$$H_0 = \frac{N \cdot 2^{L-1}}{2^L - 1}\,, \tag{4.1}$$

donde N corresponde a la dimensionalidad del patrón de entrada de la red (en este caso, N = 960) y L a la cantidad de capas ocultas de la misma. Luego, en caso de que aplique, la cantidad de neuronas en la capa oculta H_i está dada por:

$$H_i = \frac{H_{i-1}}{2}\,. \tag{4.2}$$

Así, las características de cada una de las arquitecturas estudiadas se detallan en la tabla 4.4.

Cuadro 4.4: Características de cada una de las redes feedforward entrenadas sobre los descriptores GIST de CIFAR-10. Una arquitectura denotada por FNN-i representa a una red feedforward integrada por i capas ocultas.

Nombre arquitectura	Número de neuronas por capa	Número de parámetros
FNN-1	960	932.170
FNN-2	640, 320	823.370
FNN-3	549, 274, 137	717.344
FNN-4	512, 256, 128, 64	665.162
FNN-5	495, 248, 124, 62, 31	639.602
FNN-6	487, 243, 122, 61, 31, 16	626.466

Al término de toda red FNN-i, existe una capa totalmente conectada de 10 neuronas (capa de salida).

Los pesos de todas las capas son inicializados mediante una distribución uniforme de Glorot. Como función de activación, se utiliza ReLU en las capas ocultas y softmax en la capa de salida. El entrenamiento es llevado a cabo con SGD como método de optimización, con tamaño de mini-batch 32, tasa de aprendizaje fija de 0,01, durante 50 epochs. No hay uso de momentum ni de weight decay.

La figura 4.1 muestra que, para toda profundidad posible de la red, Theano emplea los tiempos más bajos, seguido por CNTK y TensorFlow. Además, la variación de tiempo entre dos profundidades consecutivas es más bien pequeña en Theano. En el caso de CNTK, dichos aumentos son más significativos que en el framework anterior. Sin embargo, los aumentos de tiempos son más bruscos en TensorFlow.

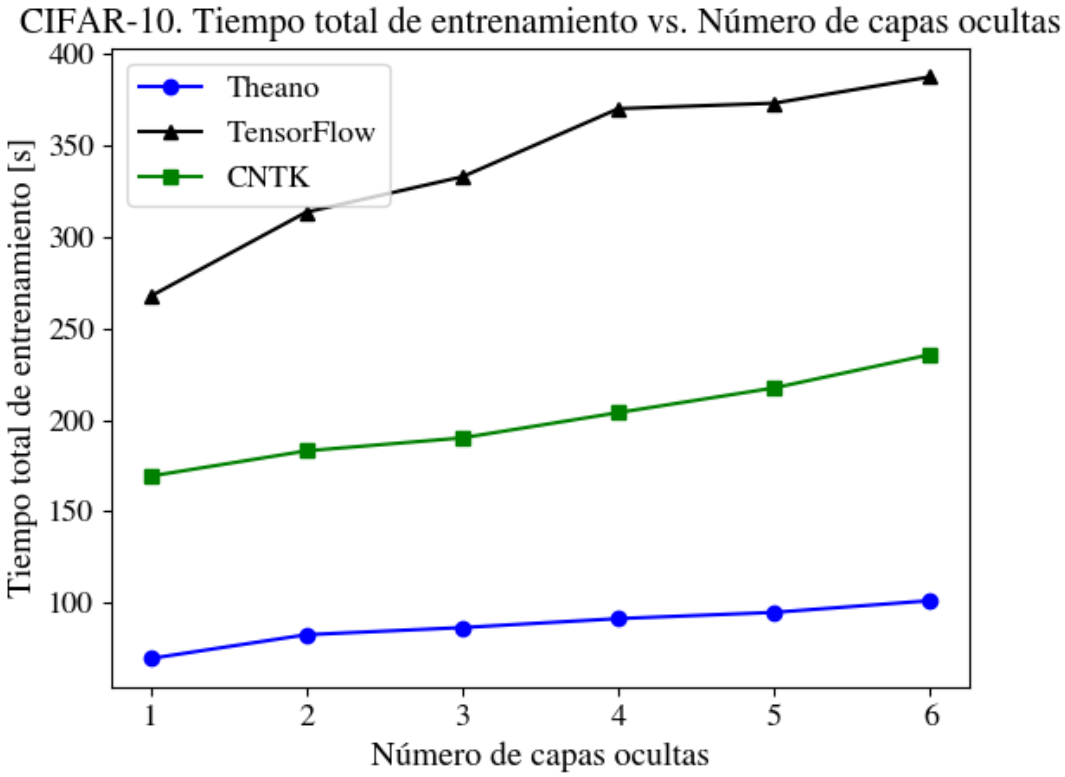


Figura 4.1: Incremento de la profundidad de una red feedforward y su impacto en el tiempo total de entrenamiento. En cada caso, la red resultante ha sido entrenada sobre los descriptores GIST del dataset CIFAR-10.

De acuerdo a la figura 4.2, los tiempos de Theano resultan ser entre 3,8 y 4 veces más bajos que los de TensorFlow, así como también entre 2,2 y 2,4 menores que los de CNTK. Por último, CNTK emplea tiempos entre 1,6 y 1,8 veces más bajos que los de TensorFlow.

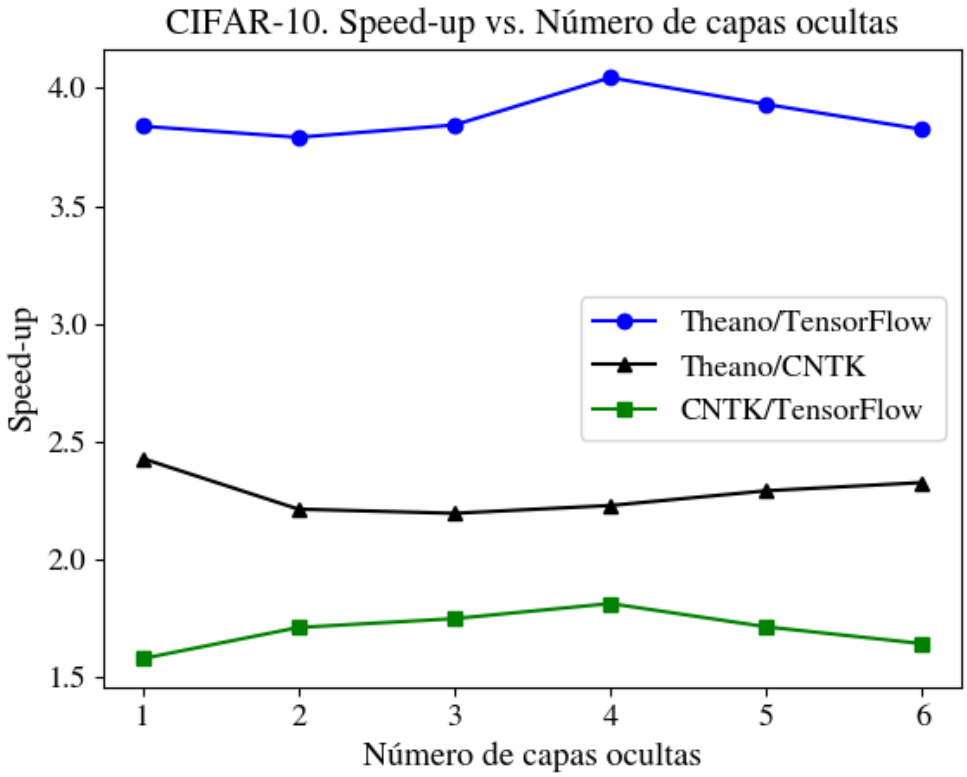


Figura 4.2: Speed-up relativo entre los tiempos de entrenamiento obtenidos por cada par de frameworks al incrementar la profundidad de una red feedforward entrenada sobre los descriptores GIST del dataset CIFAR-10.

4.6.1.1.2. Eficiencia en base al tamaño de mini-batch

Se analiza el efecto de variar el tamaño de mini-batch utilizado en el entrenamiento de FNN-6, comenzando en 8 y finalizando en 512 ejemplos por cada uno. El tamaño de mini-batch en un determinado experimento corresponde al doble del tamaño utilizado en el experimento anterior.

La figura 4.3 muestra que el tiempo total de entrenamiento de la red es inversamente proporcional al tamaño de mini-batch utilizado, análisis que es válido para todos los

frameworks estudiados. Tal como ocurrió al variar el número de capas ocultas, Theano obtiene los tiempos más bajos en todo escenario posible, seguido por CNTK y TensorFlow. Notar, sin embargo, que las diferencias de tiempo entre frameworks son más significativas mientras menor es el tamaño del mini-batch considerado.

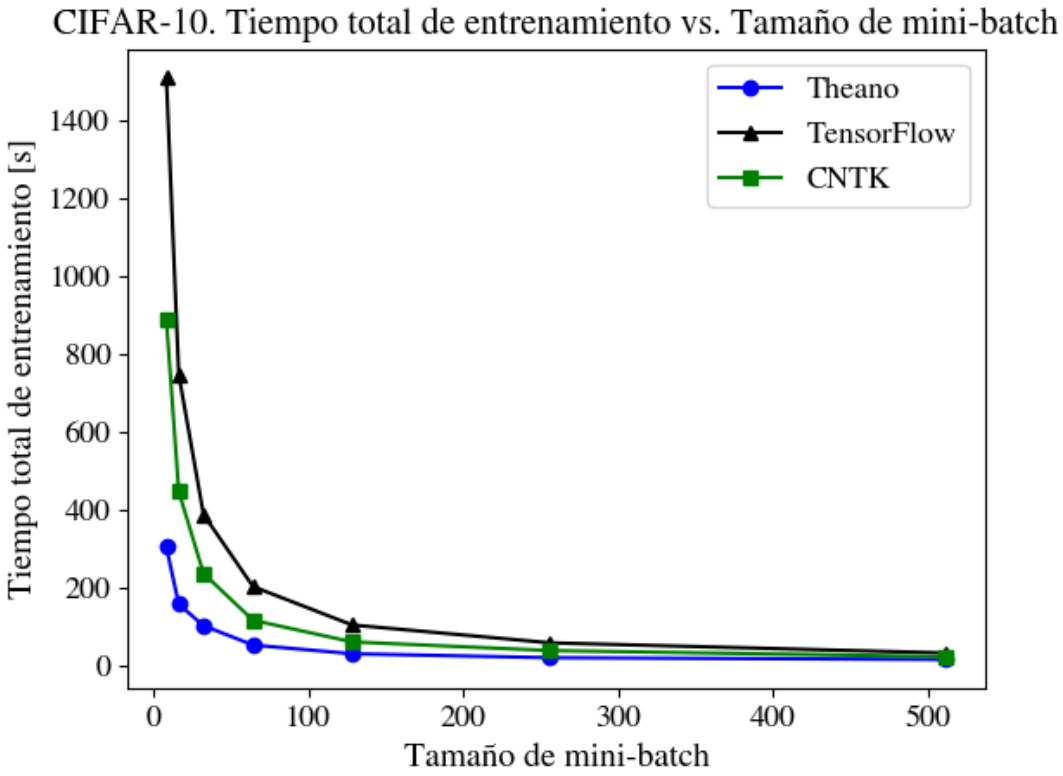


Figura 4.3: Variación del tamaño de mini-batch utilizado en el entrenamiento de FNN-6 y su impacto en el tiempo total de entrenamiento. En cada caso, la red es entrenada sobre los descriptores GIST del dataset CIFAR-10.

En la figura 4.4 se aprecia que Theano es entre 2,2 y 5 veces más rápido que TensorFlow. Además, sus tiempos son entre 1,5 y 2,9 veces más bajos que los de CNTK. Por otro lado, los tiempos de CNTK son entre 1,4 y 1,8 veces más bajos que los de TensorFlow.

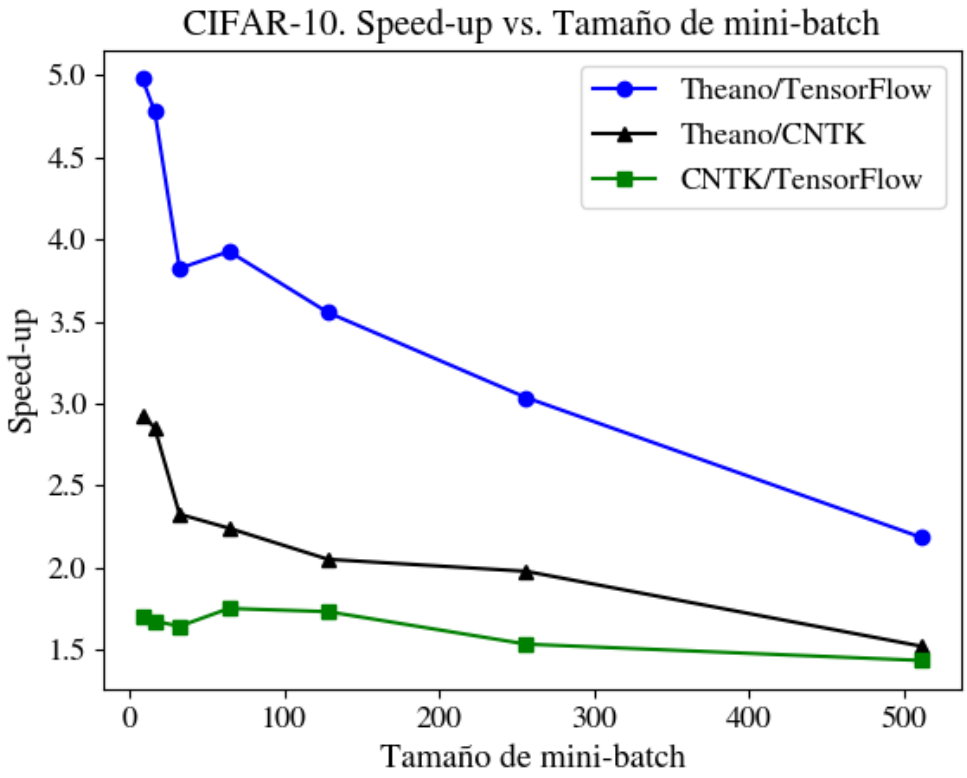


Figura 4.4: Speed-up relativo entre los tiempos de entrenamiento de FNN-6 obtenidos por cada par de frameworks al variar el tamaño de mini-batch. En cada caso, FNN-6 es entrenada sobre los descriptores GIST del dataset CIFAR-10.

4.6.1.1.3. Consumo de memoria GPU

No se han observado variaciones significativas en el consumo de memoria GPU de los frameworks, ya sea al variar la profundidad de la red o el tamaño de mini-batch utilizado. Por ejemplo, la diferencia medida entre el consumo de la red con una capa oculta y con seis capas ocultas es de tan sólo 2 [MiB] en Theano. Ante esto, se informan los consumos al entrenar la red con seis capas ocultas y un tamaño de mini-batch de 512. En el caso de Theano, es de 229 [MiB], en TensorFlow, de 204 [MiB] y en CNTK, es de 186 [MiB].

4.6.1.2. SVHN

Cada imagen es reducida a un vector de 154 atributos, conformado a partir de la concatenación de su histograma de gradientes orientados (también conocido como descriptores HOG, por sus siglas en inglés) y su histograma de tonos.

Para la obtención de los descriptores HOG se determina, en primer lugar, el gradiente de cada pixel de la imagen. El gradiente de un pixel corresponde a un vector que representa la dirección en que se produce un cambio de intensidad dentro de la imagen. Cada gradiente está definido por dos cantidades: Magnitud y ángulo. La magnitud permite cuantificar que tan grande es el cambio de intensidad de un pixel a otro: Mientras mayor es la magnitud del vector, mayor es el cambio en la intensidad. El ángulo, por otro lado, permite conocer la orientación del vector. Lo más común es medir el ángulo en un rango de entre 0^o y 180^o. Notar que los cambios más grandes de intensidad suelen producirse en los contornos de los objetos de una imagen. Luego, se definen dos matrices por cada imagen: Una que almacena las magnitudes de los gradientes y otra los ángulos. A partir de dichas matrices, se construye el histograma, el cual está dividido en 9 categorías posibles, donde cada una representa un determinado rango angular: [0, 20), [20, 40), [40, 60), [60, 80), [80, 100), [100, 120), [120, 140), [140, 160) y [160, 180). De esta manera, cada pixel incrementa la frecuencia de un rango determinado de acuerdo a la magnitud de su gradiente. Por ejemplo, si la magnitud del gradiente de un pixel es 25 y la orientación del mismo es de 73, entonces la frecuencia en el rango [60, 80] aumentará en 25 unidades.

Por otra parte, el histograma de tonos permite contabilizar la cantidad de píxeles que poseen un cierto nivel de luminosidad.

4.6.1.2.1. Eficiencia en base a número de capas ocultas

Se varía la cantidad de capas ocultas existentes en la red entre 1 y 6, en base a los mismos criterios considerados en el caso de CIFAR-10. Así, las características de cada una de las arquitecturas estudiadas se detallan en la tabla 4.4.

Cuadro 4.5: Características de cada una de las redes feedforward entrenadas sobre la representación combinada de los descriptores HOG y los histogramas de tonos del dataset SVHN. Una arquitectura denotada por FNN-i representa a una red feedforward integrada por i capas ocultas.

Nombre arquitectura	**Número de neuronas por capa**	**Número de parámetros**
FNN-1	154	25.420
FNN-2	102, 52	21.696
FNN-3	88, 44, 22	18.776
FNN-4	82, 41, 20, 11	17.304
FNN-5	79, 40, 20, 10, 5	16.590
FNN-6	78, 39, 19, 10, 5, 3	16.244

Al término de toda red FNN-i, existe una capa totalmente conectada de 10 neuronas (capa de salida). Para el entrenamiento de los modelos, se han escogido los mismos hiper parámetros que en el caso de CIFAR-10.

La figura 4.5 muestra que, al igual que con CIFAR-10, Theano emplea los tiempos más bajos de entrenamiento en todos los experimentos realizados, seguido por CNTK y TensorFlow. En general, se observa que las curvas asociadas a cada framework presentan un comportamiento similar al observado en el dataset anterior.

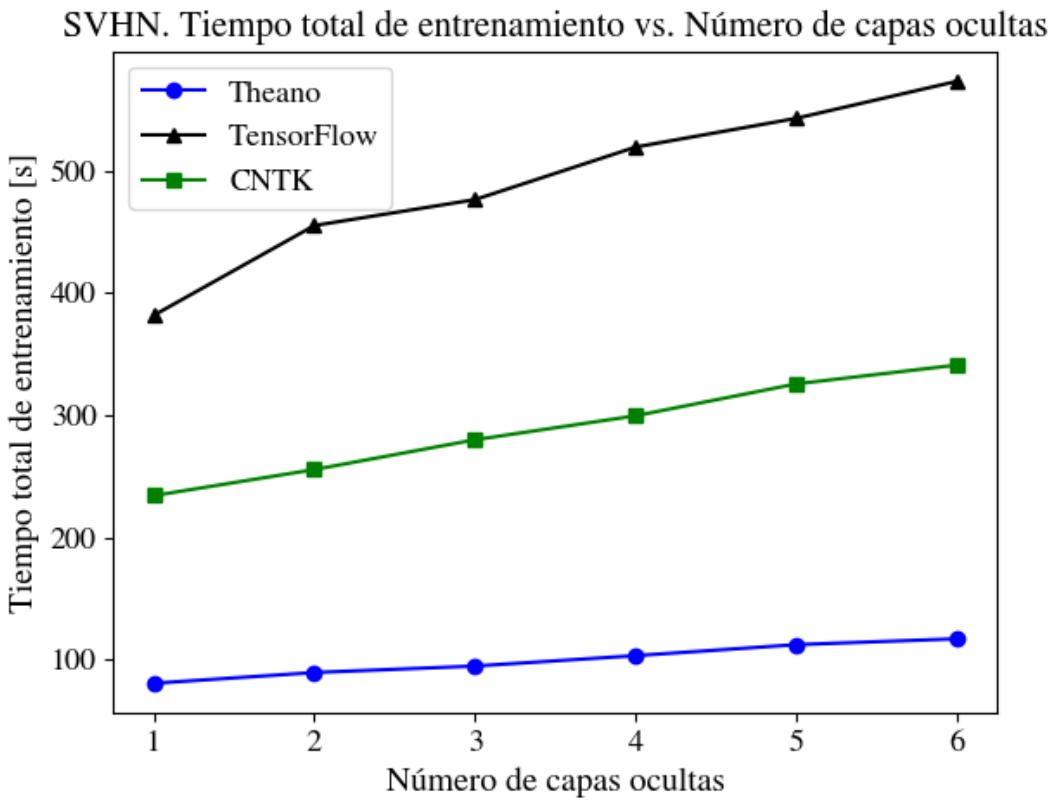


Figura 4.5: Incremento de la profundidad de una red feedforward y su impacto en el tiempo total de entrenamiento. En cada caso, la red resultante ha sido entrenada sobre la representación combinada de los descriptores HOG y los histogramas de tonos del dataset SVHN.

Según muestra la figura 4.6, los tiempos de Theano resultan ser entre 4,7 y 5 veces menores que los de TensorFlow, mientras que los de Theano son entre 2,9 y 3 veces más bajos que los de CNTK. Por otra parte, el speed-up relativo entre los tiempos de CNTK y TensorFlow varía entre 1,6 y 1,8.

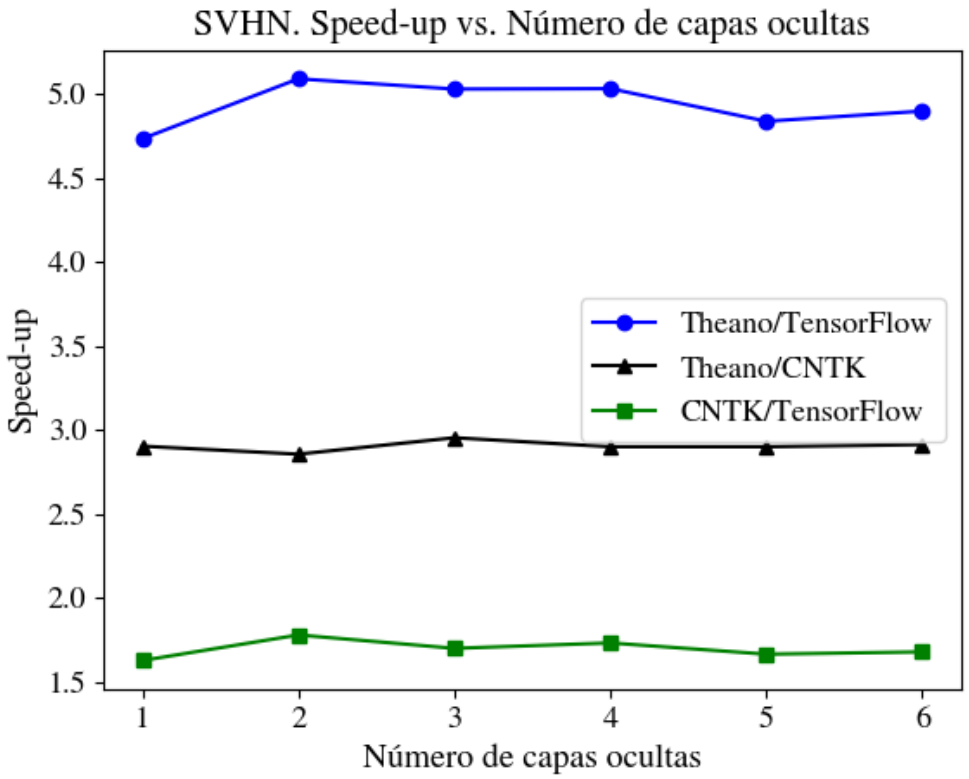


Figura 4.6: Speed-up relativo entre los tiempos de entrenamiento obtenidos por cada par de frameworks al incrementar la profundidad de una red feed-forward entrenada sobre la representación combinada de los descriptores HOG y los histogramas de tonos del dataset SVHN.

4.6.1.2.2. Eficiencia en base al tamaño de mini-batch

En forma similar a la sección 4.6.1.1.2, se aumenta paulatinamente el tamaño de mini-batch definido para el entrenamiento de FNN-6, siguiendo el mismo esquema de dicha sección.

La figura 4.7 muestra que, nuevamente, existe una relación de proporcionalidad inversa entre el tiempo total de entrenamiento de la red y el tamaño de mini-batch escogido, análisis que es válido para todo framework.

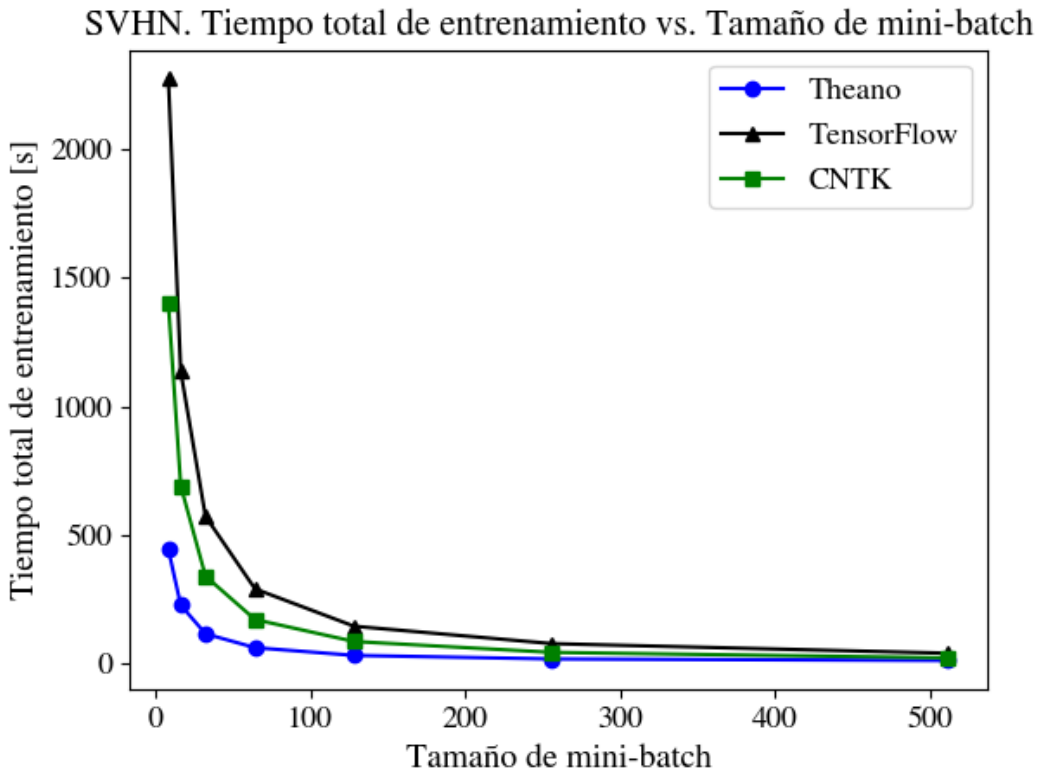


Figura 4.7: Variación del tamaño de mini-batch utilizado en el entrenamiento de FNN-6 y su impacto en el tiempo total de entrenamiento. En cada caso, la red es entrenada sobre la representación combinada de los descriptores HOG y los histogramas de tonos del dataset SVHN.

De acuerdo a la figura 4.8, Theano emplea tiempos entre 3,1 y 5,1 veces menores que los de TensorFlow. También, sus tiempos son entre 1,8 y 3,1 veces más bajos que los de CNTK. Por otro lado, los tiempos de CNTK son entre 1,6 y 1,8 veces menores que los de TensorFlow.

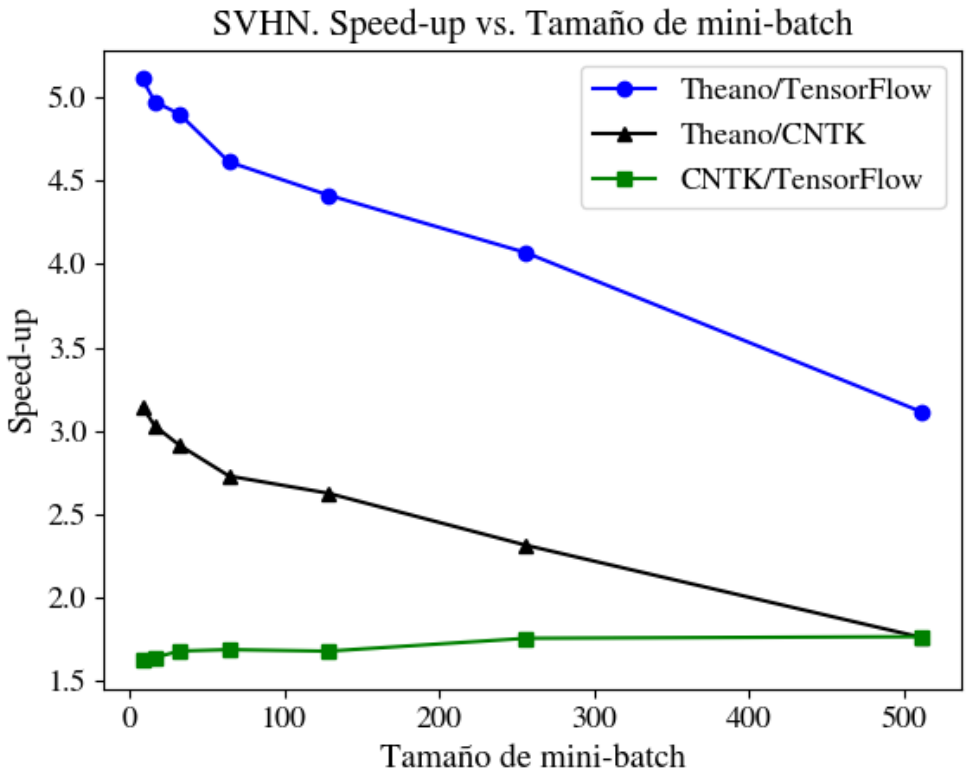


Figura 4.8: Speed-up relativo entre los tiempos de entrenamiento de FNN-6 obtenidos por cada par de frameworks al variar el tamaño de mini-batch. En cada caso, FNN-6 es entrenada sobre la representación combinada de los descriptores HOG y los histogramas de tonos del dataset SVHN.

4.6.1.2.3. Consumo de memoria GPU

Al igual que en el caso de CIFAR-10, no se registraron cambios importantes en el consumo de memoria GPU de los frameworks, ya sea al variar la profundidad de la red o el tamaño de mini-batch utilizado. Luego, nuevamente se informan los consumos al entrenar la red con seis capas ocultas y con un tamaño de mini-batch de 512. En el caso de Theano, es de 217 [MiB], en TensorFlow, de 176 [MiB] y en CNTK, es de 149 [MiB].

4.6.1.3. MNIST

A diferencia de secciones anteriores, no se aplica un extractor de características sobre las imágenes, siendo estas transformadas en vectores de 784 atributos, pero sin alterar el contenido original de los píxeles.

4.6.1.3.1. Eficiencia en base a número de capas ocultas

Como en secciones anteriores, se estudia la variación del número de capas ocultas. Las seis arquitecturas evaluadas son detalladas en la tabla 4.6.

Cuadro 4.6: Características de cada una de las redes feedforward entrenadas sobre los píxeles en bruto de MNIST. Una arquitectura denotada por FNN-i representa a una red feedforward integrada por i capas ocultas.

Nombre arquitectura	Número de neuronas por capa	Número de parámetros
FNN-1	784	623.290
FNN-2	522, 262	549.426
FNN-3	448, 224, 112	478.586
FNN-4	418, 209, 104, 53	443.646
FNN-5	404, 202, 101, 51, 26	426.277
FNN-6	398, 199, 99, 50, 25, 13	418.384

Al término de toda red FNN-i, existe una capa totalmente conectada de 10 neuronas (capa de salida). Para el entrenamiento de los modelos, se han escogido los mismos hiper parámetros que en el caso de CIFAR-10 y SVHN.

De acuerdo a la figura 4.9, es Theano quien nuevamente experimenta los tiempos de entrenamiento más bajos, seguido por CNTK y TensorFlow. Las curvas asociadas a los frameworks son similares a las observadas en datasets anteriores.

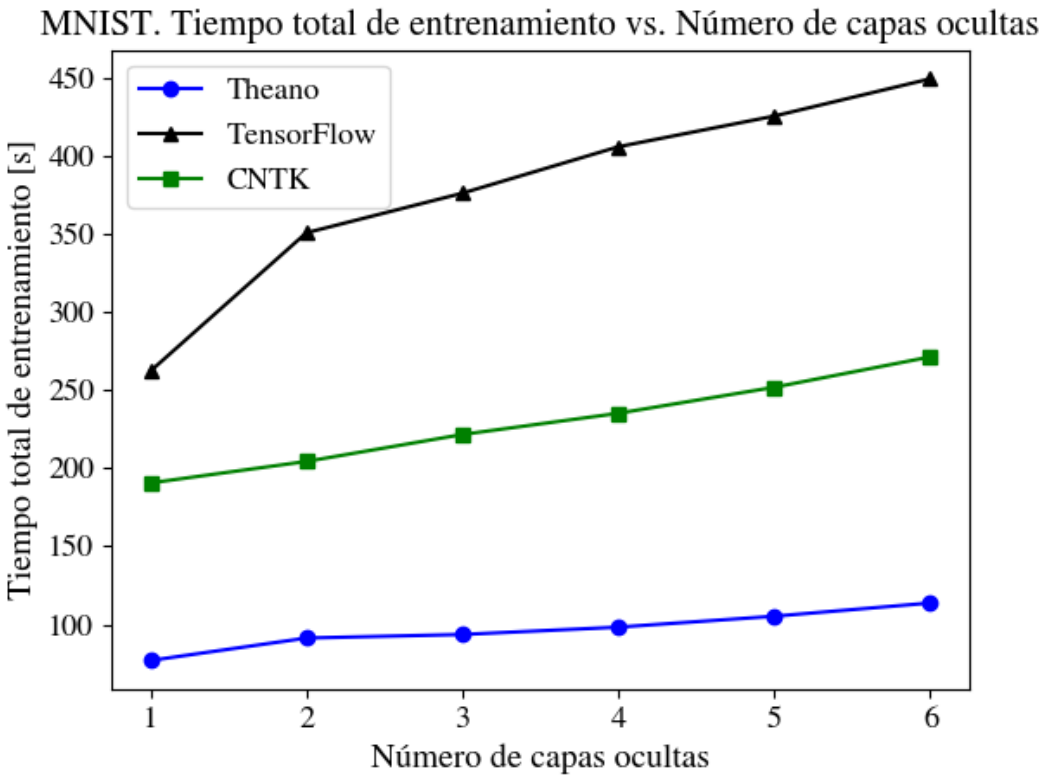


Figura 4.9: Incremento de la profundidad de una red feedforward y su impacto en el tiempo total de entrenamiento. En cada caso, la red resultante es entrenada sobre la concatenación de píxeles del dataset MNIST.

Según muestra la figura 4.2, los tiempos de Theano son entre 3,4 y 4,1 veces más bajos que los de TensorFlow. Por otra parte, los tiempos de Theano son entre 2,2 y 2,5 veces menores que los de CNTK. Además, CNTK emplea tiempos entre 1,4 y 1,7 veces más bajos que los de TensorFlow.

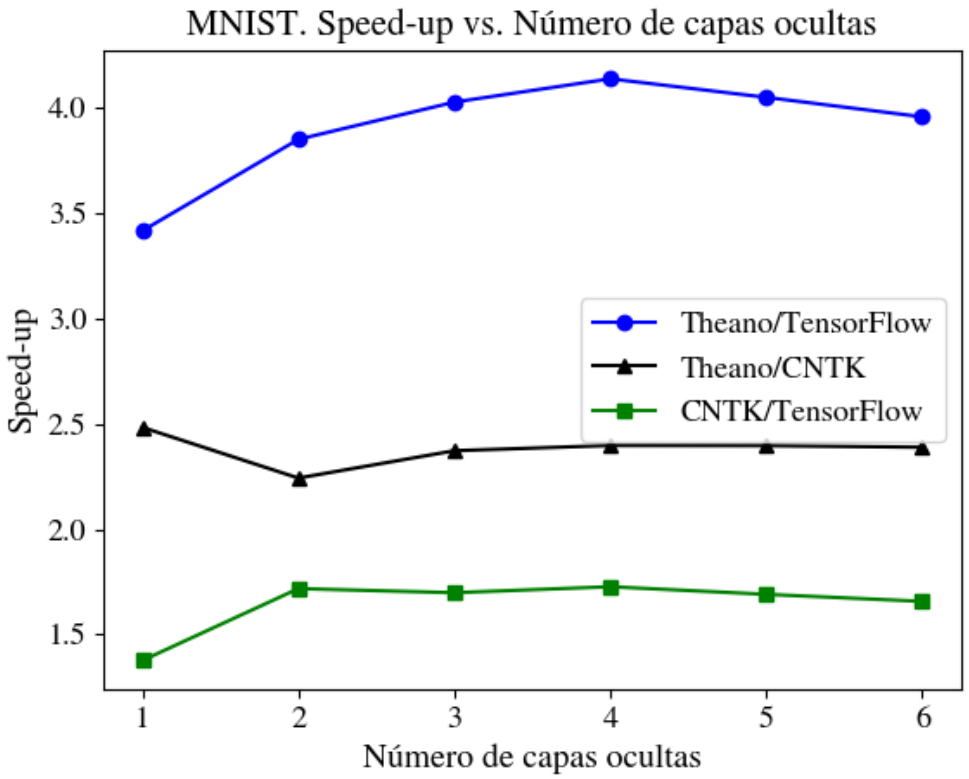


Figura 4.10: Speed-up relativo entre los tiempos de entrenamiento obtenidos por cada par de frameworks al variar la profundidad de una red feedforward. En cada caso, la red resultante es entrenada sobre la concatenación de píxeles del dataset MNIST.

4.6.1.3.2. Eficiencia en base al tamaño de mini-batch

Como en secciones previas, se estudia el efecto de cambiar el tamaño de mini-batch.

La figura 4.12 muestra que, al igual que en datasets anteriores, existe una relación de proporcionalidad inversa entre el tiempo total de entrenamiento del modelo y el tamaño de mini-batch escogido, premisa que es válida para todos los frameworks estudiados.

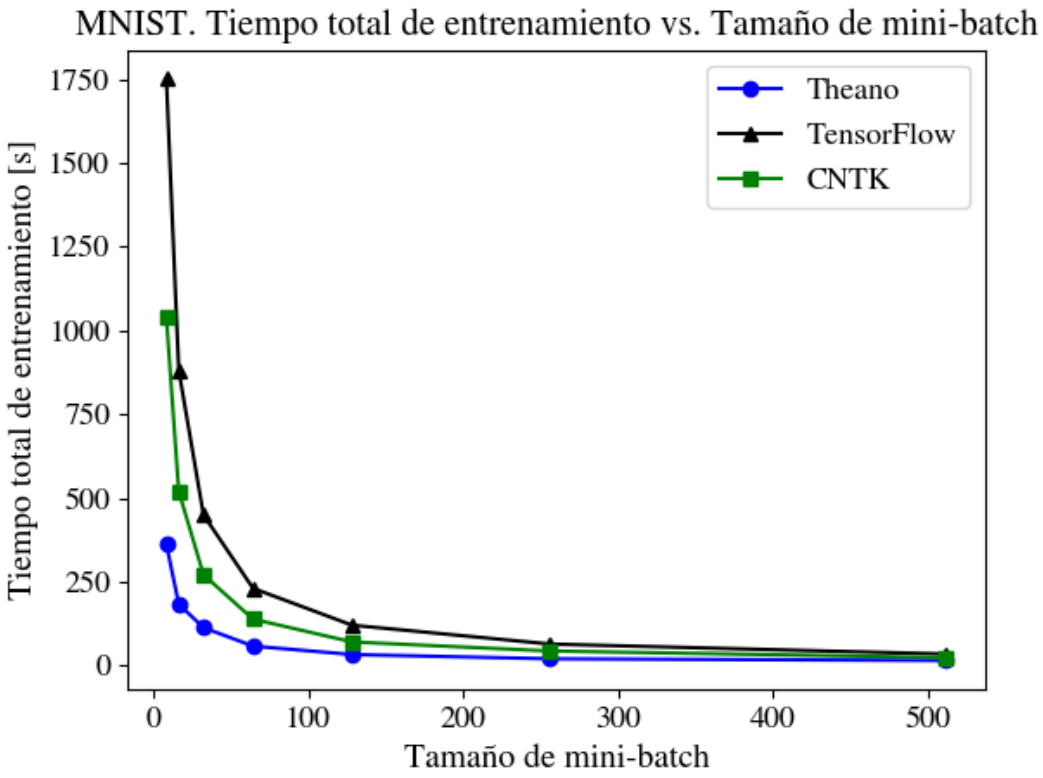


Figura 4.11: Variación del tamaño de mini-batch utilizado en el entrenamiento de FNN-6 y su impacto en el tiempo total de entrenamiento. En cada caso, FNN-6 es entrenada sobre la concatenación de píxeles del dataset MNIST.

Según muestra la figura 4.12, Theano es entre 2,2 y 4,8 veces más rápido que TensorFlow, así como también sus tiempos son entre 1,6 y 2,9 veces más bajos que los de CNTK. Por último, CNTK resulta ser entre 1,4 y 1,7 veces más rápido que TensorFlow.

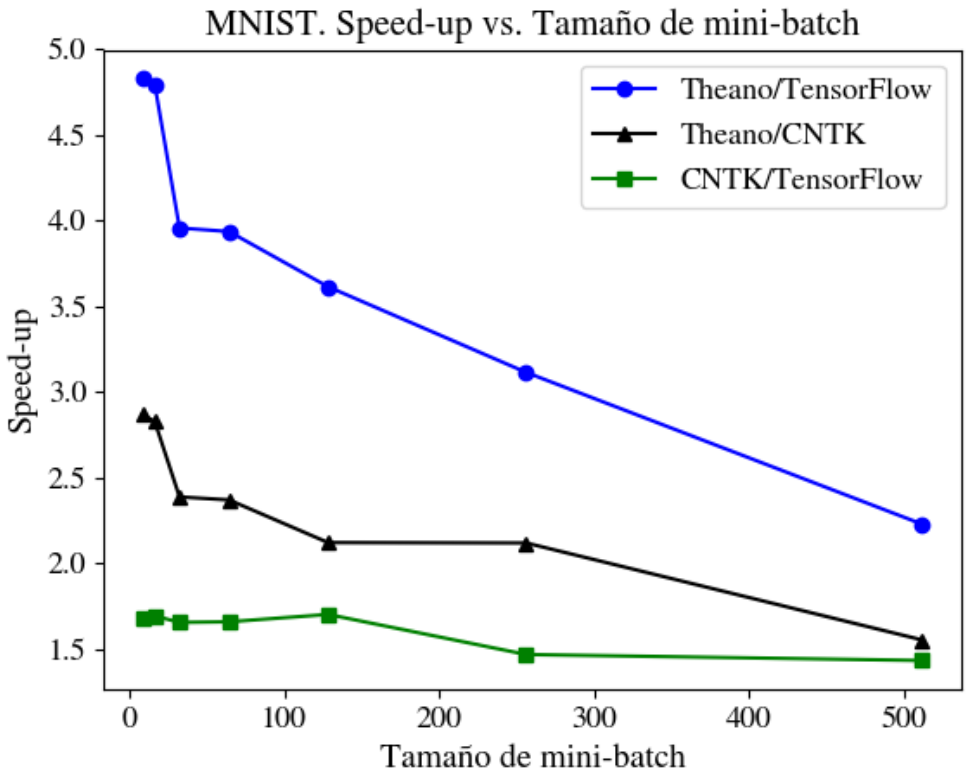


Figura 4.12: Speed-up relativo entre los tiempos de entrenamiento de FNN-6 obtenidos por cada par de frameworks al variar el tamaño de mini-batch. En cada caso, FNN-6 es entrenada sobre la concatenación de píxeles del dataset MNIST.

4.6.1.3.3. Consumo de memoria GPU

Como en datasets anteriores, no se registraron cambios importantes en el consumo de memoria GPU de los frameworks. Así, se informan los consumos al entrenar la red con seis capas ocultas y con un tamaño de mini-batch de 512. En el caso de Theano, es de 225 [MiB], en TensorFlow, de 188 [MiB] y en CNTK, de 177 [MiB].

4.6.2. Redes convolucionales

Para el estudio de redes convolucionales, se han considerado todos los datasets que se corresponden con problemas de clasificación de imágenes. Por cada uno de ellos, se

estudiaron dos variantes, elegidas en base a los siguientes criterios:

- No hacen uso de data aumentada.
- No hacen uso de ensamblados de redes.
- Utilizan módulos que pueden encontrarse en los frameworks, o bien, usan módulos que corresponden a variaciones de los que pueden encontrarse en los frameworks pero no requieren cambiar código fuente. Entiéndase por módulos todos los tipos de capas neuronales, funciones de pérdida, métodos de optimización, esquemas de inicialización, funciones de activación y esquemas de regularización que pueden encontrarse en Keras.

Como primera variante, se escoge la arquitectura más cercana a la red state of the art del problema respectivo de acuerdo a la información disponible en los sitios de Martin Thoma [2] y de Rodrigo Benenson [3]. Lo anterior es aplicable para todos los datasets, excepto NORB, para el cual se recolectaron los mejores resultados posibles a partir de la revisión de publicaciones disponibles en Google Scholar [4]. De aquí en adelante, toda arquitectura que cumpla con estas características será denominada ”maestra”.

Se utilizará la siguiente nomenclatura:

- C: Capa convolucional.
- P: Capa de pooling.
- FC: Capa totalmente conectada.

Para la selección de la segunda variante, se procede en forma análoga al párrafo anterior, pero la arquitectura debe contar con una estructura más simple respecto a las escogidas en el primer caso, es decir, de tipo CxPxCxPxFC o CxCxPxCxCxFC y será utilizada con el fin de variar el número de capas convolucionales, la cantidad de filtros presentes en las mismas y el tamaño de los filtros. Con lo anterior, se busca determinar el impacto que tienen estas modificaciones sobre las métricas de rendimiento consideradas. De ahora en adelante, toda arquitectura elegida para este propósito será denominada ”básica”.

[2] https://martin-thoma.com/sota/

[3] http://rodrigob.github.io/are_we_there_yet/build/#classification-dataset-type

[4] https://scholar.google.com/scholar?hl=en&as_sdt=2005&sciodt=0%2C5&cites=18933894925073173 33&scipsc=&q=&btnG=

4.6.2.1. MNIST

4.6.2.1.1. Arquitectura maestra

Se escogió la red DNN-W20 propuesta en [41]. Este modelo está integrado por las siguientes capas, descritas en orden de aparición:

- Capa convolucional con 20 filtros de tamaño 4 x 4, cada uno.
- Capa de max pooling con pooling size de 2 x 2.
- Capa convolucional con 40 filtros de tamaño 5 x 5, cada uno.
- Capa de max pooling con pooling size de 3 x 3.
- Capa totalmente conectada con 150 neuronas.
- Capa totalmente conectada con 10 neuronas y función de activación softmax (capa de salida).

Se hace uso de stride 1 en ambas capas convolucionales. Además, La arquitectura está conformada por 34.040 parámetros entrenables.

Como promedio de cinco repeticiones del experimento, los autores logran obtener un error de clasificación de prueba de $0,39 \pm 0,08$.

Las imágenes son normalizadas de tal forma que el ancho de cada una es de 20 píxeles. Los pesos de la red son inicializados en el rango [-0,05, 0,05] siguiendo una distribución uniforme. Se usa tangente hiperbólica escalada como función de activación de toda capa oculta, la que está dada por:

$$f(x) = 1,7159 \cdot tanh(\frac{2x}{3}) \tag{4.3}$$

El modelo es entrenado con una tasa de aprendizaje inicial de 0,001, la que es multiplicada por un factor de 0,993 hasta que alcanza el valor 0,00003, lo que lleva a la ejecución de 500 epochs. No hay uso de momentum o weight decay. Si bien en la citada publicación se utilizó Gradiente Descendiente Online como método de entrenamiento, en esta memoria se hizo uso de SGD con tamaño de mini-batch 128. Tomar en cuenta que en el trabajo original se distorsionaron ejemplos de entrenamiento escogidos alea-

toriamente previo a cada epoch por medio de transformaciones, rotaciones y cambios de escala, lo que no es hecho en este trabajo.

Los resultados obtenidos por cada framework se muestran en la tabla 4.7. Se observa como el tiempo de entrenamiento total en el caso de Theano es 1,8 veces superior al de TensorFlow y 2,5 veces más alto que el de CNTK, mientras que TensorFlow emplea un tiempo 1,4 veces superior al de CNTK. Por otra parte, el consumo de memoria GPU en el caso de TensorFlow es 1,4 y 1,6 veces superior al de CNTK y de Theano, respectivamente. La razón entre los consumos de CTNK y Theano es de 1,2. Los errores de entrenamiento son comparables entre frameworks, dándose la mayor diferencia entre CNTK y TensorFlow (0,13 %). Por otra parte, los errores de prueba son aún más similares entre sí, donde CNTK es un 0,04 % más preciso que Theano y TensorFlow. Estas diferencias pueden deberse principalmente a la aleatoriedad introducida por el esquema de inicialización utilizado y por ciertas operaciones internas de cuDNN[5].

Cuadro 4.7: Rendimiento sobre arquitectura DNN-W20. Dataset: MNIST.

Framework	**Tiempo de entrenamiento total [s]**	**Consumo de memoria GPU [MiB]**	**Error de entrenamiento [%]**	**Error de prueba [%]**
Theano	2400,0	232	1,39	1,88
TensorFlow	1352,7	378	1,27	1,88
CNTK	965,9	272	1,40	1,84

4.6.2.1.2. Arquitectura básica

Se considera el modelo CNN-Softmax propuesto en [42], el cual posee las siguientes capas, descritas en orden de aparición:

[5]La documentación de cuDNN indica que no se garantiza la reproducibilidad ni en la obtención de los gradientes con respecto a los pesos ni con respecto al input recibido por una convolución.

- Capa convolucional de 32 filtros de 5 x 5, cada uno.
- Capa de max pooling con pooling size de 2 x 2 y stride de 2.
- Capa Convolucional de 64 filtros de 5 x 5, cada uno.
- Capa de max pooling con pooling size de 2 x 2 y stride de 2.
- Capa totalmente conectada con 500 neuronas y dropout de 0,5.
- Capa totalmente conectada con 10 neuronas y función de activación softmax (capa de salida).

Gracias a esta arquitectura, los autores obtuvieron un error de prueba de 0,56 %.

Las imágenes son preprocesadas por medio de GCN. Todas las capas ocultas de la red hacen uso de la función de activación ReLU. Para el entrenamiento del modelo, se utiliza SGD con tamaño de mini-batch 128 durante 110 epochs, weight decay de 0,0005 y momentum fijo de 0,9.

No fue posible conocer de antemano la tasa de aprendizaje utilizada, debido a que no es mencionada en la publicación. Al momento de contactar a uno de los autores de la misma, este indicó la imposibilidad de acceder a los experimentos originales, que es donde eventualmente se encontraría esta información. Sin embargo, el autor recomendó hacer uso de un proceso de validación para encontrar una tasa que permita obtener buenos resultados. Siguiendo esta instrucción, se construyó un conjunto de validación conformado por 5000 ejemplos escogidos aleatoriamente a partir del conjunto de entrenamiento y que sirvió para determinar que con una tasa de aprendizaje fija de 0,01 es posible acercarse al rendimiento original, ya que se obtiene un error de entrenamiento del 0,53 % y un error de validación del 0,81 %.

De esta manera, se procedió a entrenar el modelo con el conjunto de entrenamiento original usando esta tasa. Los resultados obtenidos se muestran en la tabla 4.8. En esta ocasión, Theano emplea 2,2 y 2,3 veces más tiempo que TensorFlow y CNTK. Además, el tiempo total de entrenamiento de TensorFlow es 1,1 veces superior al de CNTK. En consecuencia, es CNTK quien nuevamente logra el mejor rendimiento en el proceso de entrenamiento, aunque por un margen más estrecho en relación a TensorFlow si se compara con la arquitectura maestra. El consumo de memoria GPU de TensorFlow es 1,1 veces mayor al de CNTK y 1,45 veces superior al de Theano, mientras que la razón entre los consumos de CNTK y Theano es de 1,35. Los errores de entrenamiento

vuelven a ser cercanos entre sí, de tal manera que la mayor diferencia se presenta entre CNTK y TensorFlow (0,06 %). De igual forma, los errores de prueba son prácticamente idénticos; luego, la mayor diferencia también se da entre CNTK y TensorFlow (0,02 %).

Cuadro 4.8: Rendimiento sobre arquitectura CNN-Softmax. Dataset: MNIST.

Framework	**Tiempo de entrenamiento total [s]**	**Consumo de memoria GPU [MiB]**	**Error de entrenamiento [%]**	**Error de prueba [%]**
Theano	830,2	258	0,54	0,82
TensorFlow	377,8	374	0,49	0,81
CNTK	358,9	349	0,55	0,83

4.6.2.1.3. Variación de la cantidad de ejemplos de entrenamiento

A continuación, se varía la cantidad de ejemplos provistos para el entrenamiento del modelo CNN-Softmax y se estudia como esto afecta el rendimiento de cada framework. Notar que tanto en este dataset como en los restantes, no se ha medido el consumo de memoria GPU, ya que el tamaño de mini-batch permanece constante y es independiente de la cantidad de ejemplos de entrenamiento utilizada.

En la tabla 4.9 se muestran los porcentajes de ejemplos que se han considerado, medidos con respecto al tamaño original del conjunto de entrenamiento.

Cuadro 4.9: Fracciones de ejemplos usadas para el entrenamiento de CNN-Softmax. Dataset: MNIST.

Fracción de ejemplos [%]	**Número de ejemplos**
33	19.800
40	24.000
57	34.200
64	38.400
83	49.800
100	60.000

La figura 4.13 muestra como varía el tiempo total de entrenamiento de CNN-Softmax en función de la fracción de ejemplos utilizados por cada framework. En ella se verifica que, para toda fracción posible, Theano emplea una cantidad ampliamente superior de tiempo para entrenar la arquitectura, respecto a TensorFlow y CNTK. Además, la recta asociada a este framework posee una pendiente mucho más pronunciada a diferencia de las rectas restantes, lo que quiere decir que su rendimiento es poco escalable con el aumento de la cantidad de ejemplos de entrenamiento. Por otra parte, TensorFlow y CNTK poseen rendimientos comparables, siendo este último el más rápido por un estrecho margen.

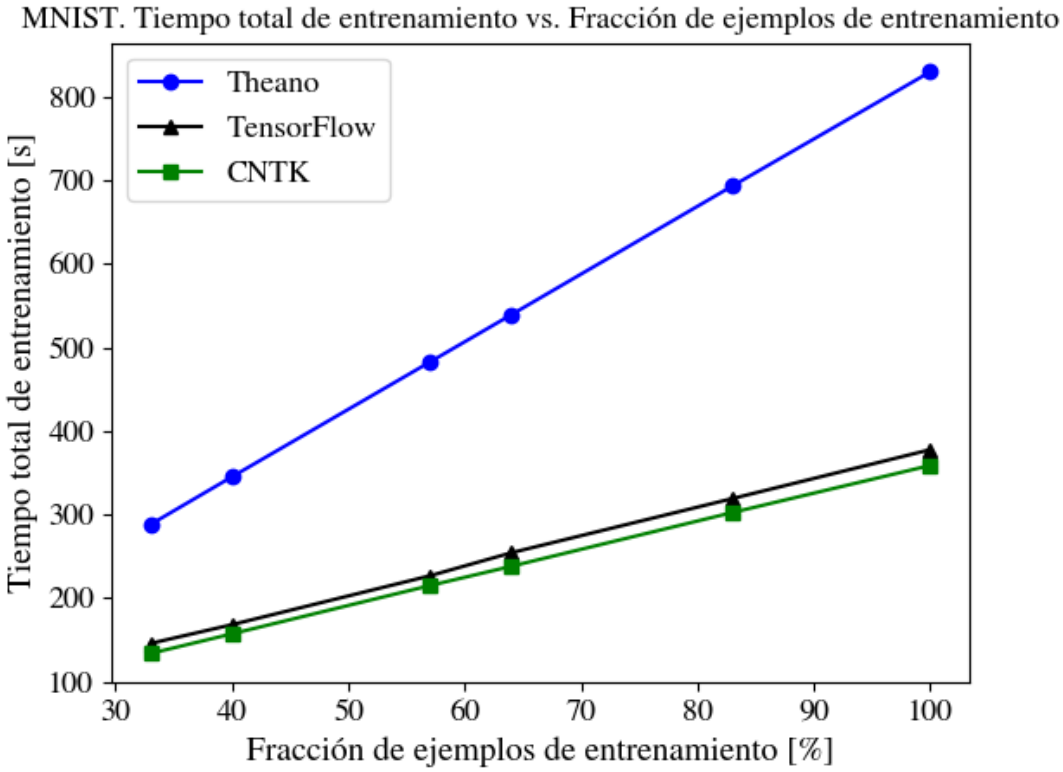


Figura 4.13: Variación de la fracción de ejemplos provistos para el entrenamiento de CNN-Softmax y su impacto en el tiempo total de entrenamiento. Dataset: MNIST.

En la figura 4.14 se muestra el cambio en la cantidad promedio de ejemplos de entrenamiento procesados por segundo al variar la fracción de datos considerada. Nuevamente, Theano presenta el rendimiento más bajo, pues es quien siempre procesa la menor cantidad de ejemplos por unidad de tiempo, siendo también incapaz de aumentar la tasa de procesamiento en forma significativa al proveer una mayor volumen de ejemplos a la red. Luego, TensorFlow y CNTK vuelven a presentar rendimientos similares, si bien en este caso las diferencias entre ambos frameworks son un poco más marcadas. Así, es CNTK quien, para toda fracción de ejemplos posible, es capaz de procesar la mayor cantidad de datos por segundo.

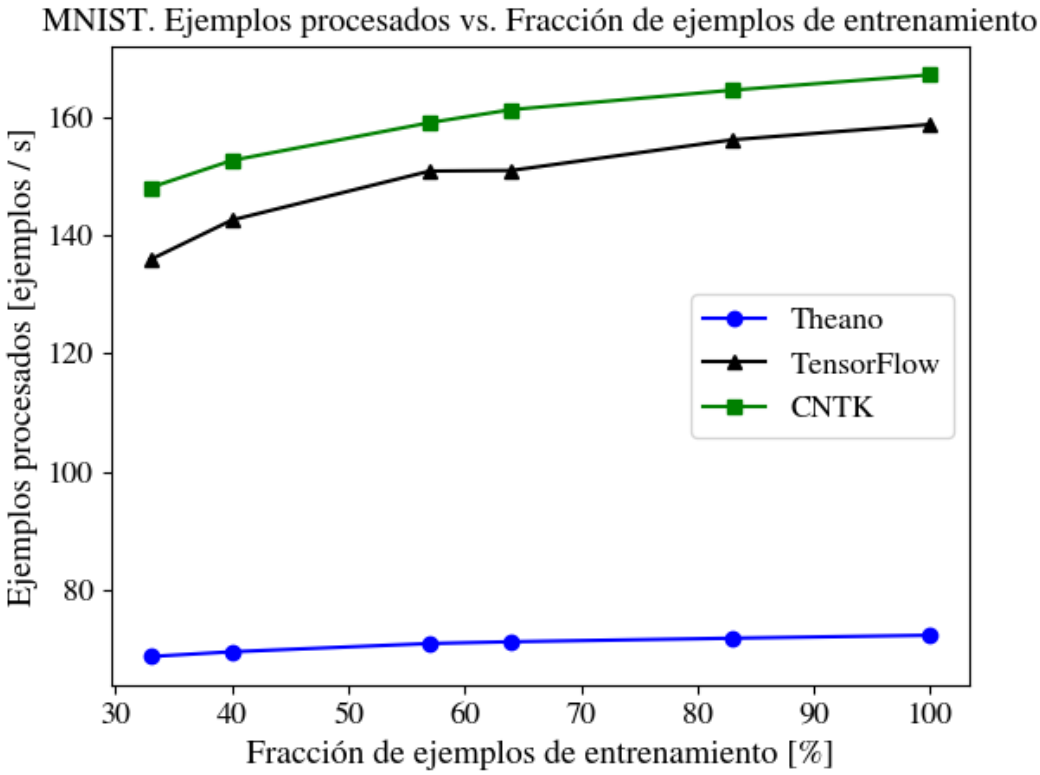


Figura 4.14: Variación de la fracción de ejemplos provistos para el entrenamiento de CNN-Softmax y su impacto en la cantidad promedio de ejemplos procesados. Dataset: MNIST.

La figura 4.15 reafirma la idea de que los tiempos totales de entrenamiento de CNTK y TensorFlow son muy similares entre sí, lo que se manifiesta en que, en cada escenario posible, el cociente entre el tiempo de entrenamiento del primero y del segundo oscila entre 0,91 y 0,95. TensorFlow, por su parte, presenta tiempos entre 2 y 2,2 veces menores que los de Theano, mientras que CNTK obtiene tiempos entre 2,2 y 2,3 veces más bajos que los de Theano.

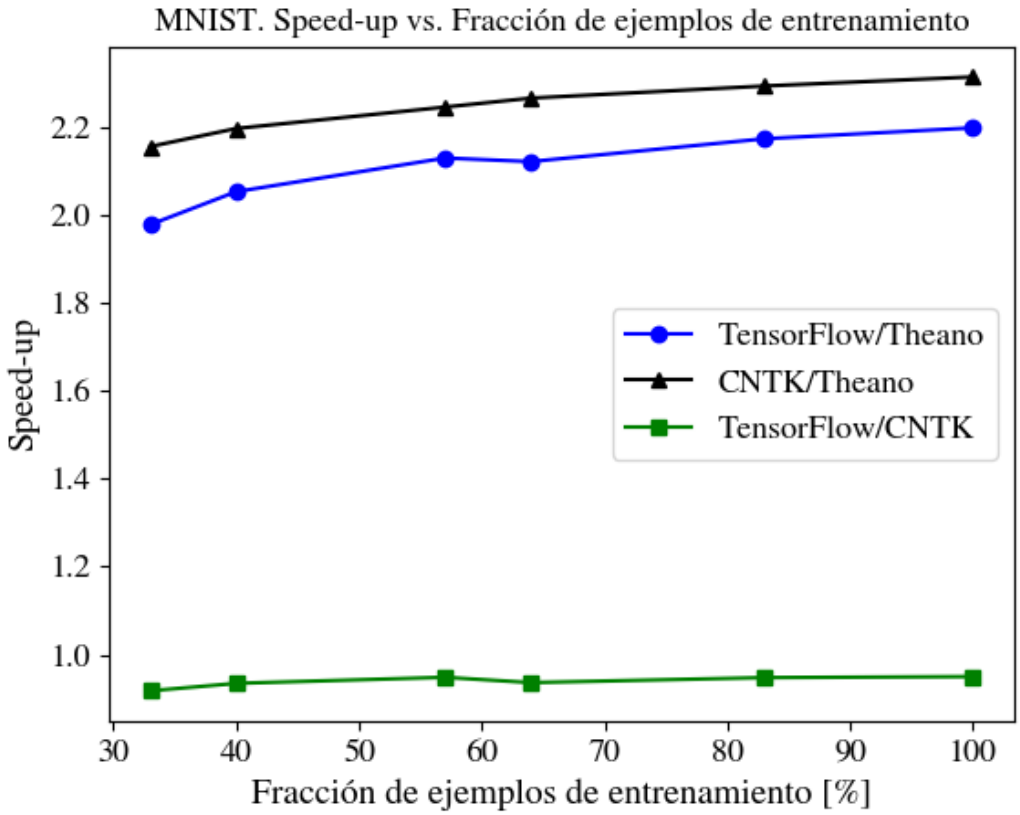


Figura 4.15: Speed-up relativo entre los tiempos de entrenamiento de CNN-Softmax para cada par de frameworks en función de la fracción de ejemplos provistos. Dataset: MNIST.

4.6.2.1.4. Variación de la cantidad de capas convolucionales

En esta sección, se aumenta la cantidad de capas convolucionales de CNN-Softmax en forma progresiva y se analiza el rendimiento de los frameworks en cada caso. La tabla 4.10 expone cada uno de los casos evaluados. Un factor de aumento n implica que cada capa convolucional de la red es replicada $n - 1$ veces. Notar que ha sido necesario aplicar padding sobre los inputs recibidos por las capas convolucionales con el fin de evitar que las dimensiones de los outputs generados se reduzcan prematuramente.

Cuadro 4.10: Aumento en la cantidad de capas convolucionales de CNN-Softmax, junto con el número total de parámetros del modelo resultante. Dataset: MNIST.

Factor de aumento	**Cantidad total de capas convolucionales**	**Cantidad total de parámetros**
1	2	1.625.606
2	4	1.753.702
3	6	1.881.798
4	8	2.009.894
5	10	2.137.990

La figura 4.16 muestra que, para todo framework, el tiempo total de entrenamiento varía linealmente en función del aumento en la cantidad de capas convolucionales. Theano es quien peor escala con dicho aumento, mientras que TensorFlow y CNTK presentan rendimientos similares, si bien sus curvas de tiempo comienzan a distanciarse levemente mientras mayor es el número de capas en la red.

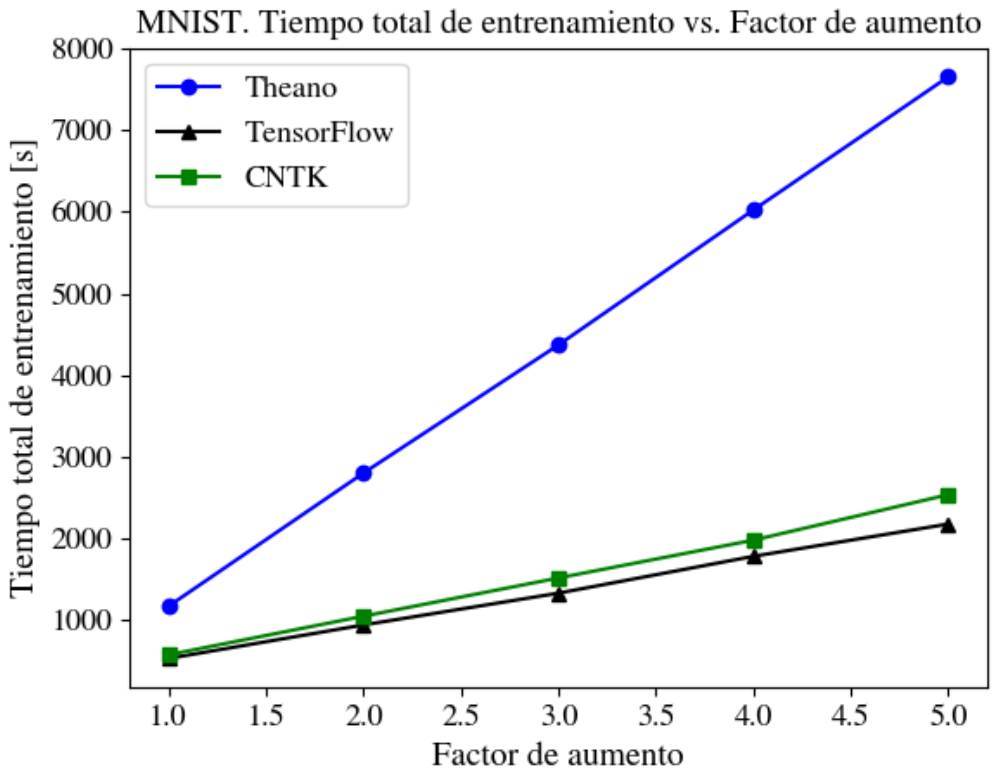


Figura 4.16: Incremento de la cantidad de capas convolucionales en CNN-Softmax y su impacto en el tiempo total de entrenamiento. Dataset: MNIST.

Los tiempos totales de TensorFlow son entre 2,2 y 3,5 veces más bajos que los de Theano, así como también los de CNTK son entre 2 y 3 veces menores que los de CNTK, según muestra la figura 4.17, situación que ratifica el pobre rendimiento de Theano. Por otra parte, TensorFlow resulta ser cerca de 1,1 veces más rápido que TensorFlow en todo escenario.

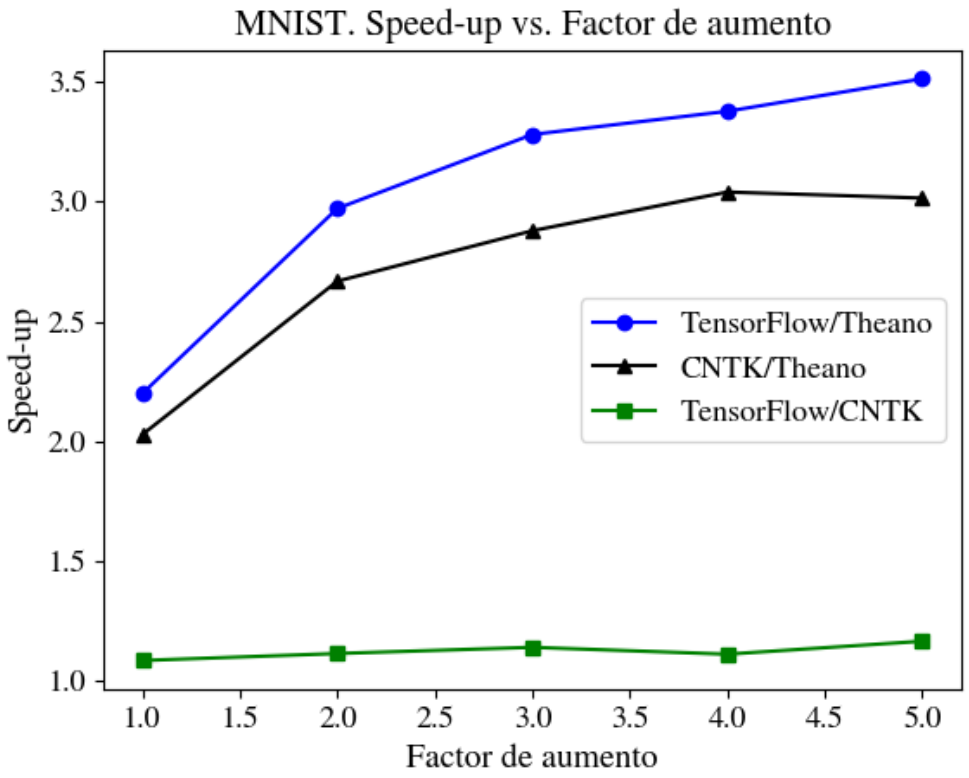


Figura 4.17: Speed-up relativo entre los tiempos de entrenamiento de CNN-Softmax para cada par de frameworks en función del factor de aumento. Dataset: MNIST.

La figura 4.18 muestra como varía el consumo de memoria GPU de cada framework al variar la cantidad de capas convolucionales. En el caso de Theano y CNTK, dicho consumo cambia en forma cuasi-lineal. Theano es quien, en todo caso estudiado, utiliza la menor cantidad de memoria, de tal forma que su consumo aumenta rápidamente sólo hasta que se triplica la cantidad original de capas, presentando una evolución más moderada cuando el factor de aumento varía entre 3 y 5. CNTK toma el segundo lugar, aumentando en forma más rápida su consumo en comparación con Theano. Finalmente, el consumo de TensorFlow aumenta muy rápidamente al variar el factor de aumento de 1 a 2, para luego mantenerse constante.

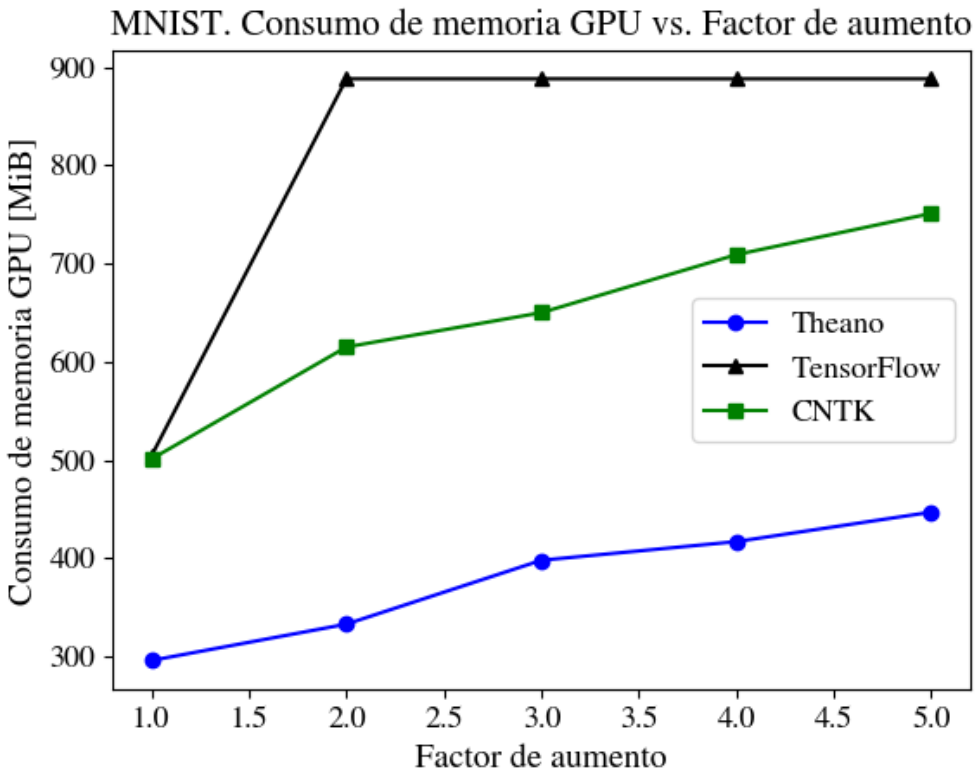


Figura 4.18: Incremento de la cantidad de capas convolucionales en CNN-Softmax y su impacto en el consumo de memoria GPU. Dataset: MNIST.

De acuerdo a la figura 4.19, el consumo en TensorFlow pasa de ser 1,7 veces superior al de Theano (con factor 1) a 2,7 veces superior (con factor 2), para que luego la relación entre ambas cantidades decrezca levemente. En forma análoga, TensorFlow pasa de utilizar prácticamente la misma cantidad de memoria que CNTK (al mantener intacta la cantidad original de capas) a utilizar 1,4 veces más este recurso, de manera que el valor del cociente entre ambas cantidades decrece levemente con factores de aumentos posteriores. Por otra parte, la utilización de memoria GPU en CNTK es entre 1,7 y 1,8 superior a la de Theano.

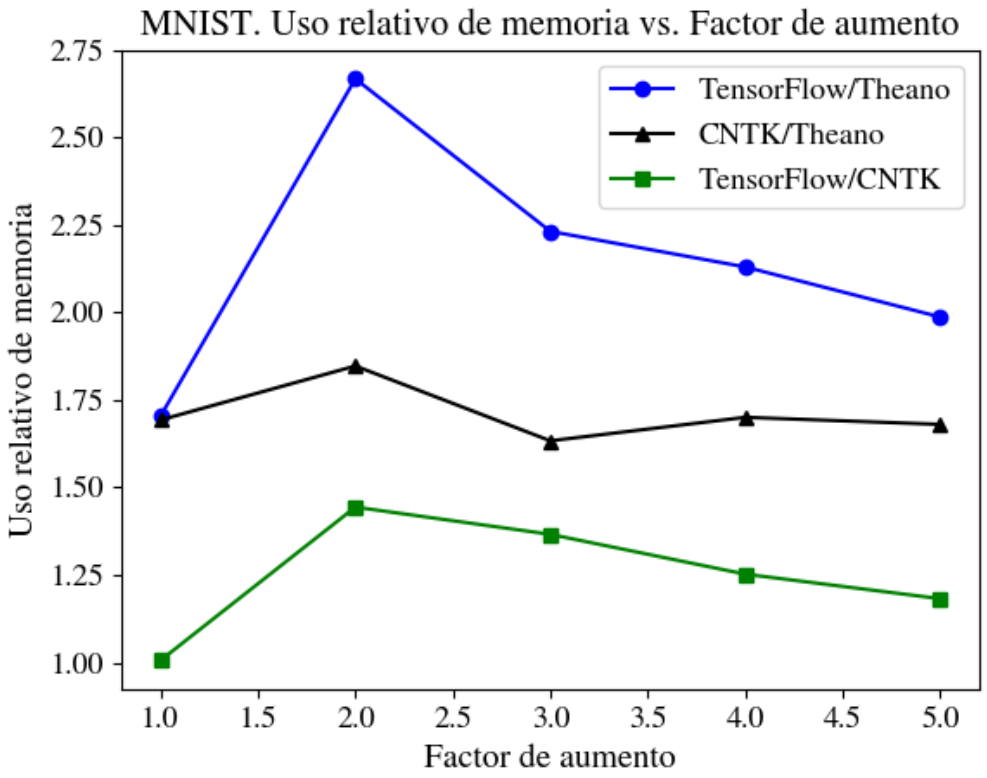


Figura 4.19: Uso relativo de memoria GPU de cada par de frameworks en el entrenamiento de CNN-Softmax como función del factor de aumento. Dataset: MNIST.

4.6.2.1.5. Variación de la cantidad de filtros convolucionales

En esta sección, se varía la cantidad de filtros usados en cada capa convolucional de CNN-Softmax. En cada variación, se reduce / aumenta el número de filtros de las capas originales por un factor de variación múltiplo de 2. Luego, un factor de $\frac{1}{2}$ implica reducir a la mitad la cantidad original de filtros en cada capa. En la tabla 4.11 se muestra la cantidad de filtros considerados en cada variación junto con la cantidad total de parámetros del modelo resultante.

Cuadro 4.11: Cantidad de filtros y parámetros para cada variación estudiada de CNN-Softmax, donde C_i corresponde a la capa convolucional i de la red. Dataset: MNIST.

Factor de variación	**Número de filtros** C_1	**Número de filtros** C_2	**Cantidad total de parámetros**
$\frac{1}{4}$	8	16	136.934
$\frac{1}{2}$	16	32	274.758
1	32	64	569.606
2	64	128	1.236.102
4	128	256	2.876.294

En la figura 4.20 se muestra como cambia el tiempo total de entrenamiento en base al factor de variación. En general, puede decirse que las curvas presentan un comportamiento súper lineal sobre dicho factor. Al reducir la cantidad de filtros, es CNTK quien presenta los tiempos más bajos de entrenamiento. Sin embargo, al aumentar esta cantidad, es TensorFlow quien logra una mayor eficiencia temporal. Una vez más, Theano obtiene los tiempos más altos en todo escenario posible, siendo quien peor escala con el número de filtros.

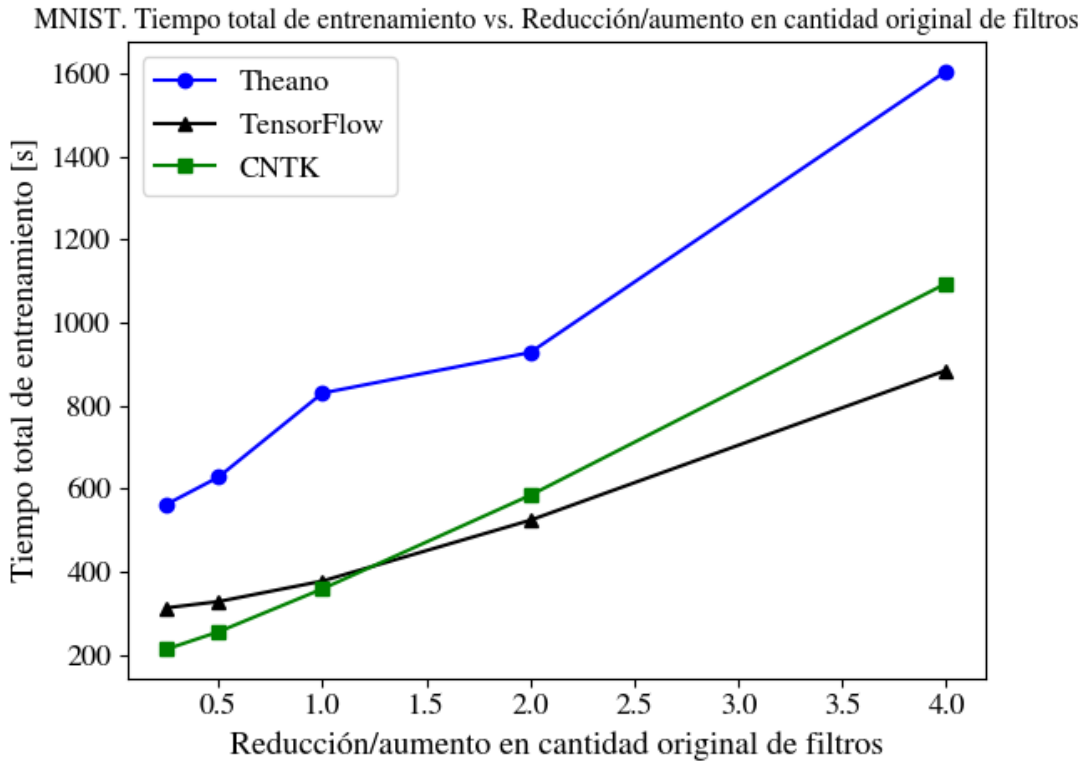


Figura 4.20: Variación de la cantidad de filtros convolucionales en CNN-Softmax y su impacto en el tiempo total de entrenamiento. Dataset: MNIST.

En la figura 4.21 se muestra que TensorFlow es entre 1,8 y 2,2 veces más rápido que Theano, de manera que la diferencia máxima de tiempos entre ambos se da cuando la cantidad de filtros se mantiene intacta y la mínima cuando es duplicada. Por otra parte, el speed-up relativo entre CNTK y Theano es decreciente en todo momento. CNTK es entre 1,5 y 2,6 veces más rápido que Theano, de tal manera que el speed-up es máximo al reducir el número de filtros a la cuarta parte y mínimo al cuadruplicar dicho número. Junto con lo anterior, el speed-up relativo entre TensorFlow y CNTK evoluciona en forma semejante a la función raíz cuadrada, siendo 0,7 su valor mínimo y 1,2 su valor máximo.

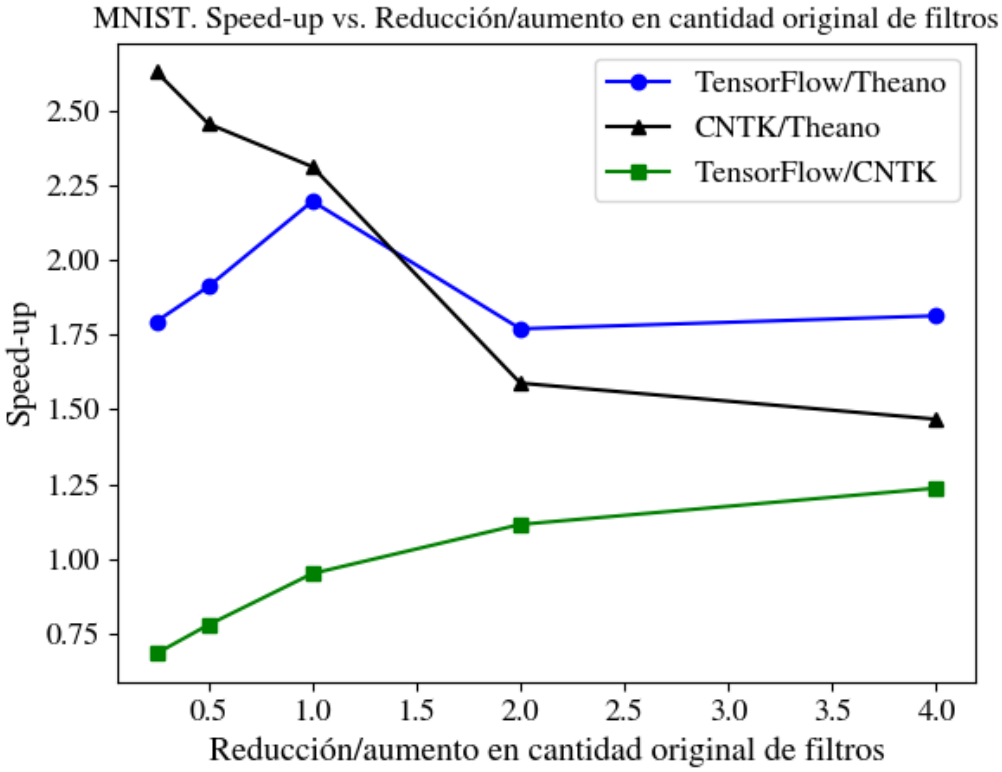


Figura 4.21: Speed-up relativo entre los tiempos de entrenamiento de CNN-Softmax para cada par de frameworks en función de la variación en la cantidad de filtros. Dataset: MNIST.

La figura 4.22 muestra que el consumo de memoria GPU en función de la variación aplicada cambia en forma súper lineal en TensorFlow y CNTK, mientras que en Theano lo hace en forma lineal. Se observa que al reducir el número de filtros, los consumos son muy similares entre los tres frameworks, pero la situación cambia radicalmente al aumentar dicho número, ya que las diferencias entre ellos se acrecientan notablemente.

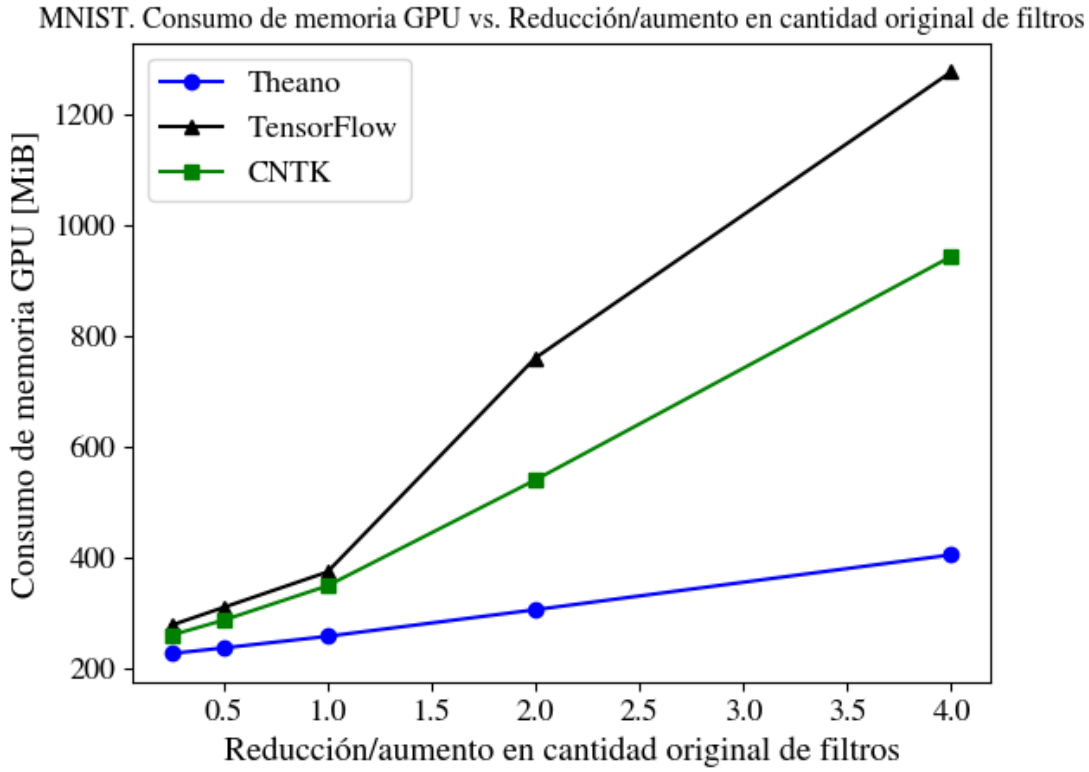


Figura 4.22: Variación de la cantidad de filtros convolucionales de CNN-Softmax y su impacto en el consumo de memoria GPU. Dataset: MNIST.

La figura 4.23 muestra que el uso relativo de memoria GPU entre TensorFlow y Theano crece en todo momento, de tal forma que el primero utiliza entre 1,2 y 3,1 veces más memoria que el segundo. Por otra parte, el consumo relativo entre CNTK y Theano evoluciona en forma lineal, donde CNTK consume entre 1,1 y 2,3 veces más memoria que Theano. La relación entre TensorFlow y CNTK se mantiene prácticamente inalterable al reducir la cantidad inicial de filtros, para luego aumentar levemente al duplicarla. Así, TensorFlow consume entre 1,1 y 1,4 veces más memoria que CNTK.

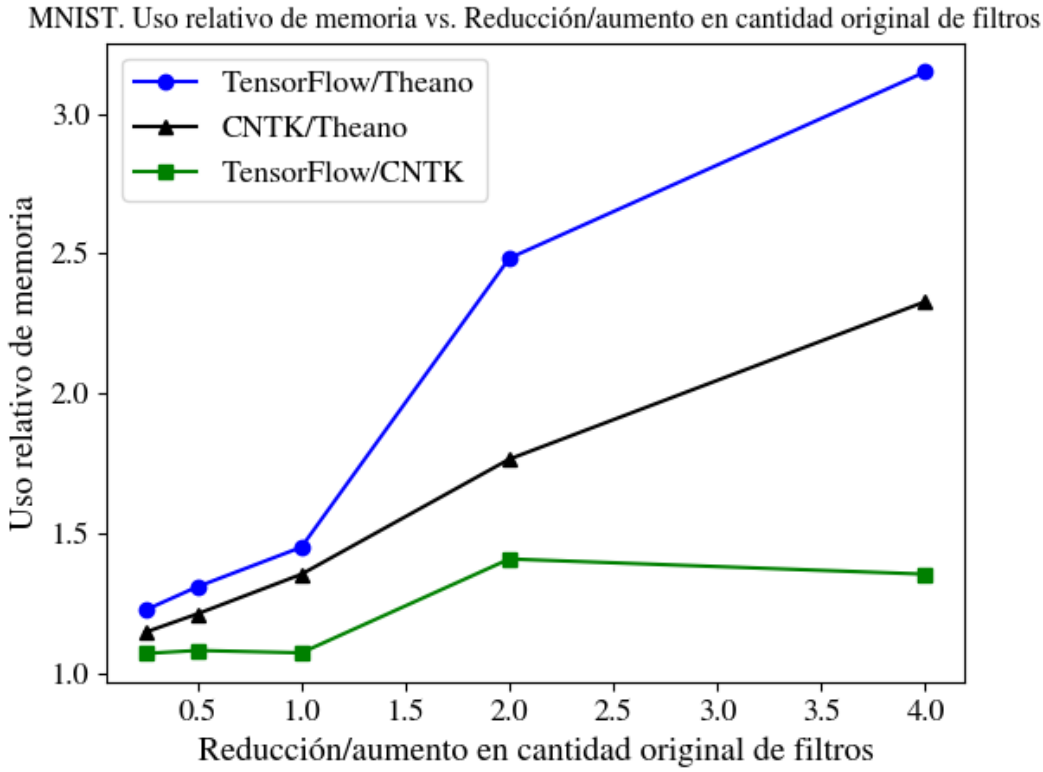


Figura 4.23: Uso relativo de memoria GPU entre cada par de frameworks en el entrenamiento de CNN-Softmax como función de la variación de la cantidad de filtros. Dataset: MNIST.

4.6.2.1.6. Variación de tamaños de filtros convolucionales

Cuadro 4.12: Disminución / incremento de los tamaños de los filtros de las capas convolucionales de CNN-Softmax, junto con la cantidad de parámetros resultantes. C_i representa la capa convolucional i. Dataset: MNIST.

Cambio en tamaños de filtros	**Tamaños filtros** C_1	**Tamaños filtros** C_2	**Cantidad total de parámetros**
-3	2 x 2	2 x 2	1.165.926
-2	3 x 3	3 x 3	824.326
-1	4 x 4	4 x 4	550.886
0	5 x 5	5 x 5	569.606
1	6 x 6	6 x 6	368.486
2	7 x 7	7 x 7	235.526

La figura 4.24 muestra que el tiempo total de entrenamiento de CNN-Softmax decrece linealmente al aumentar los tamaños de los filtros, para todo framework. Además, Theano presenta un rendimiento totalmente inferior al de las otras herramientas, lo que se manifiesta en lo rápido que aumenta el tiempo medido al momento de reducir las dimensiones de los filtros. TensorFlow, por otra parte, no experimenta grandes variaciones en los tiempos empleados. CNTK experimenta un rendimiento similar al de TensorFlow, pero el decrecimiento del tiempo se da en forma más marcada. Se puede observar que al disminuir los tamaños de los filtros, TensorFlow presenta un mejor rendimiento respecto a CNTK, pero al aumentar los tamaños, los papeles se invierten.

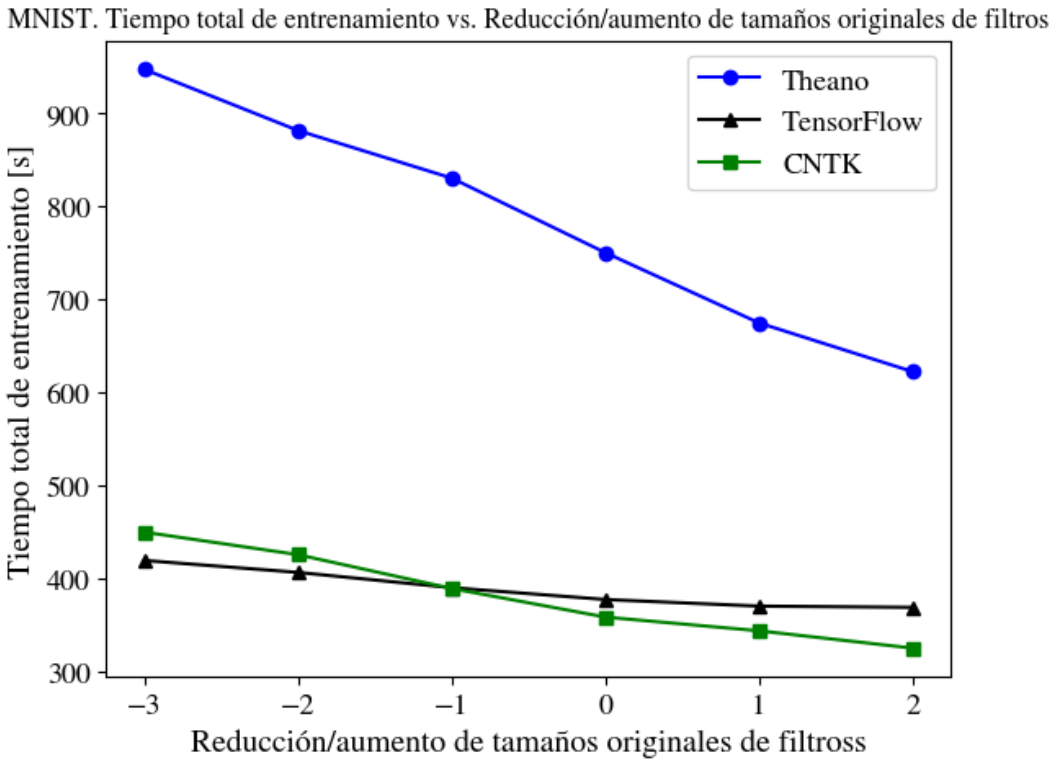


Figura 4.24: Variación de los tamaños de los filtros convolucionales de CNN-Softmax y su impacto en el tiempo total de entrenamiento. Dataset: MNIST.

En la figura 4.25 se muestra que TensorFlow es entre 1,7 y 2,3 veces más rápido que Theano al variar las dimensiones de los filtros. También, CNTK es entre 1,9 y 2,1 veces más rápido que Theano; la máxima diferencia entre los rendimientos de ambos se presenta al reducir los tamaños en una unidad, mientras que la mínima al aumentar dichos tamaños en dos unidades. Por otra parte, el speed-up relativo entre TensorFlow y CNTK varía entre 0,9 y 1,1.

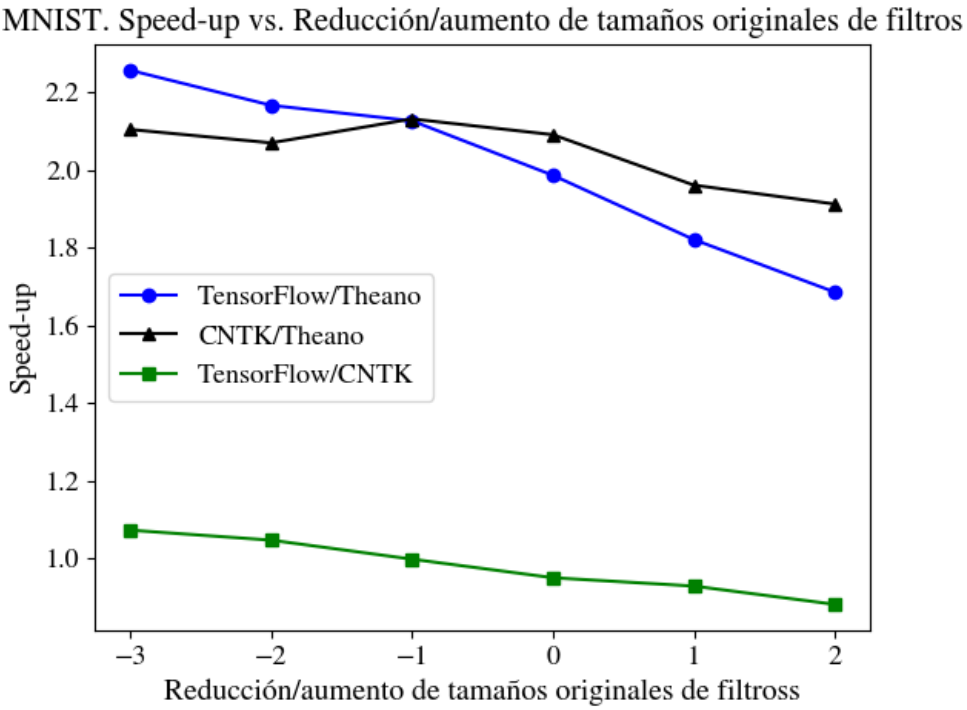


Figura 4.25: Speed-up relativo entre los tiempos de entrenamiento de CNN-Softmax para cada par de frameworks en función de la variación en la cantidad de filtros. Dataset: MNIST.

En la figura 4.27 se muestra, que en todo escenario, TensorFlow emplea el mayor uso de memoria GPU. El comportamiento del framework es, a lo menos, curioso, ya que en los escenarios extremos (esto es, reducción de las dimensiones en tres unidades y aumento de las mismas en dos unidades) el consumo de memoria es prácticamente idéntico, mientras que en los experimentos restantes, el consumo es notablemente inferior y se mantiene prácticamente constante en todos ellos. Por otra parte, la utilización de la GPU decrece lentamente en CNTK mientras más grandes son los tamaños de los filtros resultantes. En Theano, la utilización del recurso también decrece, pero en forma lineal.

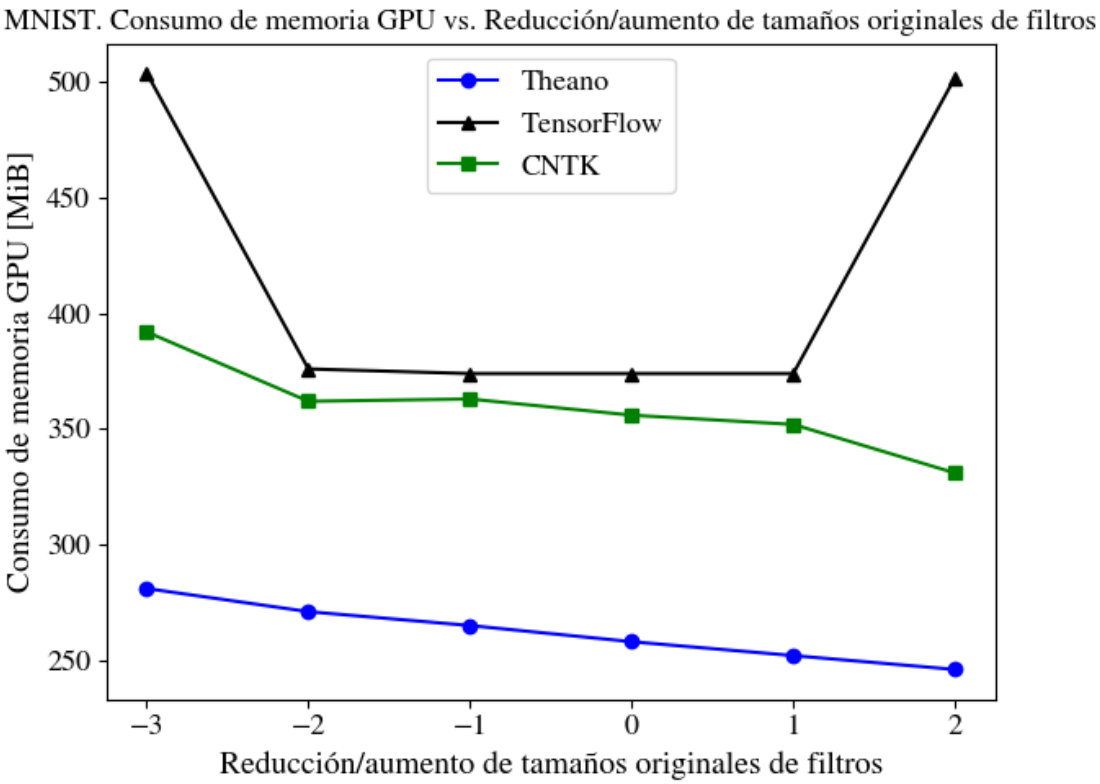


Figura 4.26: Variación de los tamaños de los filtros convolucionales de CNN-Softmax y su impacto en el consumo de memoria GPU. Dataset: MNIST.

De acuerdo a la figura 4.26, TensorFlow consume entre 1,4 y 2 veces más memoria que Theano; la diferencia mínima entre los frameworks se da al reducir los tamaños de los filtros en dos unidades, mientras que el valor máximo al aumentar dichos tamaños en dos unidades. El uso relativo de memoria entre CNTK y Theano, en cambio, varía entre 1,3 y 1,4, donde el valor mínimo también se da al reducir las dimensiones de los filtros en dos unidades y el máximo al aumentarlas en dos unidades. Por último, el cociente entre los consumos de TensorFlow y CNTK varía entre 1 y 1,3; el valor mínimo es alcanzado al aplicar una reducción de una unidad, mientras que el máximo es obtenido con un aumento de dos unidades.

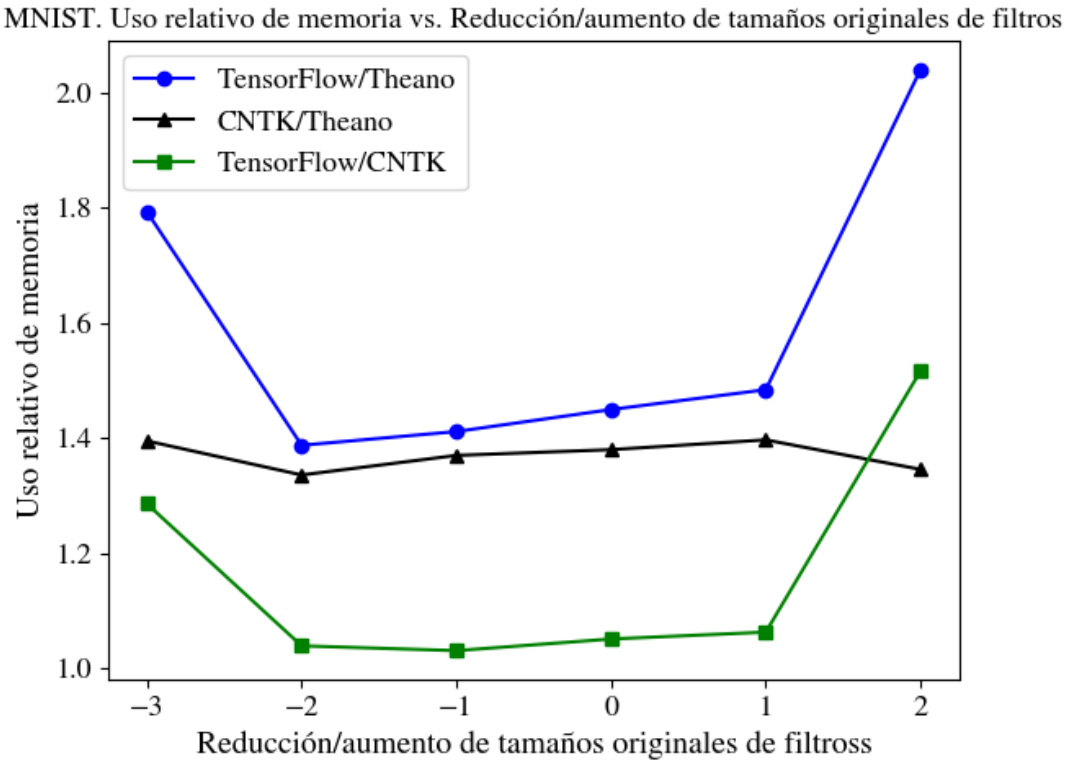


Figura 4.27: Razón entre los consumos de memoria GPU para cada par de frameworks en el entrenamiento de CNN-Softmax como función de la variación en el tamaño de los filtros. Dataset: MNIST.

4.6.2.2. CIFAR-10

4.6.2.2.1. Arquitectura maestra

Se analiza el modelo All-CNN-C propuesto en [43]. Su estructura es detallada a continuación, en el orden de aparición de las capas:

- Dos capas convolucionales consecutivas con 96 filtros de tamaño 3 x 3, cada una.
- Una capa convolucional con 96 filtros de tamaño 3 x 3 y stride 2.
- Dos capas convolucionales consecutivas con 192 filtros de tamaño 3 x 3, cada una.
- Una capa convolucional con 192 filtros de tamaño 3 x 3 y stride 2.

- Una capa convolucional con 192 filtros de tamaño 3 x 3, cada uno.
- Una capa convolucional con 192 filtros de tamaño 1 x 1, cada uno.
- Una capa convolucional con 10 filtros de tamaño 1 x 1, cada uno.
- Una capa de global average pooling.
- Capa totalmente conectada, con 10 neuronas y función de activación softmax.

La arquitectura está integrada por 1.369.738 parámetros entrenables. Además, el error de prueba obtenido por los autores fue de un 9,08 %.

Las imágenes son preprocesadas aplicando GCN en primer lugar y luego ZCA whitening. Para todas las capas ocultas se utiliza ReLU como función de activación. El modelo es entrenado con SGD durante 350 epochs. No se indica el tamaño de mini-batch considerado, por lo que es fijado en 100, que fue el tamaño escogido en la arquitectura básica. Tampoco se especifica cual es la tasa de aprendizaje inicial, aunque se menciona que se experimentó con los valores 0,25, 0,1, 0,05 y 0,01. Toda tasa es multiplicada por un factor de 0,1 después de 200, 250 y 300 epochs. Se usó momentum fijo de 0,9. Para regularizar el modelo, se aplica dropout de 0,2 sobre la capa de entrada y de 0,5 para cada capa convolucional de stride 2. El modelo también es regularizado con weight decay de 0,001.

Para determinar la mejor tasa de aprendizaje inicial posible, se construyó un conjunto de validación a partir de 5.000 ejemplos escogidos aleatoriamente desde el conjunto de entrenamiento. Gracias a este procedimiento, se determinó que el mejor valor inicial para el hiper parámetro es de 0,05, con el que se obtuvieron los siguientes resultados:

- Error de entrenamiento: 9,18 %
- Error de validación: 12,24 %

En la tabla 4.13 se muestran los resultados de entrenar la arquitectura sobre el conjunto de entrenamiento completo con esta tasa de aprendizaje. Nuevamente, los tiempos totales de TensorFlow y CNTK son muy similares entre sí, de tal forma que la razón entre ambas cantidades es de 1,01. Dichos tiempos, además, corresponden a la mitad del que es empleado por Theano. En este contexto, el consumo de memoria GPU por parte de TensorFlow es 2,1 y 2,9 veces superior al de Theano y CNTK, respectivamente. Además, la razón entre los consumos de CNTK y Theano es de 1,4. Los errores de

entrenamiento y de prueba son similares, si bien las diferencias entre ellos aumentan levemente en comparación con MNIST. En entrenamiento, la diferencia más grande se da entre TensorFlow y Theano (0,18 %), mientras que en la fase de evaluación, se da entre TensorFlow y CNTK (0,33 %).

Cuadro 4.13: Rendimiento sobre arquitectura All-CNN-C. Dataset: CIFAR-10.

Framework	**Tiempo de entrenamiento total [s]**	**Consumo de memoria GPU [MiB]**	**Error de entrenamiento [%]**	**Error de prueba [%]**
Theano	20098,3	706	9,08	12,93
TensorFlow	10006,9	2070	9,26	13,12
CNTK	10060,1	992	9,14	12,79

4.6.2.2.2. Arquitectura básica

Se estudia el modelo CNN + ReLU propuesto en [44], el cual se describe a continuación, de acuerdo al orden en que aparecen sus capas:

- Capa convolucional con 96 filtros de 5 x 5 y padding de 2.
- Capa de max pooling con pooling size de 3 x 3 y stride de 2.
- Capa convolucional con 128 filtros de 5 x 5 y padding de 2.
- Capa de average pooling con pooling size de 3 x 3 y stride de 2.
- Capa convolucional con 256 filtros de 5 x 5 y padding de 2.
- Capa de average pooling con pooling size de 3 x 3 y stride de 2.
- Dos capas totalmente conectadas consecutivas de 2048 neuronas y dropout de 0,5, cada una.

- Capa totalmente conectada integrada por 10 neuronas y función de activación softmax (capa de salida).

Cada capa convolucional considera un stride de 1. El error de prueba obtenido por los autores es de un $12,56\,\% \pm 0,26$, como promedio de cinco experimentos, cada uno con una inicialización aleatoria diferente para los parámetros de la red.

Para el pre-procesamiento de las imágenes, se calcula el valor promedio de cada pixel, considerando el conjunto de entrenamiento completo y cada canal por separado. Luego, dichos valores son restados de cada imagen. Para toda capa oculta, se utiliza ReLU como función de activación. Se aplica dropout de 0,25 previo a cada capa de pooling. Además, se aplica dropout de 0,25, 0,25 y 0,5 después de cada capa de pooling, respectivamente. El modelo es entrenado durante 150 epochs por medio de SGD con tamaño de mini-batch 100. La tasa de aprendizaje cambia en epochs específicos: Es de 0,01 desde el epoch 1, es de 0,001 desde el epoch 100, es de 0,0001 desde el epoch 134 y es de 0,00001 desde el epoch 142. Se aplica weight decay de 0,001. Además, se utiliza un momentum inicial de 0,5 que aumenta hasta un valor máximo de 0,9 de acuerdo a la siguiente ecuación, que corresponde a una versión modificada de la original:

$$m_i = m_{max} - (m_{max} - m_0) \cdot \exp(-0,06i) \tag{4.4}$$

Donde m_i es el valor del momentum en el epoch i, m_{max} es su valor máximo y m_0 es su valor inicial. La ecuación original ha sido modificada debido a que esta actualiza el momentum en base al número de **iteración** actual, que, como es sabido, no es lo mismo que número de epoch. Sin embargo, el entrenamiento de un modelo en Keras sólo puede desarrollarse en base a epochs, no iteraciones, llevando a que fuese necesario encontrar una ecuación equivalente que actualizara el momentum de acuerdo al número de epoch actual y por tanto fuese compatible con Keras.

En la tabla 4.14 se muestran los resultados obtenidos para esta configuración. TensorFlow consigue el tiempo total de entrenamiento más bajo, seguido por CNTK y Theano. En esta ocasión, CNTK emplea 1,2 veces más tiempo que TensorFlow. Así, se presenta una mayor diferencia entre ambos con respecto a casos anteriores, si bien los dos vuelven a tener un rendimiento ampliamente superior al de Theano, ya que este requiere de 1,9 y 1,2 veces más tiempo, respectivamente, para entrenar el modelo. Por otra parte, CNTK es quien consigue la mayor utilización de la GPU, consumiendo

1,2 y 2,3 veces más memoria que TensorFlow y Theano, respectivamente. Además, el consumo de TensorFlow es 1,9 veces superior al de Theano. A pesar de que las diferencias entre errores de entrenamiento son más marcadas con respecto a casos anteriores (2,91 % entre TensorFlow y CNTK), los errores de prueba continúan siendo similares, de tal manera que la diferencia más grande, que se da entre los mismos frameworks anteriores, es de un 0,7 %.

Cuadro 4.14: Rendimiento sobre arquitectura CNN + ReLU. Dataset: CIFAR-10.

Framework	**Tiempo de entrenamiento total [s]**	**Consumo de memoria GPU [MiB]**	**Error de entrenamiento [%]**	**Error de prueba [%]**
Theano	3906,9	537	12,71	16,07
TensorFlow	2008,5	1014	15,58	16,73
CNTK	2317,9	1203	12,67	16,03

4.6.2.2.3. Variación de cantidad de ejemplos de entrenamiento

Al igual que en la sección 4.6.2.1.3, se estudia el impacto que tiene variar la cantidad de ejemplos utilizados en el entrenamiento de CNN + ReLU sobre las métricas de rendimiento previamente definidas. En la tabla 4.15 se indican las fracciones de datos consideradas.

Cuadro 4.15: Fracciones de ejemplos usadas para el entrenamiento de CNN + ReLU. Dataset: CIFAR-10.

Fracción de ejemplos [%]	**Número de ejemplos**
33	16.500
40	20.000
57	28.500
64	32.000
83	41.500
100	60.000

En la figura 4.28 es posible apreciar como varía el tiempo total de entrenamiento de CNN + ReLU en base al porcentaje de ejemplos considerado. Al igual que en el caso de MNIST, Theano presenta el peor rendimiento de todos. Sin embargo, ahora es TensorFlow quien consigue los menores tiempos de entrenamiento en todos los casos. Si bien TensorFlow y CNTK presentan rendimientos comparables para subconjuntos pequeños de ejemplos (33 % y 40 % del total), las diferencias entre ambos comienzan a ser mayores a medida que la fracción de datos disponible aumenta.

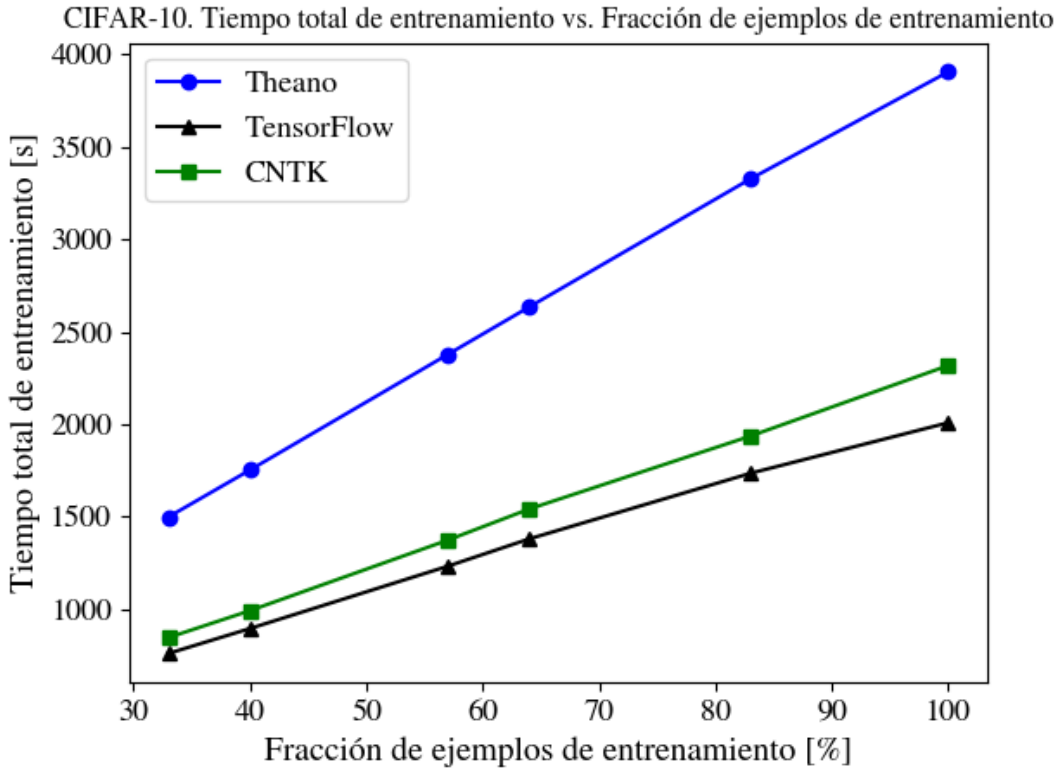


Figura 4.28: Variación de la fracción de ejemplos provistos para el entrenamiento de CNN + ReLU y su impacto en el tiempo total de entrenamiento. Dataset: CIFAR-10.

La figura 4.29 permite observar el cambio que experimenta la cantidad promedio de imágenes procesadas por CNN + ReLU al variar la fracción de ejemplos disponibles. Ningún framework logra mejorar de sobremanera su capacidad de procesar imágenes al aumentar la disponibilidad de estas. TensorFlow alcanza un rendimiento levemente superior al de CNTK en todo escenario.

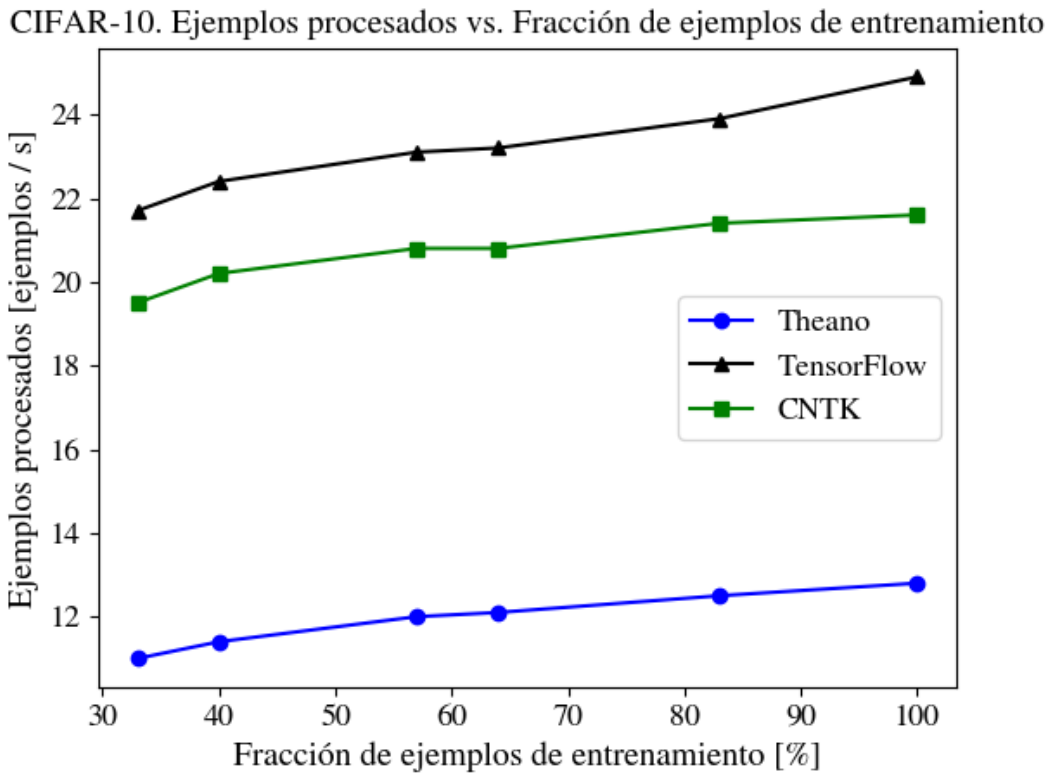


Figura 4.29: Variación de la fracción de ejemplos provistos para el entrenamiento de CNN + ReLU y su impacto en la cantidad promedio de ejemplos de entrenamiento procesados por segundo. Dataset: CIFAR-10.

La figura 4.30 muestra que los tiempos de TensorFlow son entre 1,1 y 1,2 veces menores que los de CNTK, lo que confirma que dichos tiempos puedan considerarse como cercanos. Por otra parte, CNTK presenta tiempos entre 1,7 y 1,8 veces más bajos que los de Theano, mientras que TensorFlow obtiene tiempos entre 1,9 y 2 veces menores que los de Theano, por lo que puede decirse que las proporciones tienden a mantenerse entre frameworks.

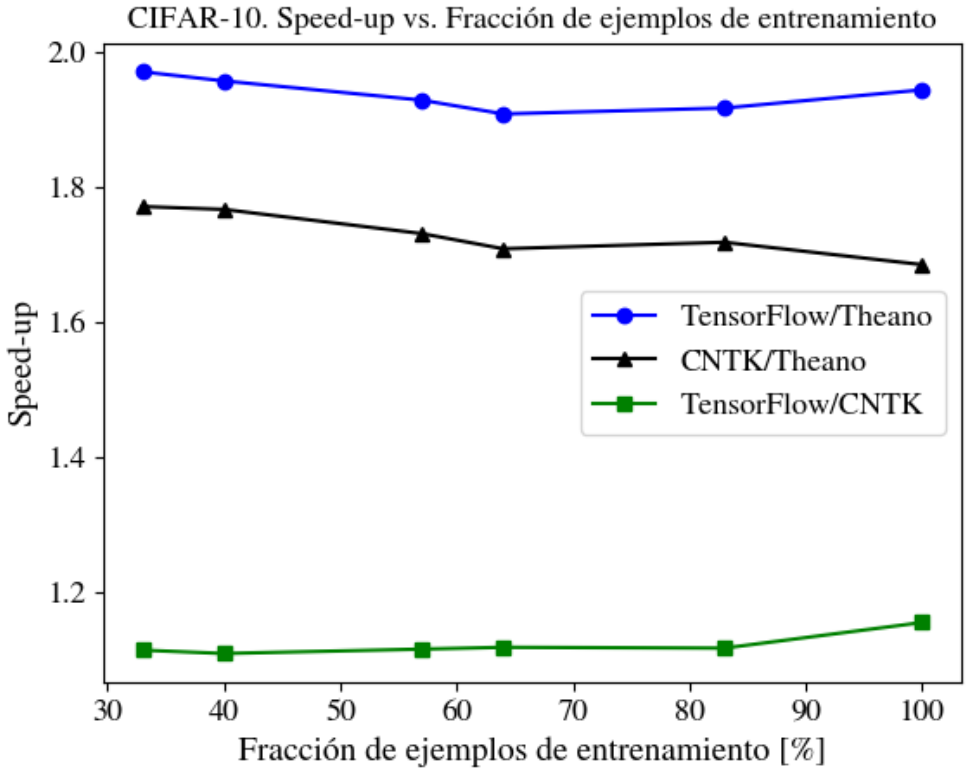


Figura 4.30: Speed-up relativo entre los tiempos de entrenamiento de CNN + ReLU para cada par de frameworks en función de la fracción de ejemplos provistos. Dataset: CIFAR-10.

4.6.2.2.4. Variación de la cantidad de capas convolucionales

Así como en la sección 4.6.2.1.4, cada capa convolucional de CNN + ReLU es replicada entre 2 y 5 veces para analizar el impacto que tiene el aumento en la profundidad de la red sobre el rendimiento de cada framework. La tabla 4.16 muestra las características de las configuraciones estudiadas. Notar que ha sido necesario aplicar padding sobre los inputs recibidos por las capas convolucionales con el fin de evitar la reducción prematura de sus dimensiones.

Cuadro 4.16: Aumento de la cantidad de capas convolucionales en CNN + ReLU, junto con el número total de parámetros del modelo resultante. Dataset: CIFAR-10.

Factor de aumento	**Cantidad total de capas convolucionales**	**Cantidad total de parámetros**
1	3	10.071.562
2	6	12.350.442
3	9	14.629.322
4	12	16.908.202
5	15	19.187.082

Tal y como muestra la figura 4.31, el tiempo total de entrenamiento vuelve a cambiar en forma lineal con el factor de aumento, para todo framework. Nuevamente, Theano presenta el peor rendimiento de todos. CNTK y TensorFlow presentan rendimientos comparables, pero que se distancian levemente con el aumento de la cantidad de capas convolucionales. Sin embargo, a diferencia de como ocurrió con MNIST, es CNTK quien lleva la ventaja en todo momento, excepto cuando la cantidad original de capas se mantiene inalterable.

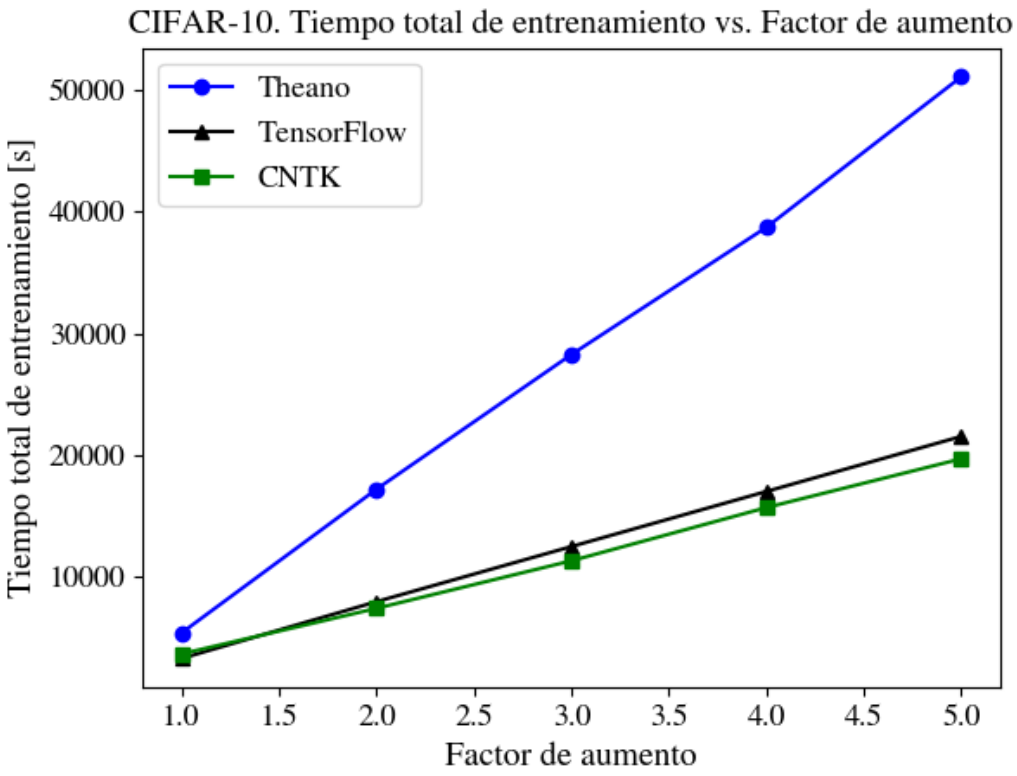


Figura 4.31: Incremento de la cantidad de capas convolucionales en CNN + ReLU y su impacto en el tiempo total de entrenamiento. Dataset: CIFAR-10.

De acuerdo a la figura 4.32, CNTK resulta ser 1,1 veces más rápido que TensorFlow en todo escenario, excepto al conservar la cantidad original de capas convolucionales, donde TensorFlow es 1,1 veces más rápido que CNTK. Por otra parte, los tiempos totales de TensorFlow son entre 1,6 y 2,3 veces más bajos que los de Theano, mientras que los de CNTK son entre 1,5 y 2,6 veces menores que los de Theano.

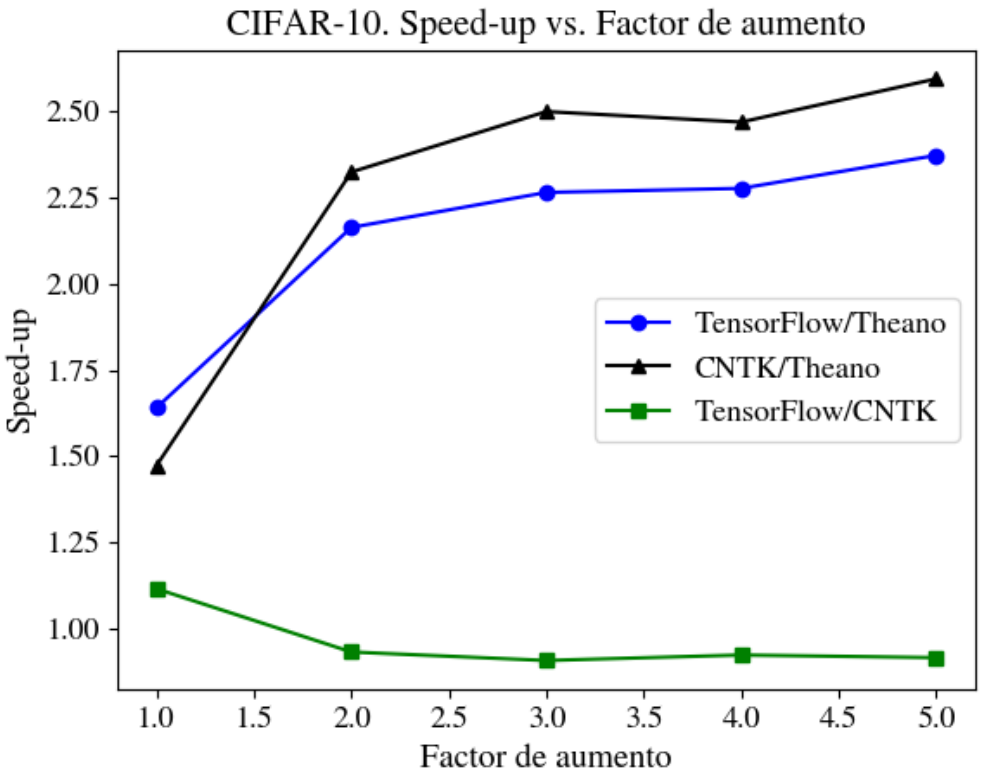


Figura 4.32: Speed-up relativo entre los tiempos de entrenamiento de CNN + ReLU para cada par de frameworks en función del factor de aumento. Dataset: CIFAR-10.

La figura 4.33 muestra que el consumo de memoria GPU aumenta en forma lineal en Theano y CNTK. Se observa, además, que tal aumento se da en forma más rápida en comparación a MNIST. Por otro lado, TensorFlow muestra el mismo comportamiento que en el dataset anterior: La utilización de memoria crece linealmente hasta duplicar la cantidad original de capas convolucionales, para luego mantener un consumo constante en los experimentos restantes. Notar, además, que al conservar la cantidad original de capas y también al quintuplicar dicha cantidad, CNTK lleva a cabo una mayor explotación de la GPU en comparación con TensorFlow, situación que no se presenta con otros factores de aumento.

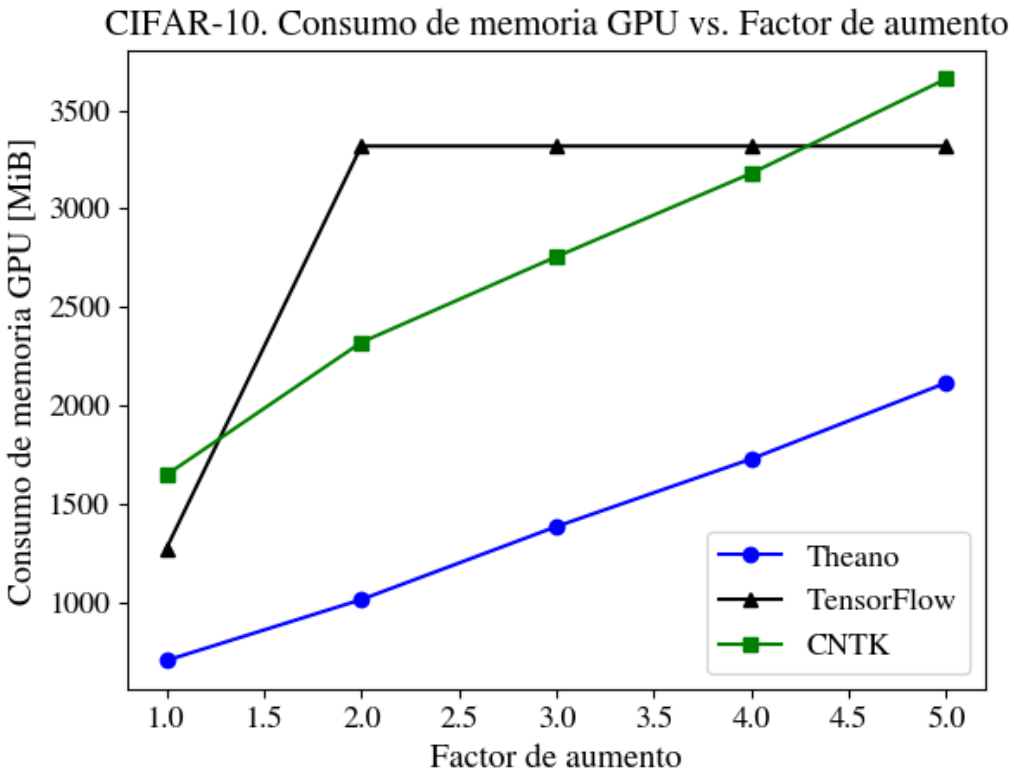


Figura 4.33: Incremento de la cantidad de capas convolucionales en CNN + ReLU y su impacto en el consumo de memoria GPU. Dataset: CIFAR-10.

La figura 4.34 expone que el cociente entre los consumos de TensorFlow y Theano alcanza su peak en 3,2 para un factor de aumento de 2, para luego decrecer linealmente hasta alcanzar el valor 1,6 cuando se quintuplica el número de capas. Por otro lado, la relación TensorFlow/CNTK alcanza un valor máximo de 1,4, también con un factor de 2, mientras que su valor mínimo es de 0,77 al mantener intacta la cantidad inicial de capas. Finalmente, la relación CNTK/Theano es decreciente en todo momento y varía entre 1,7 y 2,3.

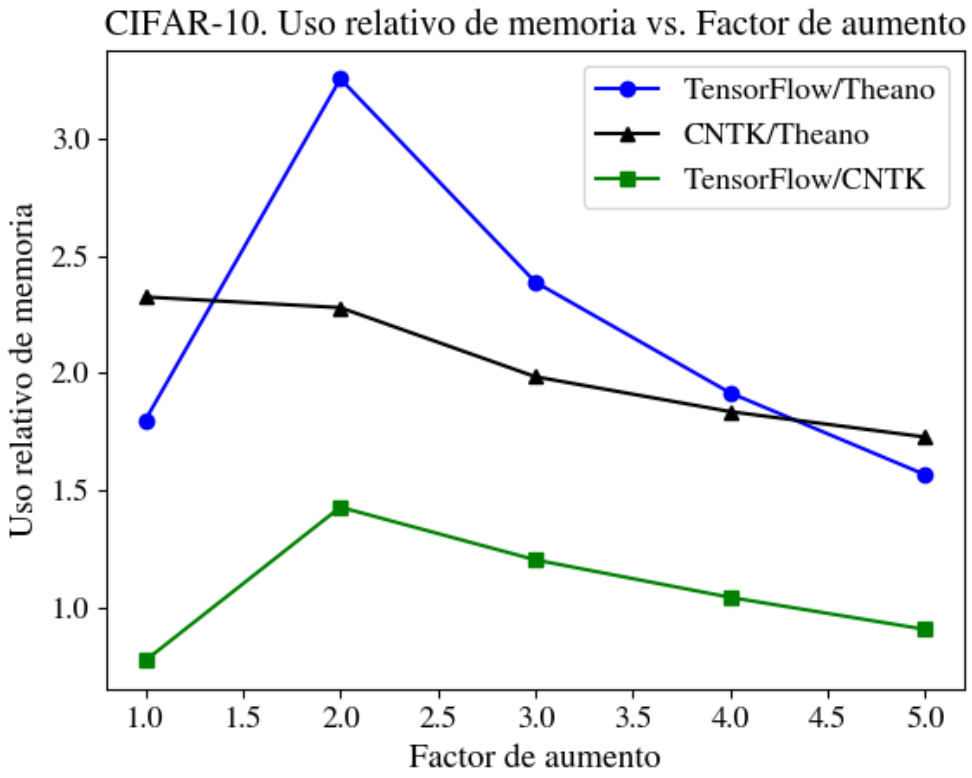


Figura 4.34: Uso relativo de memoria GPU entre cada par de frameworks en el entrenamiento de CNN + ReLU como función del factor de aumento. Dataset: CIFAR-10.

4.6.2.2.5. Variación en la cantidad de filtros convolucionales

De forma análoga a la sección 4.6.2.1.5, se altera la cantidad de filtros utilizados en cada capa convolucional de CNN + ReLU. La tabla 4.17 muestra la cantidad de filtros presentes en las capas convolucionales de cada configuración estudiada junto con el número total de parámetros asociados.

Cuadro 4.17: Cantidad de filtros y parámetros para cada variación estudiada de CNN + ReLU, donde C_i corresponde a la capa convolucional i del modelo. Dataset: CIFAR-10.

Factor de variación	**Número de filtros C1**	**Número de filtros C2**	**Número de filtros C3**	**Cantidad total de parámetros**
$\frac{1}{4}$	24	32	64	4.422.282
$\frac{1}{2}$	48	64	128	4.766.474
1	96	128	256	5.877.258
2	192	256	512	9.788.426
4	384	512	1024	24.369.162

La figura 4.35 muestra que al reducir la cantidad de filtros convolucionales a la cuarta parte en cada capa, el tiempo total de entrenamiento es prácticamente idéntico entre frameworks. Sin embargo, al aumentar la cantidad de filtros en la red, las diferencias comienzan a acrecentarse. Theano es el menos escalable. Por otro lado, TensorFlow y CNTK presentan rendimientos muy similares, presentándose una diferencia levemente mayor entre ambos sólo al cuadruplicar los filtros. Además, los incrementos de tiempo son muchos menos bruscos en comparación a Theano. Al reducir la cantidad de filtros, es TensorFlow quien consigue los tiempos más bajos, mientras que al aumentar dicha cantidad, CNTK logra los tiempos más reducidos. Notar que las curvas siguen un comportamiento súper lineal.

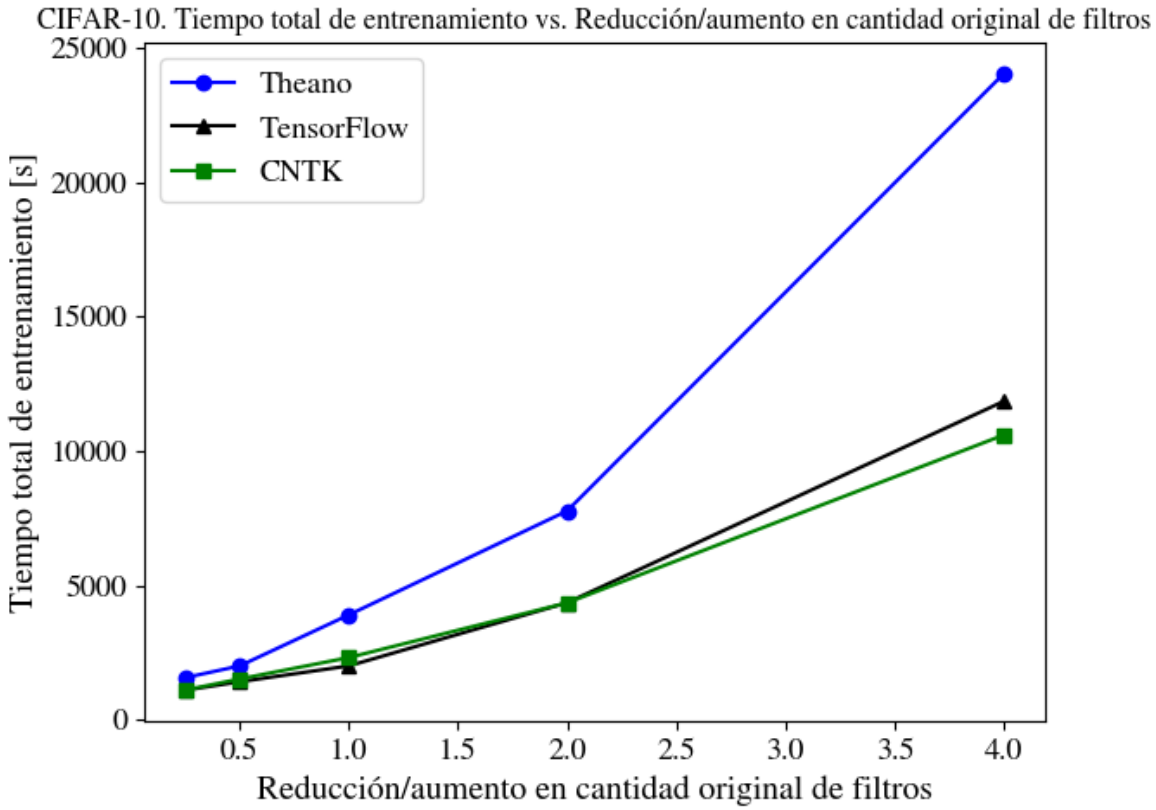


Figura 4.35: Variación de la cantidad de filtros convolucionales en CNN + ReLU y su impacto en el tiempo total de entrenamiento. Dataset: CIFAR-10.

La pobre escalabilidad de Theano se refleja en la figura 4.36, donde se aprecia que el framework es entre 1,4 y 2 veces más lento que TensorFlow y entre 1,3 y 2,3 veces más lento que CNTK. En ambos casos, el speed-up relativo alcanza su valor mínimo con la mitad del número original de filtros y su valor máximo con el cuádruple de dicho número. Por otro lado, el speed-up relativo entre los tiempos de TensorFlow y CNTK aumenta hasta que el factor de variación es 1, para luego decrecer hasta alcanzar su valor más bajo. El cociente máximo es de 1,2, mientras que el mínimo de 0,9.

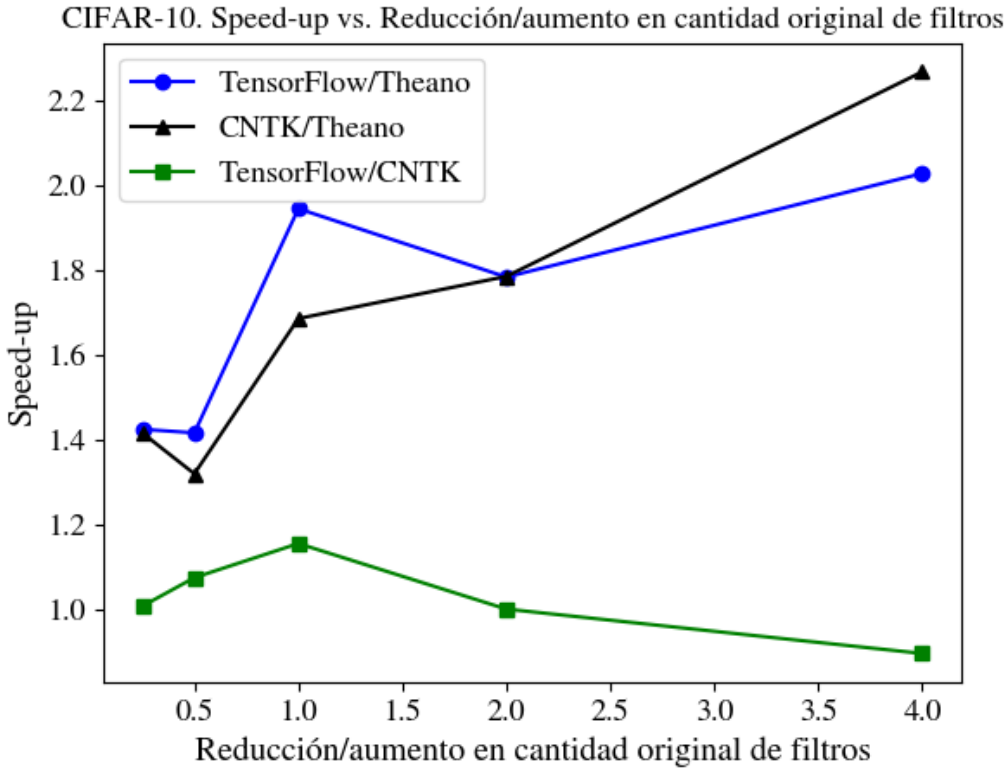


Figura 4.36: Speed-up relativo entre los tiempos de entrenamiento de CNN + ReLU de cada par de frameworks en función de la variación en la cantidad de filtros. Dataset: CIFAR-10.

La figura 4.37 muestra que el consumo de memoria GPU de TensorFlow evoluciona en forma súper lineal en base al factor de variación, mientras que en Theano y CNTK el cambio se manifiesta en forma lineal. Al igual que en el caso de MNIST, los consumos son muy similares entre los tres frameworks al reducir la cantidad original de filtros, pero se hacen muy disímiles al aumentarla. Más aún, es posible observar que en el caso de reducción, CNTK utiliza cantidades de memoria levemente superiores a las de TensorFlow.

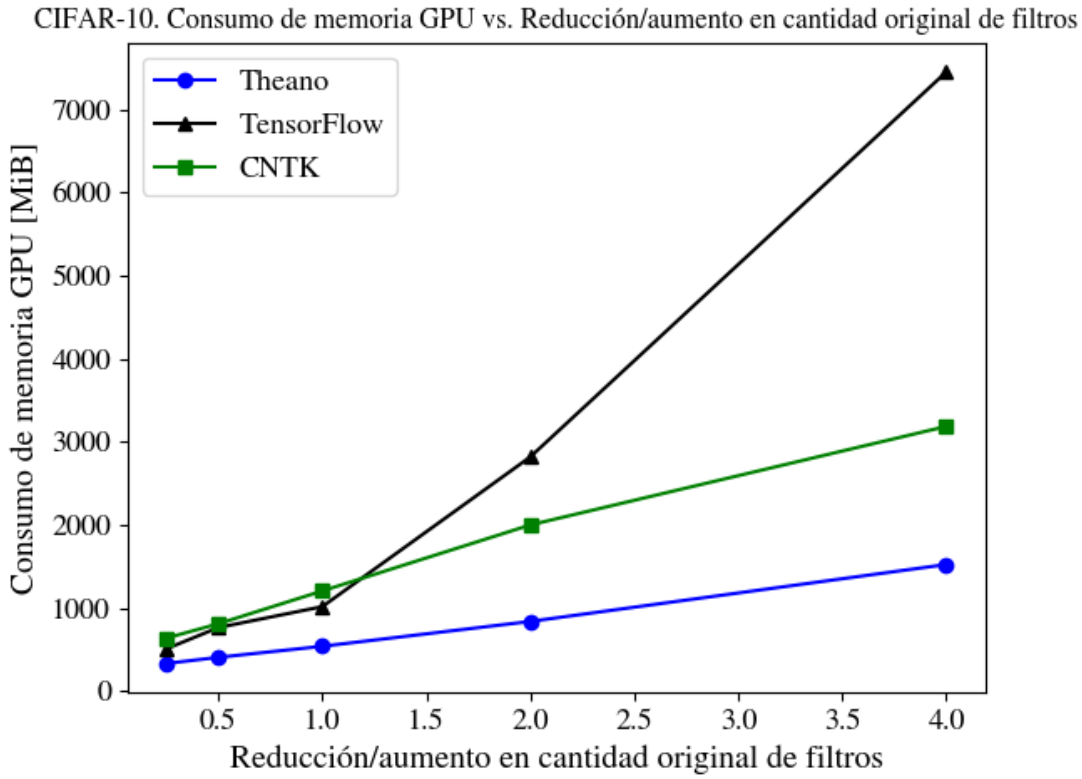


Figura 4.37: Variación de la cantidad de filtros convolucionales de CNN + ReLU y su impacto en el consumo de memoria GPU. Dataset: CIFAR-10.

La figura 4.38 expone que el uso relativo de memoria entre TensorFlow y Theano tiende a ser creciente, alcanzando el valor mínimo de 1,5 con la cantidad original de filtros y el valor máximo de 4,9 al cuadruplicar dicha cantidad. Por otro lado, el consumo relativo entre CNTK y Theano aumenta lentamente para luego decrecer en el último experimento; su valor mínimo es de 1,9 al reducir el número de filtros a un cuarto del original, mientras que el valor máximo de 2,4 es obtenido con el doble de dicho número. Finalmente, la razón entre TensorFlow y CNTK tiende a crecer rápidamente al aumentar el factor de variación, de manera que el primero consume entre 0,8 y 2,3 veces la cantidad de memoria que utiliza el segundo, con factores de variación $\frac{1}{4}$ y 4, respectivamente.

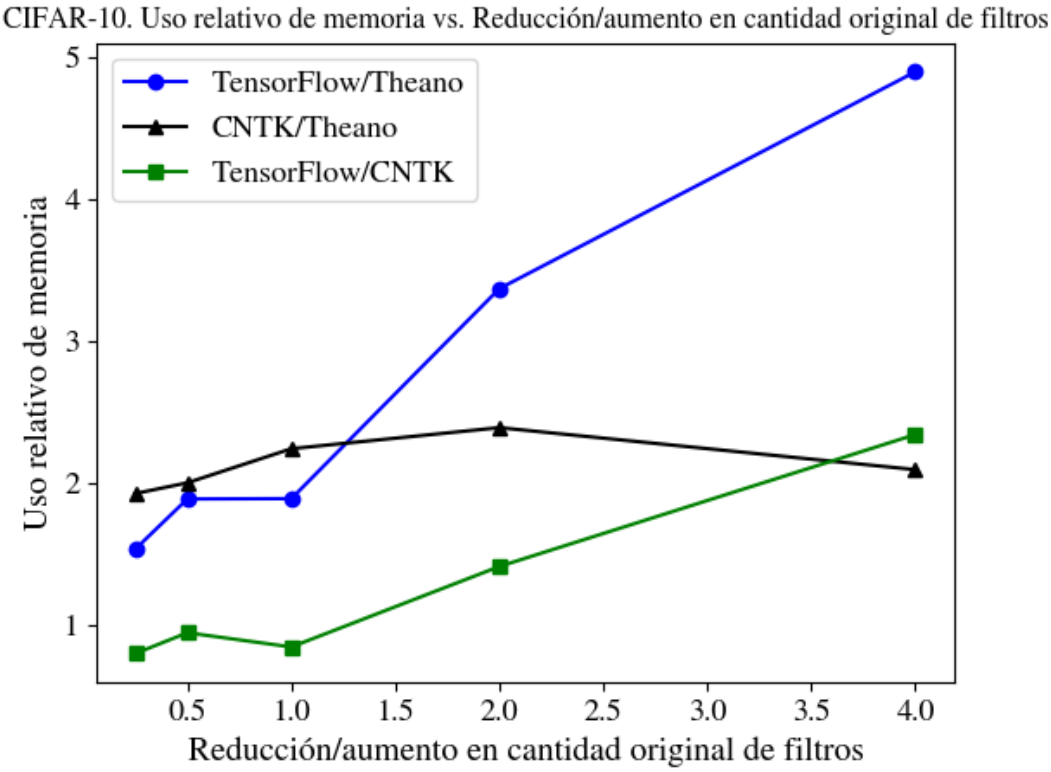


Figura 4.38: Razón entre los consumos de memoria GPU para cada par de frameworks en el entrenamiento de CNN-Softmax como función de la variación en la cantidad de filtros. Dataset: CIFAR-10.

4.6.2.2.6. Variación de tamaños de filtros convolucionales

Tal como en la sección 4.6.2.1.6, se procede a disminuir / aumentar el ancho y el alto de los filtros de cada capa convolucional existente en CNN + ReLU. La tabla 4.18 resume cada una de las configuraciones estudiadas. Nuevamente, la cantidad de parámetros no disminuye monótonamente. En esta ocasión, el fenómeno se presenta cuando la tercera capa de pooling produce outputs de dimensiones idénticas entre dos arquitecturas consecutivas. Notar, además, que fue necesario duplicar las dimensiones originales de las imágenes para evitar su reducción prematura al aumentar los tamaños de los filtros.

Cuadro 4.18: Disminución / incremento en el tamaño de los filtros de las capas convolucionales de CNN + ReLU, junto con la cantidad de parámetros resultantes. C_i representa la capa convolucional i. Dataset: CIFAR-10.

Cambio en tamaño de filtros	**Tamaño filtros C1**	**Tamaño filtros C2**	**Tamaño filtros C3**	**Cantidad total de parámetros**
-3	2 x 2	2 x 2	2 x 2	9.119.338
-2	3 x 3	3 x 3	3 x 3	9.346.058
-1	4 x 4	4 x 4	4 x 4	7.042.026
0	5 x 5	5 x 5	5 x 5	7.450.122
1	6 x 6	6 x 6	6 x 6	6.376.042
2	7 x 7	7 x 7	7 x 7	6.965.514

La figura 4.39 muestra que, para todo framework, el tiempo total de entrenamiento, si bien tiende a disminuir, no experimenta grandes variaciones al aumentar los tamaños de los filtros. Se observa, además, que los tiempos de Theano son mucho mayores a los de CNTK y TensorFlow. Por otra parte, los tiempos de estos dos últimos se mantienen cercanos, aunque no tanto como en secciones anteriores, siendo los tiempos de TensorFlow inferiores a los de CNTK en todos los experimentos realizados.

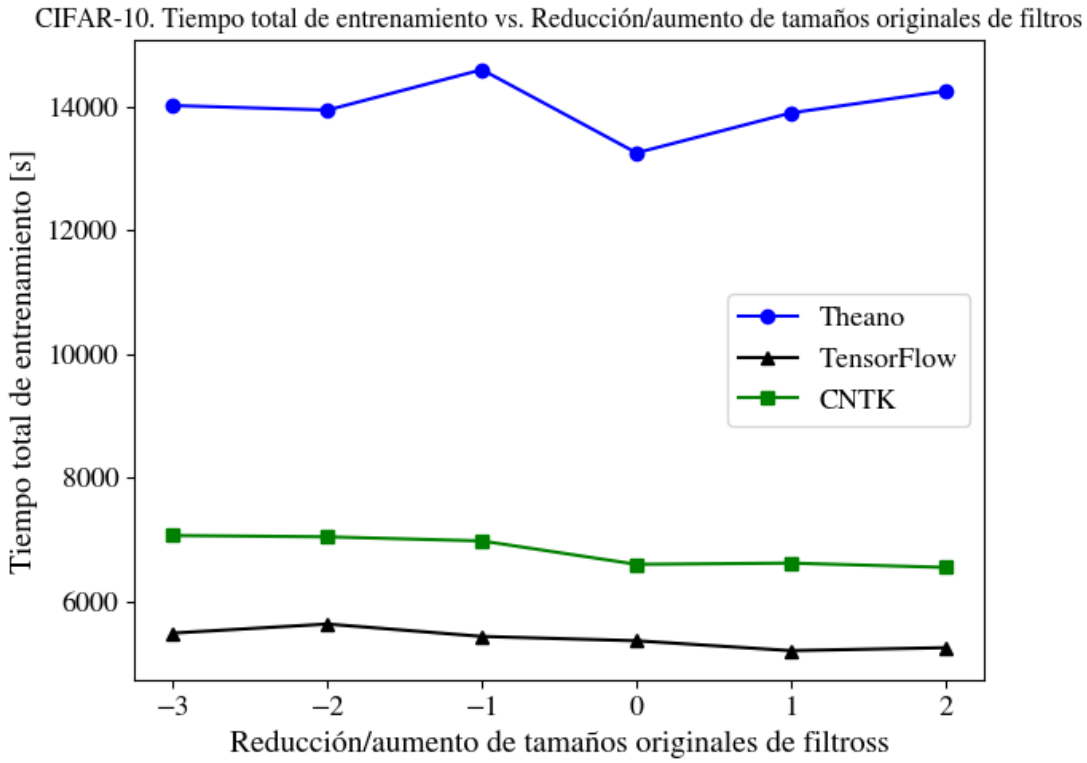


Figura 4.39: Variación de los tamaños de los filtros convolucionales de CNN + ReLU y su impacto en el tiempo total de entrenamiento. Dataset: CIFAR-10.

De acuerdo a la figura 4.40, TensorFlow es entre 2,5 y 2,7 veces más rápido que Theano, de manera que la diferencia más estrecha entre ambos se presenta al mantener intactos los tamaños originales de los filtros, mientras que la diferencia más grande se manifiesta al reducir los tamaños en una unidad. Además, CNTK es entre 2 y 2,2 veces más rápido que Theano; el valor mínimo se da al reducir los tamaños dos unidades, mientras que el máximo, al aumentarlos en tres unidades. Por otro lado, TensorFlow es entre 1,2 y 1,3 veces más rápido que CNTK; la relación alcanza el valor mínimo al mantener las dimensiones originales, mientras que el valor máximo es conseguido al reducir los tamaños en tres unidades.

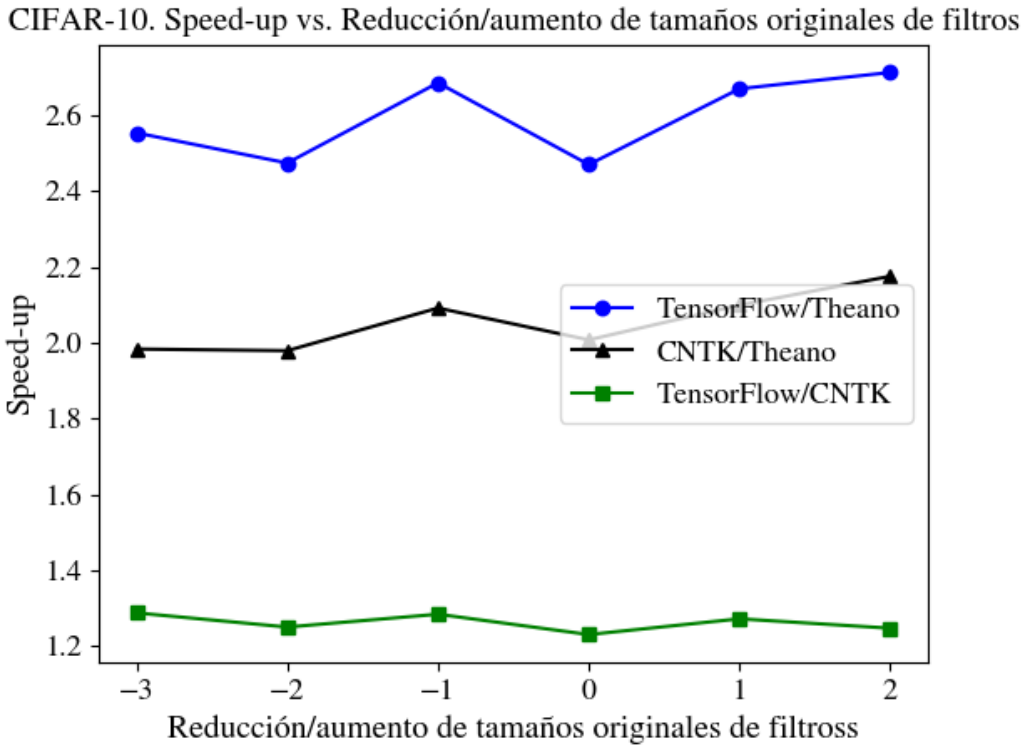


Figura 4.40: Razón entre los tiempos de entrenamiento de CNN + ReLU para cada par de frameworks en función de la variación en el tamaño de los filtros. Dataset: CIFAR-10.

Según muestra la figura 4.41, el consumo de memoria GPU en TensorFlow se mantiene prácticamente constante en cada experimento, excepto al reducir los tamaños en tres unidades, escenario en el que es 100 [MiB] menor. En el caso de CNTK, la evolución del consumo sigue un comportamiento bastante particular, ya que aumenta rápidamente al pasar de una reducción de -3 a una reducción de -1, para luego descender lentamente hasta un experimentar una brusca disminución al aumentar los tamaños de los filtros en dos unidades. Por otra parte, el consumo en Theano desciende cuasilinealmente.

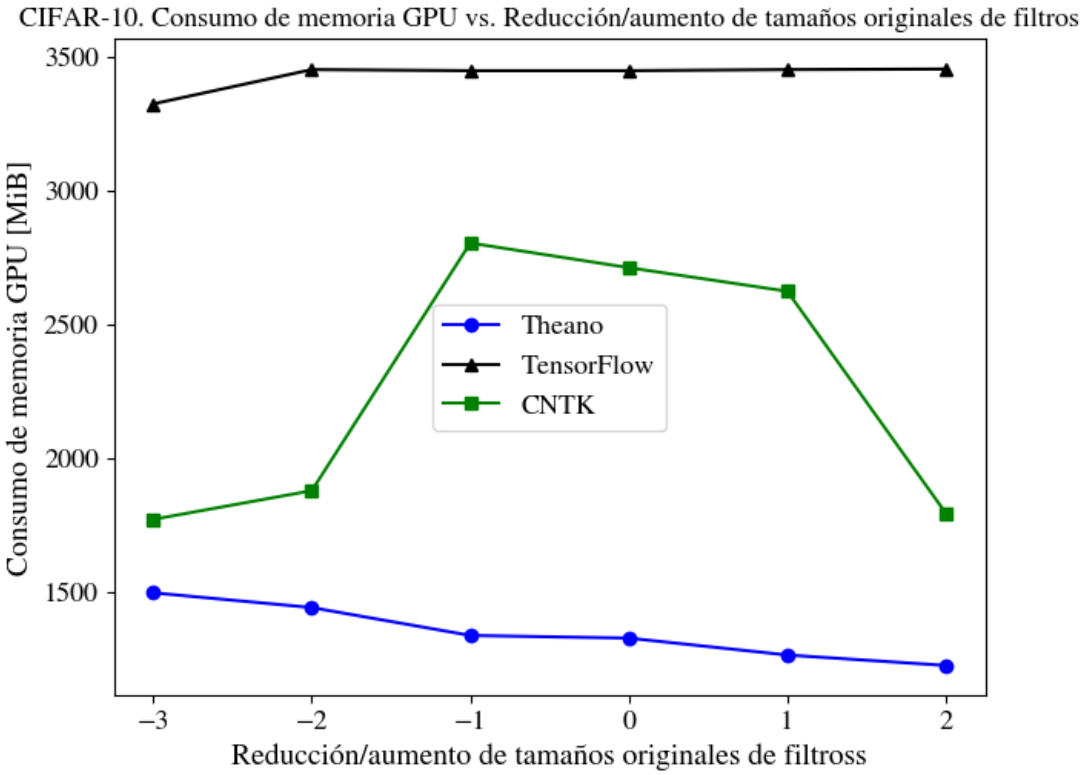


Figura 4.41: Variación de los tamaños de los filtros convolucionales de CNN + ReLU y su impacto en el consumo de memoria GPU. Dataset: CIFAR-10.

La figura 4.42 muestra que TensorFlow consume entre 2,2 y 2,8 veces más memoria que Theano. Por otro lado, CNTK logra una utilización de memoria GPU entre 1,2 y 2,1 veces más alta que la de Theano, donde el valor mínimo es obtenido al aplicar una reducción de dos unidades, mientras que el valor máximo es conseguido al reducir las dimensiones en una unidad. Finalmente, los consumos de TensorFlow llegan a ser entre 1,2 y 1,9 veces más altos que los de CNTK; los valores mínimo y máximo de la relación son conseguidos con una reducción de una unidad y un aumento de dos unidades, respectivamente.

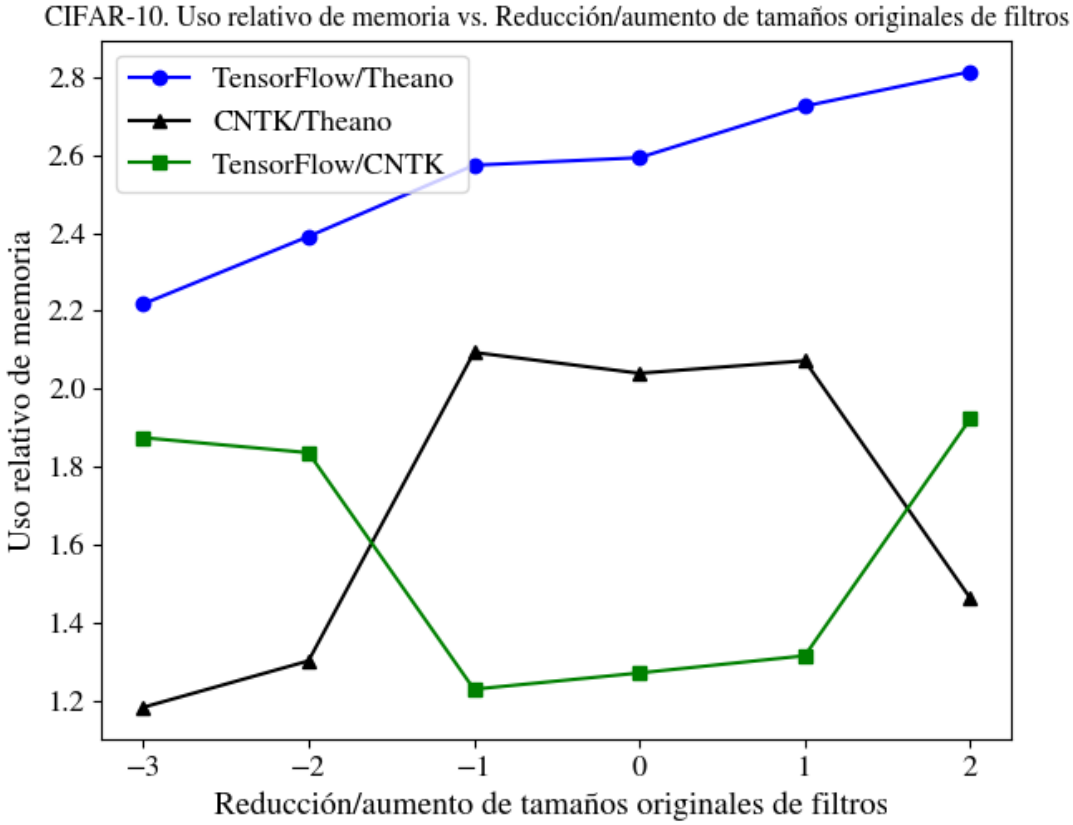


Figura 4.42: Uso relativo de memoria GPU entre cada par de frameworks en el entrenamiento de CNN + ReLU como función de la variación del tamaño de los filtros. Dataset: CIFAR-10.

4.6.2.3. SVHN

4.6.2.3.1. Arquitectura maestra

Se estudia la red WRN-16-8 propuesta en [46]. Su estructura puede observarse en la figura 4.43. Por defecto, las capas convolucionales consideran un stride de 1, excepto donde se especifica lo contrario. En este último escenario, la capa es denotada por la expresión $fxf\ conv\ F/s$, donde fxf define las dimensiones de los filtros, F la cantidad de estos y s el stride aplicado. La etiqueta $avgpool/8$ representa el empleo de una capa de average pooling con pooling size de 8 x 8. La expresión $FC, 10$ denota una capa totalmente conectada de 10 neuronas.

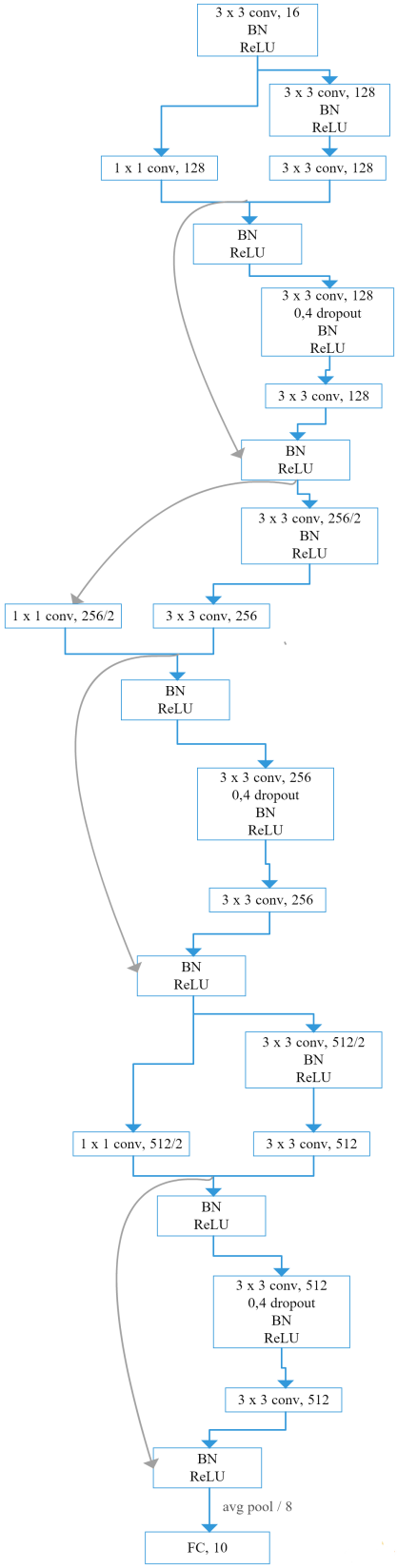


Figura 4.43: Estructura de WRN-16-8.

La arquitectura está conformada por 10.961.370 parámetros entrenables. Además, WRN-16-8 consigue un error de prueba del 1,54 %.

Para el pre-procesamiento de los datos, se divididen las intensidades de pixel de cada imagen por 255 de manera que se sitúen en el rango [0, 1]. Los pesos de cada capa son inicializados por medio de una distribución He Normal. Para toda capa oculta, se utiliza ReLU como función de activación y BN previo a la primera capa convolucional de cada bloque residual (i.e. de cada bloque de dos capas convolucionales consecutivas). Para el entrenamiento del modelo, se utiliza SGD como método de optimización, tamaño de mini-batch 128 y se entrena durante 160 epochs. La tasa de aprendizaje inicial es de 0,01 y es multiplicada por 0,1 en los epochs 80 y 120. Además, se utiliza weight decay de 0,0005 y momentum de Nesterov de 0,9.

En la tabla 4.19 se muestra el rendimiento de cada framework sobre WRN-16-8. Cabe hacer notar que al experimentar con esta arquitectura se registran los tiempos más elevados de los todos los experimentos realizados hasta ahora, debido a la gran cantidad de parámetros que posee la red. Aquí, las diferencias de tiempo entre los frameworks son notables. TensorFlow consigue el tiempo de entrenamiento total más bajo, siendo aproximadamente la mitad del empleado por CNTK, la mayor diferencia hasta ahora entre ambos frameworks. Más aún, TensorFlow entrena el modelo en aproximadamente 27 horas menos que Theano, esto es, la cuarta parte del tiempo requerido por este último. Respecto a los errores, nuevamente las diferencias en la evaluación son mínimas, siendo la más notable la existente entre TensorFlow y CNTK (0,2 %).

Cuadro 4.19: Rendimiento sobre arquitectura WRN-16-8. Dataset: SVHN.

Framework	**Tiempo de entrenamiento total [s]**	**Consumo de memoria GPU [MiB]**	**Error de entrenamiento [%]**	**Error de prueba [%]**
Theano	134023,3	2983	0,00	5,94
TensorFlow	34594,9	3830	0,00	5,97
CNTK	64476,7	5459	0,00	5,77

4.6.2.3.2. Arquitectura básica

Ante la imposibilidad de encontrar un modelo que satisficiera los requerimientos planteados en las secciones 4.3 y 4.6.2, se trabaja con una red propuesta por el profesor Ricardo Ñanculef, la que, para efectos prácticos, será denominada basic-SVHN. Las capas de dicha red se describen a continuación, en orden de aparición:

- Una capa convolucional con 16 filtros de 5 x 5.
- Una capa de max pooling con pooling size de 2 x 2.
- Una capa de dropout de 0,25.
- Una capa convolucional con 512 filtros de 7 x 7.
- Una capa de max pooling con pooling size de 2 x 2.
- Una capa de dropout de 0,5.
- Dos capas totalmente conectadas con 1024 neuronas y dropout de 0,5, cada una.
- Una capa totalmente conectada con 10 neuronas y función de activación softmax (capa de salida).

Los datos son preprocesados dividiendo las intensidades de pixel de cada imagen por 255. Existe uso de BN después de cada capa convolucional y a continuación de las capas totalmente conectadas. Como función de activación, se considera ReLU, la cual es aplicada sobre el output de cada capa de BN. A continuación de las capas de pooling, se aplica dropout de 0,25 y 0,5, respectivamente. A continuación de cada capa totalmente conectada, se aplica dropout de 0,5. El modelo es entrenado con ADAM como método de optimización, tamaño mini-batch 256, durante 160 epochs. Se utiliza una tasa de aprendizaje inicial de 0,01, siendo posteriormente multiplicada por el factor 0,1 en los epochs 80 y 120. Además, se utiliza weight decay de 0,0005.

En la tabla 4.20 se muestra el rendimiento de cada framework sobre esta arquitectura. Al igual que para la red maestra, TensorFlow es quien mejor rinde en términos de tiempo, siendo ampliamente superior a CNTK y Theano. TensorFlow entrena el modelo en aproximadamente 1,5 horas menos que CNTK y en casi 5,5 horas menos que Theano, es decir, la mitad y la cuarta parte de lo que estos dos últimos demoran, respectivamente. Respecto a la utilización de la GPU, CNTK consume 1,5 veces más memoria que Ten-

sorFlow y 2,8 veces más que Theano. Nuevamente, los errores, tanto de entrenamiento como de prueba, son muy cercanos entre sí. En el primer caso, TensorFlow logra una diferencia de 0,06 % respecto a sus competidores. En el segundo caso, la mayor diferencia se presenta entre CNTK y Theano (0,6 %).

Cuadro 4.20: Rendimiento sobre arquitectura basic-SVHN. Dataset: SVHN.

Framework	**Tiempo de entrenamiento total [s]**	**Consumo de memoria GPU [MiB]**	**Error de entrenamiento [%]**	**Error de prueba [%]**
Theano	25913,8	1269	0,54	9,00
TensorFlow	6637,0	2298	0,48	8,59
CNTK	12394,5	3544	0,54	8,39

4.6.2.3.3. Variación de cantidad de ejemplos de entrenamiento

Como en datasets anteriores, se varía el porcentaje de imágenes utilizadas para el entrenamiento de la arquitectura básica. La tabla 4.21 muestra las cantidades de datos correspondientes a cada uno de los porcentajes considerados.

Cuadro 4.21: fracciones de ejemplos usadas para el entrenamiento de basic-SVHN. Dataset: SVHN.

Fracción de ejemplos [%]	**Número de ejemplos**
33	24.174
40	29.302
57	41.756
64	46.884
83	60.803
100	73.257

En la figura 4.44 se aprecia que TensorFlow obtiene el tiempo de entrenamiento total más bajo de todos para cada una de los subconjuntos de entrenamiento considerados. Notar que de todos los datasets estudiados hasta ahora, es en este donde la distancia entre las curvas de TensorFlow y CNTK más se acrecienta con el aumento de ejemplos de entrenamiento, lo que puede ser un indicio de que tan bien (o que tan mal) escalan los frameworks al aumentar significativamente la cantidad de filtros en una o más capas convolucionales (tal y como ocurre en el caso de la segunda capa convolucional de esta arquitectura). Nuevamente, Theano emplea el mayor de los tiempos para entrenar el modelo, en cada caso.

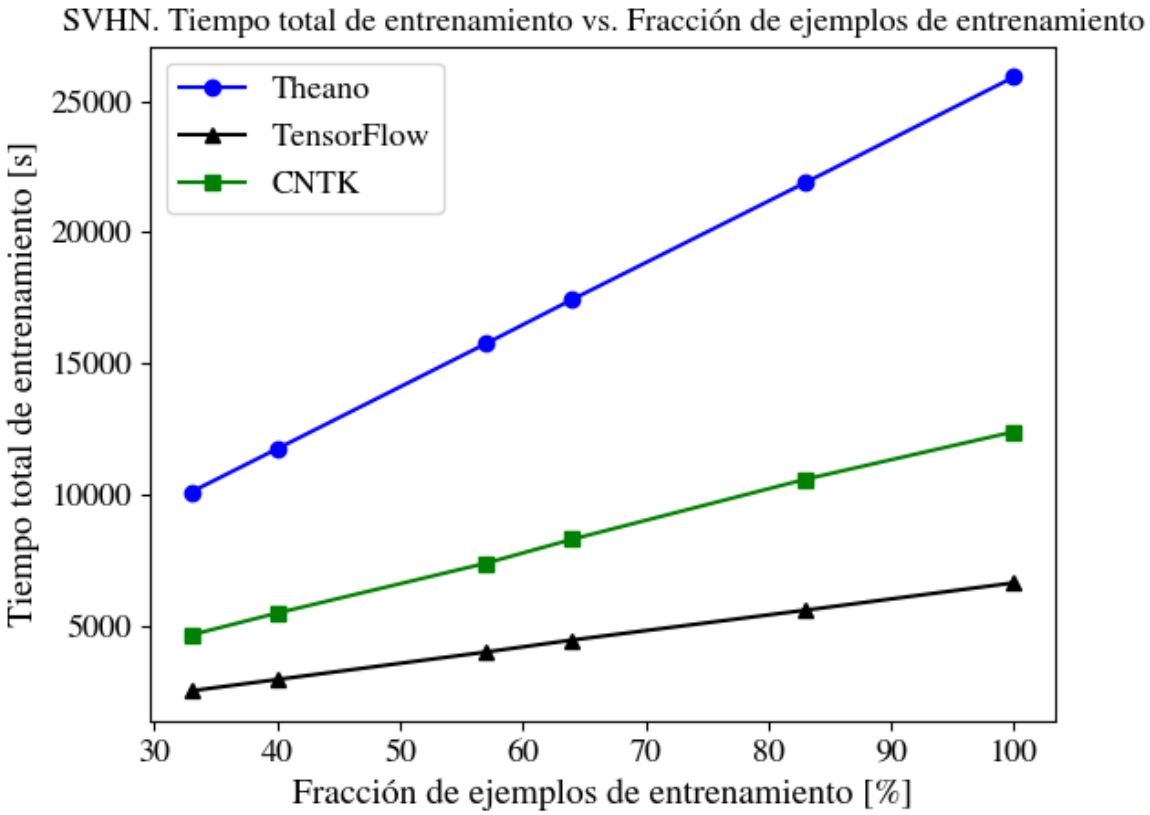


Figura 4.44: Variación de la fracción de ejemplos provistos para el entrenamiento de basic-SVHN y su impacto en el tiempo total de entrenamiento. Dataset: SVHN.

La figura 4.45 muestra que Theano no es capaz de procesar más de 2 ejemplos completos por segundo, para toda fracción de datos posible. De igual forma, CNTK tampoco logra que esta métrica aumente, aunque es capaz de procesar 5 ejemplos completos en cada segundo. Por otro lado, TensorFlow consigue un aumento de 2 ejemplos procesados, desde 9 ejemplos completos por segundo para el 33 % del conjunto de entrenamiento original, hasta 11 ejemplos completos por segundo con el conjunto completo.

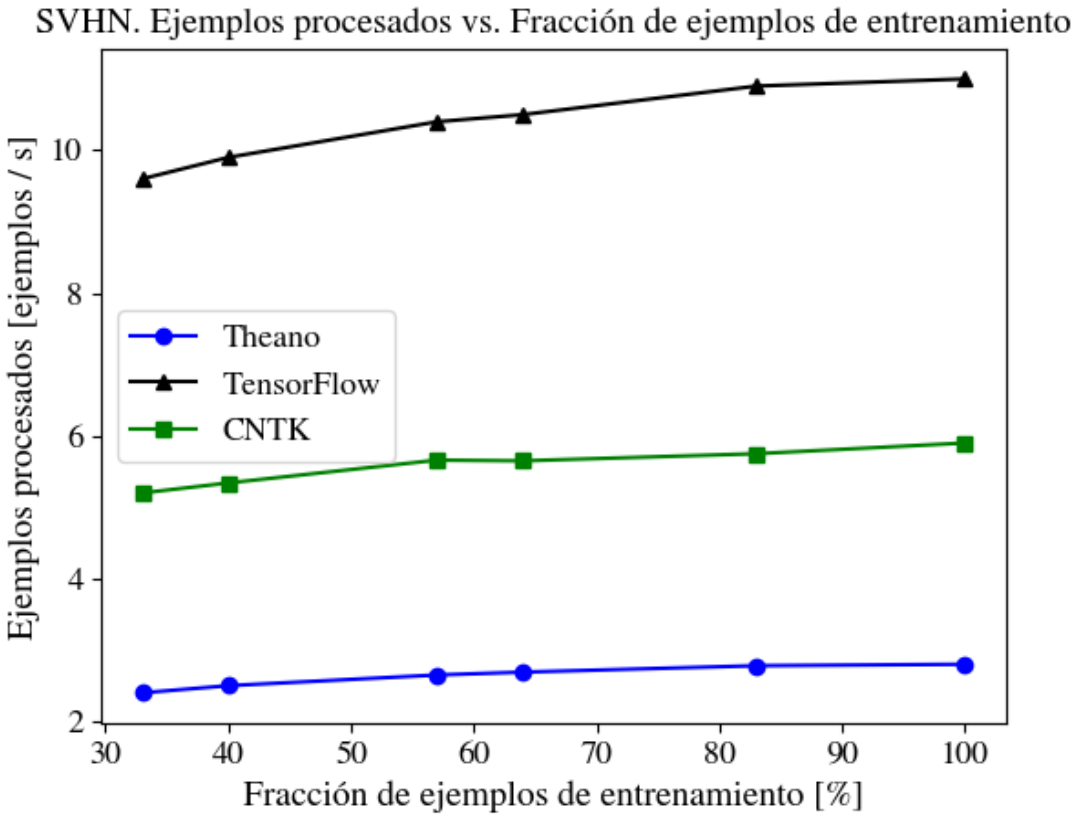


Figura 4.45: Variación de la fracción de ejemplos provistos para el entrenamiento de basic-SVHN y su impacto en la cantidad promedio de ejemplos procesados. Dataset: SVHN.

La figura 4.46 muestra que TensorFlow emplea entre 1,8 y 1,9 veces menos tiempo que CNTK en cada caso, estableciendo una brecha importante entre ambos frameworks con respecto a los dos datasets anteriores. De igual manera, la relación entre los tiempos de TensorFlow y Theano también experimenta grandes cambios, siendo el primero entre 3,9 y 4 veces más rápido que el segundo. Además, CNTK resulta ser entre 2,1 y 2,2 veces más rápido que Theano en cada escenario.

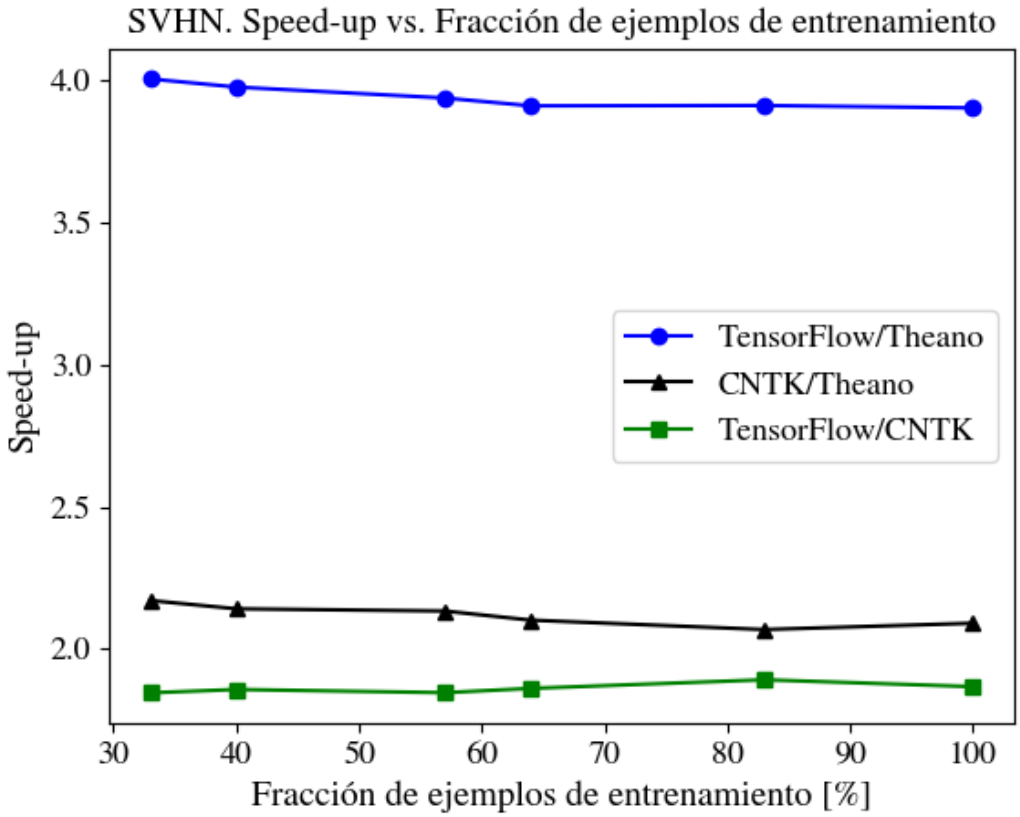


Figura 4.46: Speed-up relativo de tiempos de entrenamiento de basic-SVHN entre cada par de frameworks en función de la fracción de ejemplos provistos. Dataset: SVHN.

4.6.2.3.4. Variación de la cantidad de capas convolucionales

En forma similar a secciones previas, se estudia el impacto de aumentar la cantidad de capas convolucionales en basic-SVHN. La tabla 4.22 muestra cada una de las arquitecturas estudiadas.

Cuadro 4.22: Aumento en la cantidad de capas convolucionales de basic-SVHN, junto con el número total de parámetros del modelo resultante. Dataset: SVHN.

Factor de aumento	**Cantidad total de capas convolucionales**	**Cantidad total de parámetros**
1	2	35.023.594
2	4	47.876.634
3	6	60.729.674
4	8	73.582.714
5	10	86.435.754

La figura 4.47 da cuenta de que cada framework presenta un comportamiento particular que difiere del de los demás. Al igual que en casos anteriores, el tiempo total de entrenamiento aumenta rápidamente y de forma lineal en Theano. Por otro lado, en TensorFlow el tiempo también aumenta en forma lineal, pero de forma mucho más moderada que en Theano. Notar también que a diferencia de datasets anteriores, el comportamiento del tiempo en CNTK no varía en forma lineal, si no más bien en forma súper lineal. En estas condiciones, su rendimiento es notablemente inferior al de TensorFlow, lo que no hace más que aumentar las sospechas de que CNTK es altamente sensible a incrementos significativos en la cantidad de parámetros de una red convolucional. Más aún, no fue posible aumentar la cantidad de capas convolucionales en un factor de 5 en CNTK, al producirse un error de memoria GPU insuficiente.

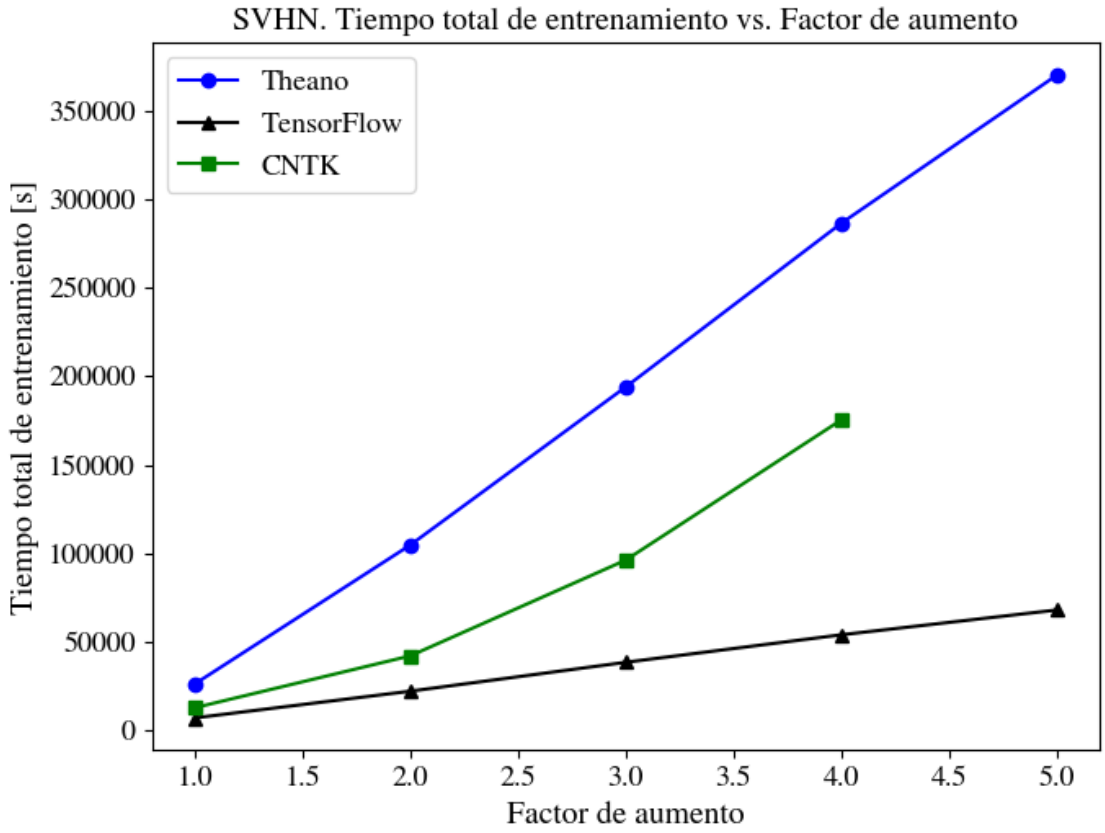


Figura 4.47: Incremento de la cantidad de capas convolucionales en basic-SVHN y su impacto en el tiempo total de entrenamiento. Dataset: SVHN.

Tal y como muestra la figura 4.48, las diferencias entre frameworks son las más destacables hasta ahora, ya que los tiempos de TensorFlow son entre 3,9 y 5,5 veces más bajos que los de Theano. Además, el speed-up relativo entre CNTK y Theano decrece desde 2,5 hasta 1,6, lo que deja en evidencia que el rendimiento de CNTK, en vez de mantenerse cercano al de TensorFlow, comienza a acercarse cada vez más al de Theano. Esta idea se reafirma al constatar que TensorFlow pasa de ser 1,9 a 3,3 veces más rápido que CNTK.

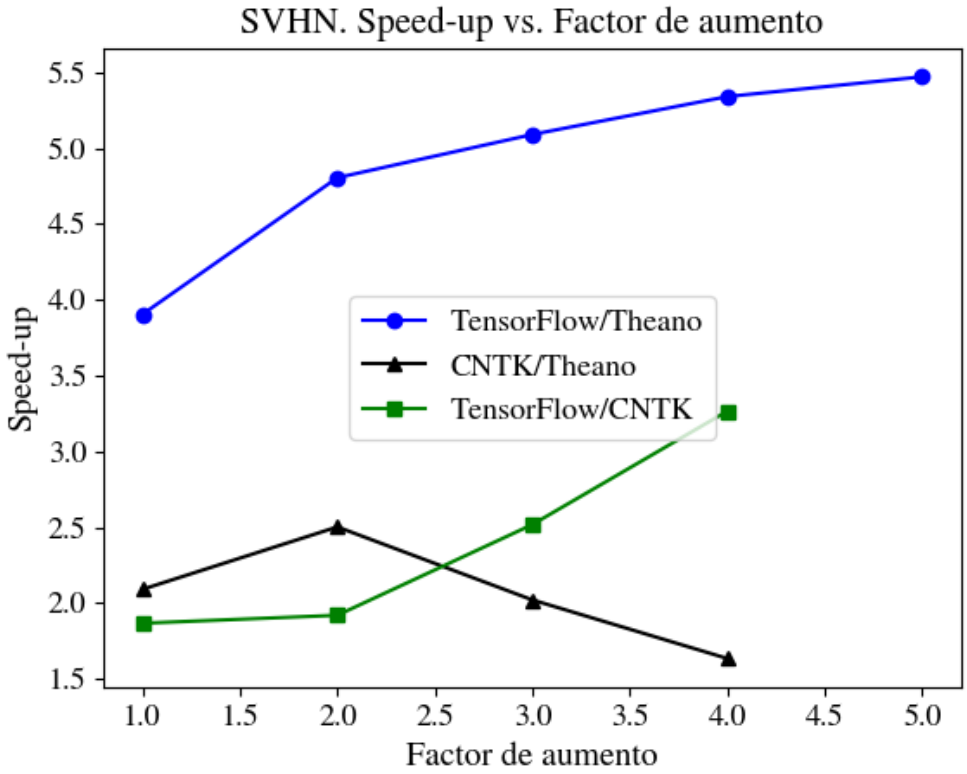


Figura 4.48: Razón entre los tiempos de entrenamiento de basic-SVHN para cada par de frameworks en función del factor de aumento. Dataset: SVHN.

A diferencia de lo observado en datasets anteriores, las curvas de consumo de CNTK y TensorFlow presentan un comportamiento similar al de la función raíz cuadrada, según muestra la figura 4.49. Sin embargo, la utilización de la GPU es ampliamente superior en el primer framework, lo que impide que este sea capaz de entrenar el modelo al quintuplicar el número de capas convolucionales. Por otra parte, la curva asociada a Theano evoluciona cuasi-linealmente y de modo mucho más moderado en base al factor de aumento.

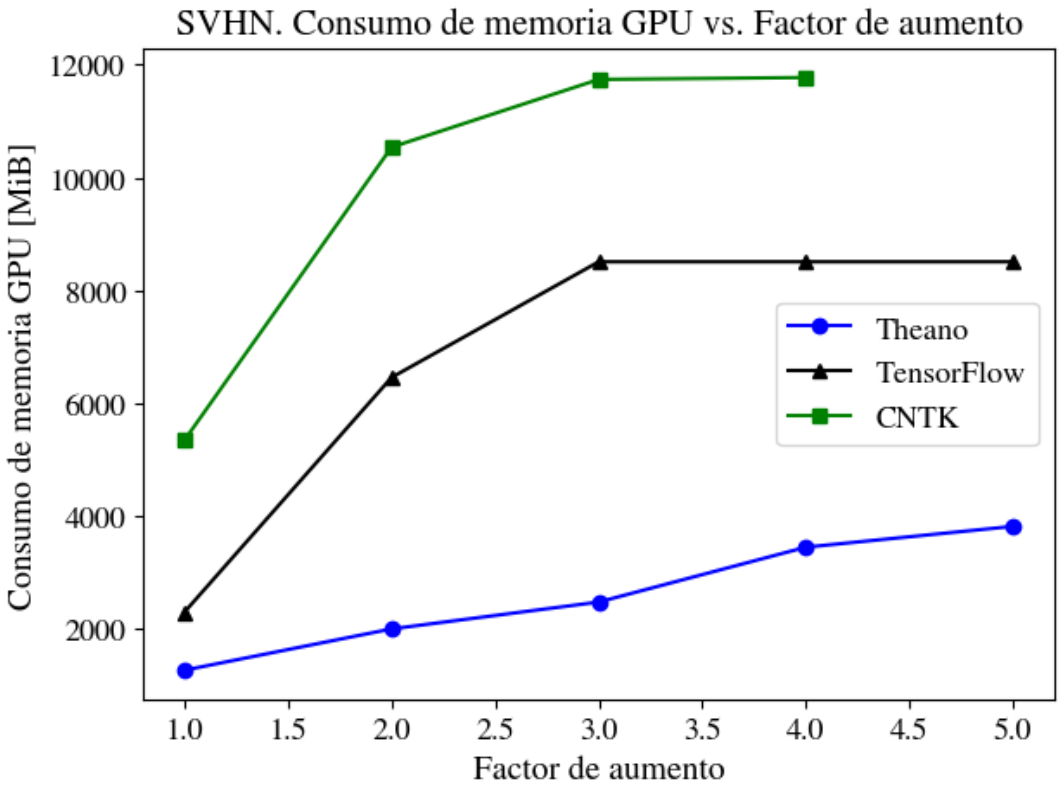


Figura 4.49: Incremento de la cantidad de capas convolucionales en basic-SVHN y su impacto en el consumo de memoria GPU. Dataset: SVHN.

En la figura 4.50 se aprecia que TensorFlow consume entre 1,8 y 3,4 veces más memoria GPU que Theano. Además, CNTK requiere entre 1,4 y 2,5 veces más memoria que TensorFlow, invirtiendo los papeles con respecto a lo observado en datasets anteriores. Por otra parte, el cociente entre los consumos de CNTK y Theano alcanza el valor máximo de 5,3 con un factor de 2, para luego decrecer y alcanzar un valor mínimo de 3,4 con un factor de 4.

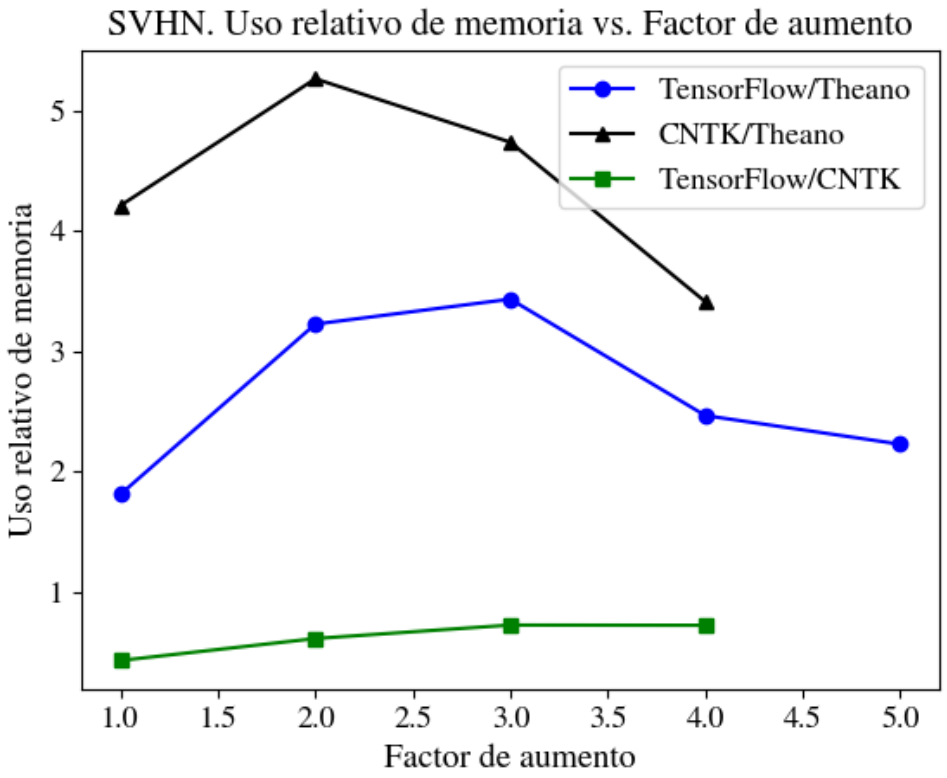


Figura 4.50: Uso relativo de memoria GPU entre cada par de frameworks en el entrenamiento de basic-SVHN como función del factor de aumento. Dataset: SVHN.

4.6.2.3.5. Variación de la cantidad de filtros convolucionales

Como en secciones anteriores, se modifica la cantidad de filtros en cada capa convolucional de basic-SVHN. Las configuraciones estudiadas se muestran en la tabla 4.23.

Cuadro 4.23: Cantidad de filtros y parámetros para cada variación de basic-SVHN, donde C_i corresponde a la capa convolucional i del modelo. Dataset: SVHN.

Factor de variación	**Número de filtros** C_1	**Número de filtros** C_2	**Cantidad total de parámetros**
$\frac{1}{4}$	4	128	9.479.362
$\frac{1}{2}$	8	256	17.943.930
1	16	512	35.023.594
2	32	1024	69.785.034
4	64	2048	141.724.682

La figura 4.51 muestra que el tiempo total de entrenamiento varía en forma exponencial en Theano y CNTK, mientras que en TensorFlow lo hace linealmente. Si bien los tiempos de CNTK y TensorFlow son cercanos al reducir la cantidad de filtros a la cuarta parte, las diferencias entre ambos son cada vez más notables al aumentar dicha cantidad, de tal manera que el rendimiento de CNTK se asemeja más al de Theano que al de TensorFlow. Notar, además, que CNTK es incapaz de entrenar la red al cuadruplicar el número de filtros, como consecuencia de un error de memoria GPU insuficiente.

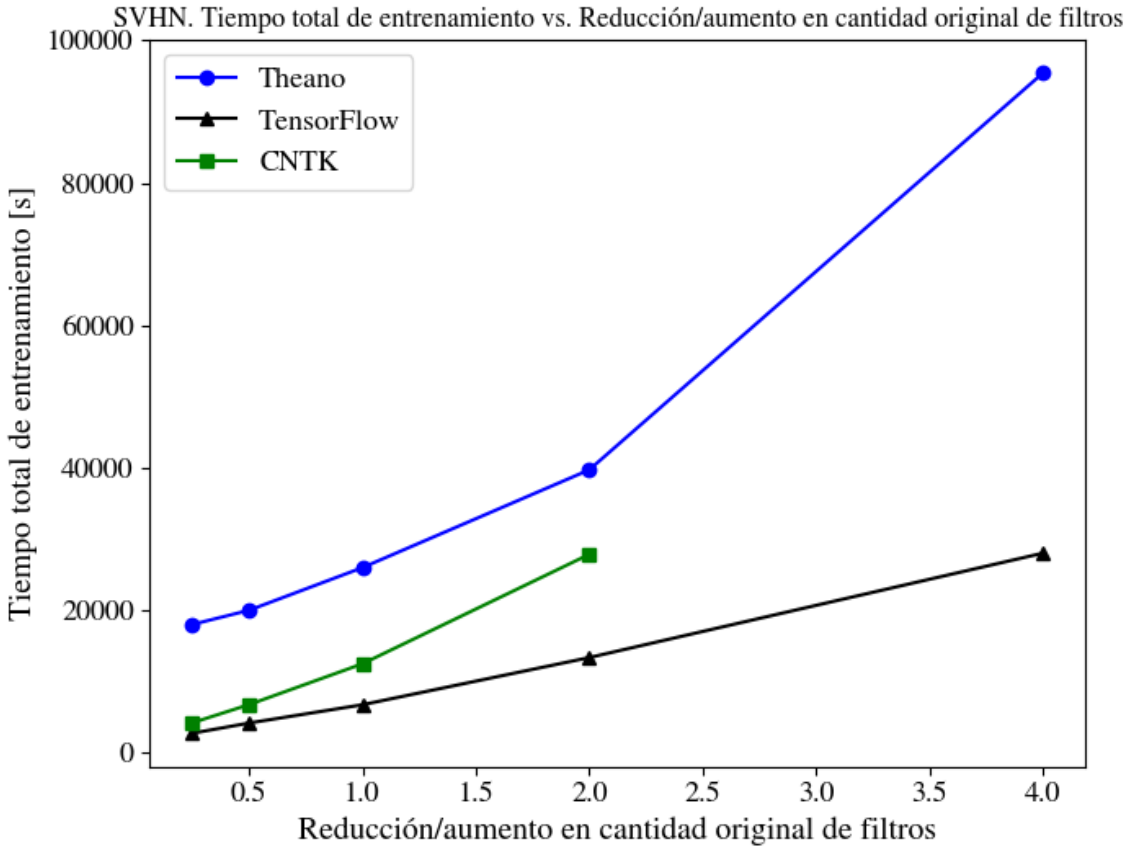


Figura 4.51: Variación de la cantidad de filtros convolucionales en basic-SVHN y su impacto en el tiempo total de entrenamiento. Dataset: SVHN.

La figura 4.52 muestra que el speed-up relativo entre los tiempos de TensorFlow y Theano tiende a descender con el aumento de la cantidad de filtros, siendo el primero entre 3 y 6,8 veces más rápido que el segundo al aumentar dicha cantidad al doble de la original y al reducirla a un cuarto, respectivamente. El speed-up relativo entre CNTK y Theano también es descendente, siendo el primero entre 1,4 y 4,4 veces más rápido que el segundo y donde los valores máximo y mínimo de esta razón son alcanzados con un cuarto y con el doble de los filtros originales, en cada caso. Por otra parte, el speed-up relativo entre TensorFlow y CNTK crece linealmente con el aumento aplicado sobre el número de filtros, siendo el primero entre 1,6 y 2,1 veces más rápido que el segundo.

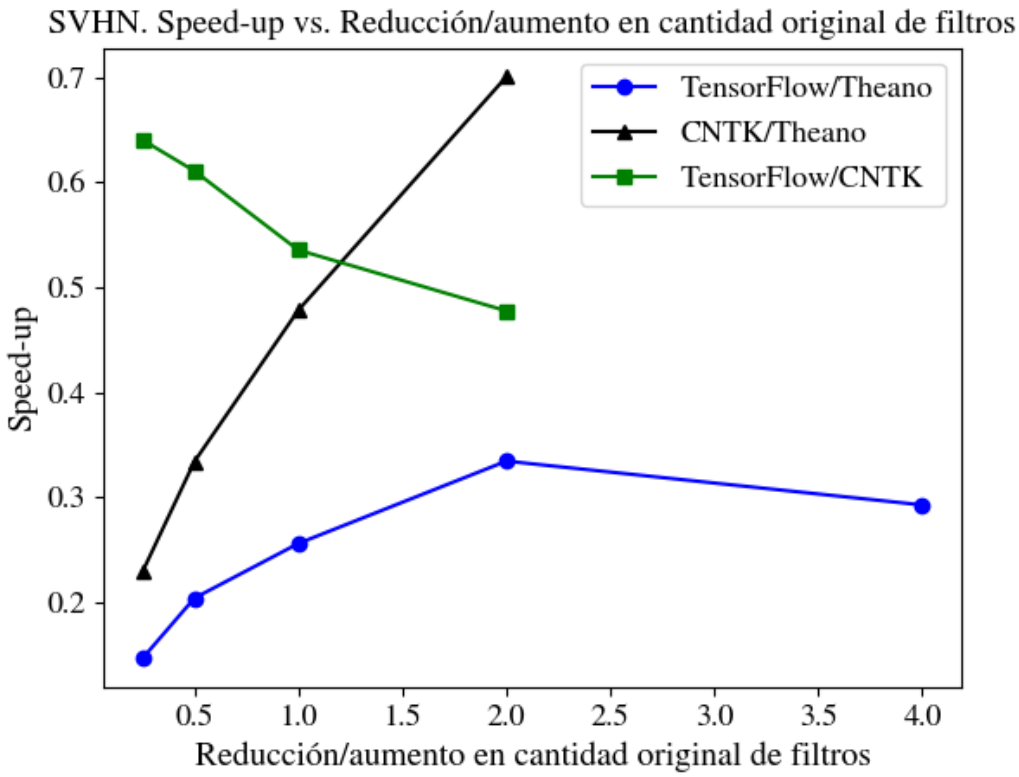


Figura 4.52: Razón entre los tiempos de entrenamiento de basic-SVHN para cada par de frameworks en función de la variación en la cantidad de filtros. Dataset: SVHN.

En la figura 4.53 se aprecia que el cambio en el consumo de memoria GPU en función del factor de variación se manifiesta de maneras muy diferentes en cada framework. En Theano, el cambio evoluciona en forma lineal, mientras que en TensorFlow lo hace súper linealmente hasta duplicar el número de filtros, manteniéndose constante al cuadruplicar dicho número. Por otra parte, el consumo en CNTK varía rápidamente, siguiendo un comportamiento también súper lineal. En este dataset, los consumos son muy similares sólo al reducir el número de filtros a la cuarta parte. Además, la utilización de GPU en CNTK es superior en todo momento a la de TensorFlow, a diferencia de lo observado en los casos anteriormente estudiados.

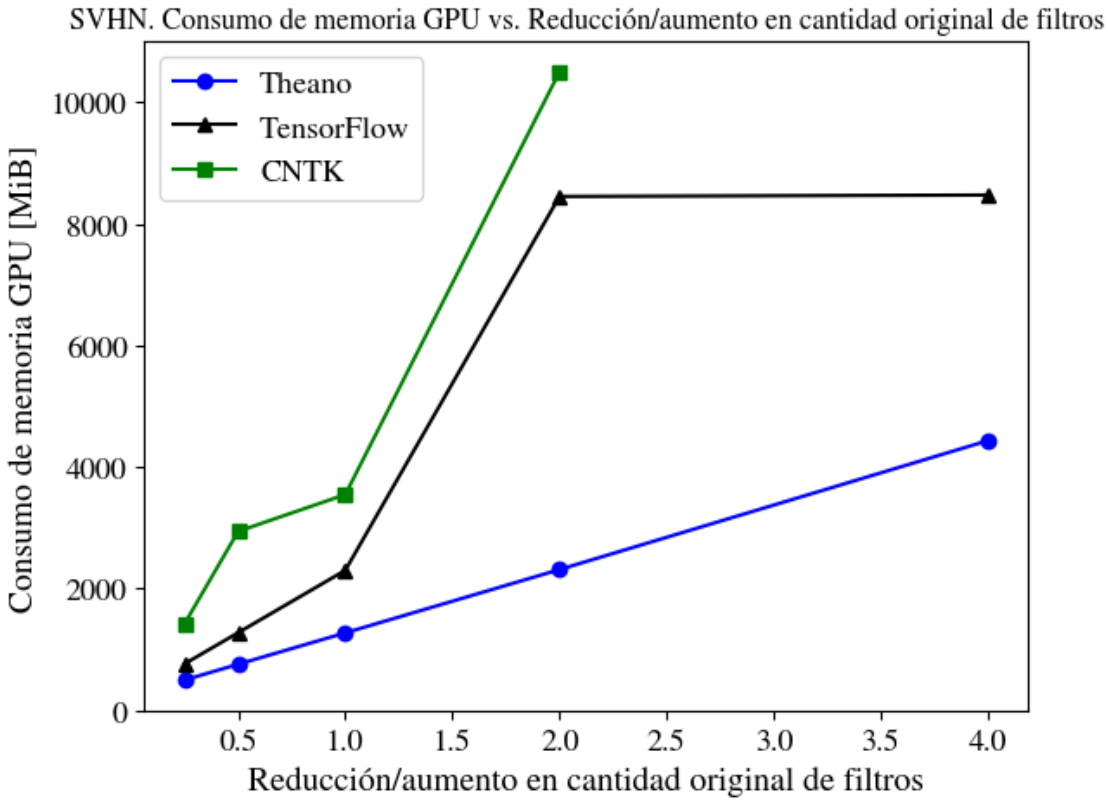


Figura 4.53: Variación de la cantidad de filtros convolucionales de basic-SVHN y su impacto en el consumo de memoria GPU. Dataset: SVHN.

La figura 4.54 muestra que TensorFlow consume entre 1,5 y 3,7 veces más memoria que Theano, donde el valor mínimo es alcanzando con un cuarto de la cantidad inicial de filtros, mientras que el valor máximo con el doble. Por otra parte, el consumo de CNTK es entre 2,8 y 4,5 veces superior al de Theano, donde el valor mínimo también es obtenido con la cantidad original de filtros, mientras que el valor máximo con el doble. Además, TensorFlow consume entre 0,4 y 0,8 veces la cantidad de memoria que es utilizada por CNTK, de manera que el valor mínimo es alcanzado con la mitad de los filtros iniciales, mientras que el máximo se corresponde una vez más con el doble; esto es, CNTK hace una utilización entre 1,3 y 2,5 veces superior a la de TensorFlow.

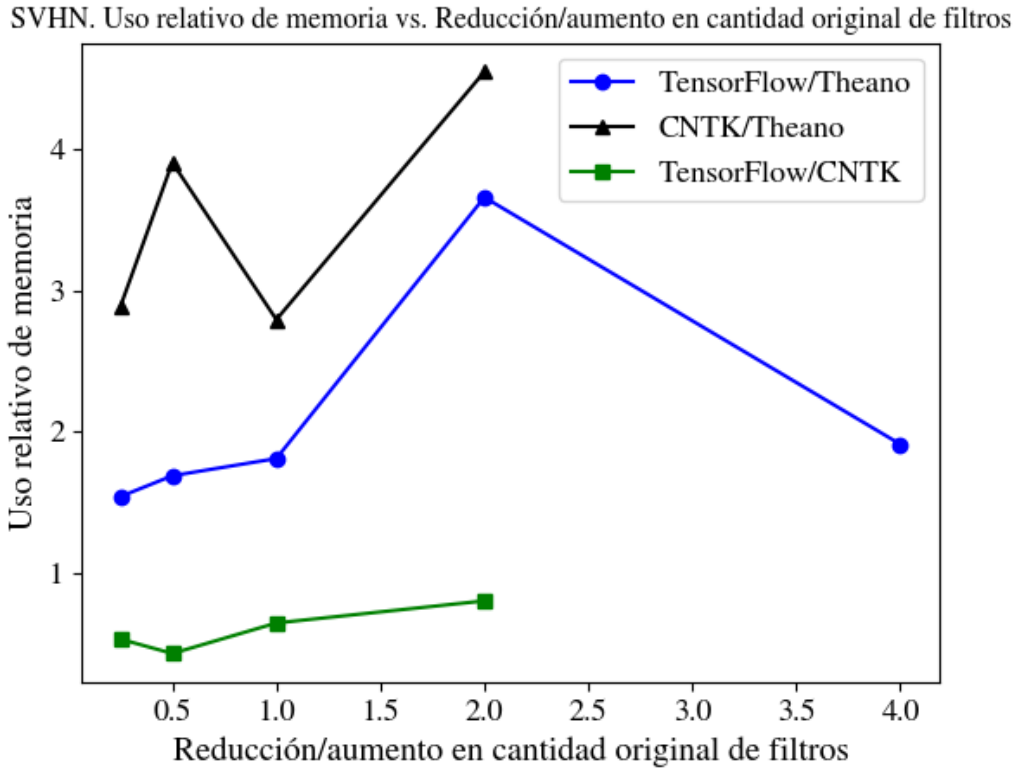


Figura 4.54: Razón entre los consumos de memoria GPU para cada par de frameworks en el entrenamiento de basic-SVHN como función de la variación en la cantidad de filtros. Dataset: SVHN.

4.6.2.3.6. Variación de tamaños de filtros convolucionales

En forma análoga a secciones anteriores, se estudia el impacto de modificar los tamaños de los filtros convolucionales en basic-SVHN. La tabla 4.24 expone cada una de las modificaciones consideradas. Notar que no ha sido necesario aumentar las dimensiones de las imágenes originales, pero si remover la aplicación de padding sobre los inputs de las capas convolucionales, ya que para ciertos tamaños de filtros, el padding que debe ser aplicado es de tipo asimétrico, característica no implementada eficientemente en CNTK.

Cuadro 4.24: Disminución / incremento en el tamaño de los filtros de las capas convolucionales de basic-SVHN, junto con la cantidad de parámetros resultantes. C_i representa la capa convolucional i. Dataset: SVHN.

Cambio en tamaño de filtros	**Tamaño filtros C1**	**Tamaño filtros C2**	**Cantidad total de parámetros**
-3	2 x 2	4 x 4	20.072.186
-2	3 x 3	5 x 5	14.378.986
-1	4 x 4	6 x 6	9.750.842
0	5 x 5	7 x 7	9.857.770
1	6 x 6	8 x 8	6.311.162
2	7 x 7	9 x 9	3.829.610

La figura 4.55 muestra que, para todo framework, el tiempo total de entrenamiento decrece linealmente en función de la variación aplicada sobre los tamaños de los filtros. La reducción del tiempo es más rápida en Theano, mientras que en TensorFlow se da de forma más lenta, si bien los tiempos en este último son muy inferiores a los del primero. Nuevamente, los tiempos de TensorFlow y CNTK son cada vez más cercanos entre sí mientras mayores son las dimensiones de los filtros resultantes.

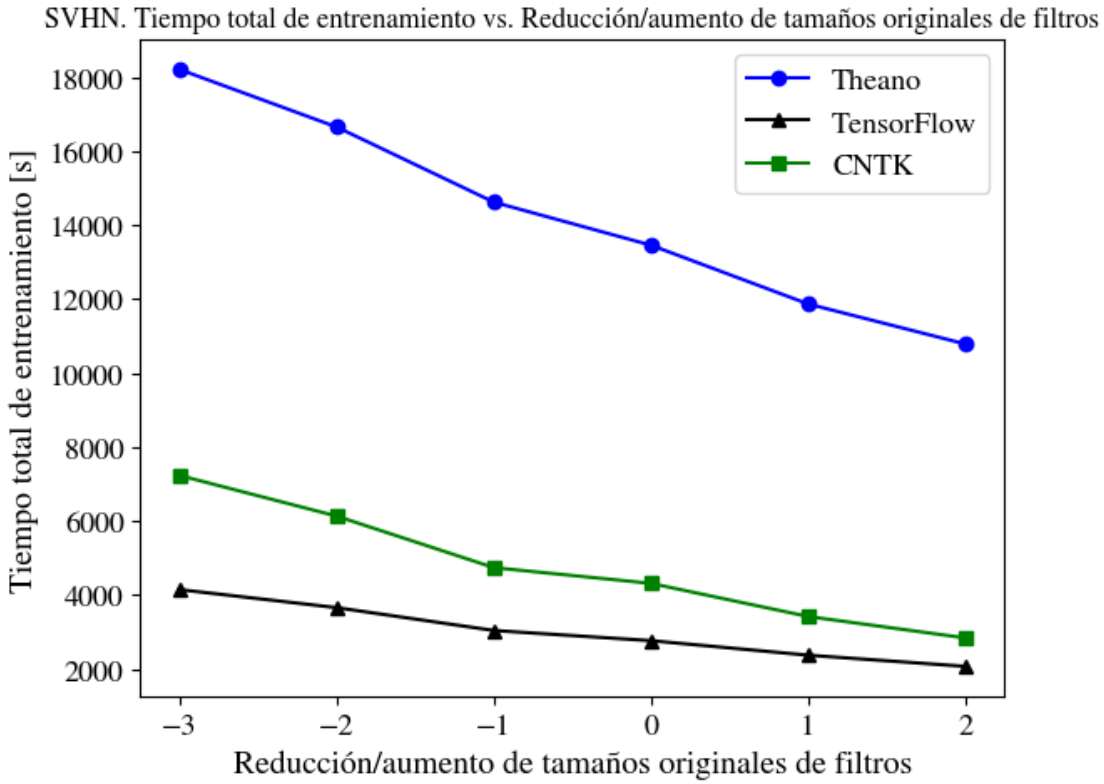


Figura 4.55: Variación de los tamaños de los filtros convolucionales de basic-SVHN y su impacto en el tiempo total de entrenamiento. Dataset: SVHN.

La figura 4.56 muestra que TensorFlow es entre 4,4 y 5,2 veces más rápido que Theano. También, CNTK es entre 2,5 y 3,8 veces más rápido que Theano. Finalmente, TensorFlow es entre 1,4 y 1,7 veces más rápido que CNTK, si bien, a diferencia de los speed-up anteriores, la diferencia máxima entre frameworks se presenta al reducir las dimensiones de los filtros en tres unidades, mientras que la diferencia es mínima al aumentar dichas dimensiones en dos unidades.

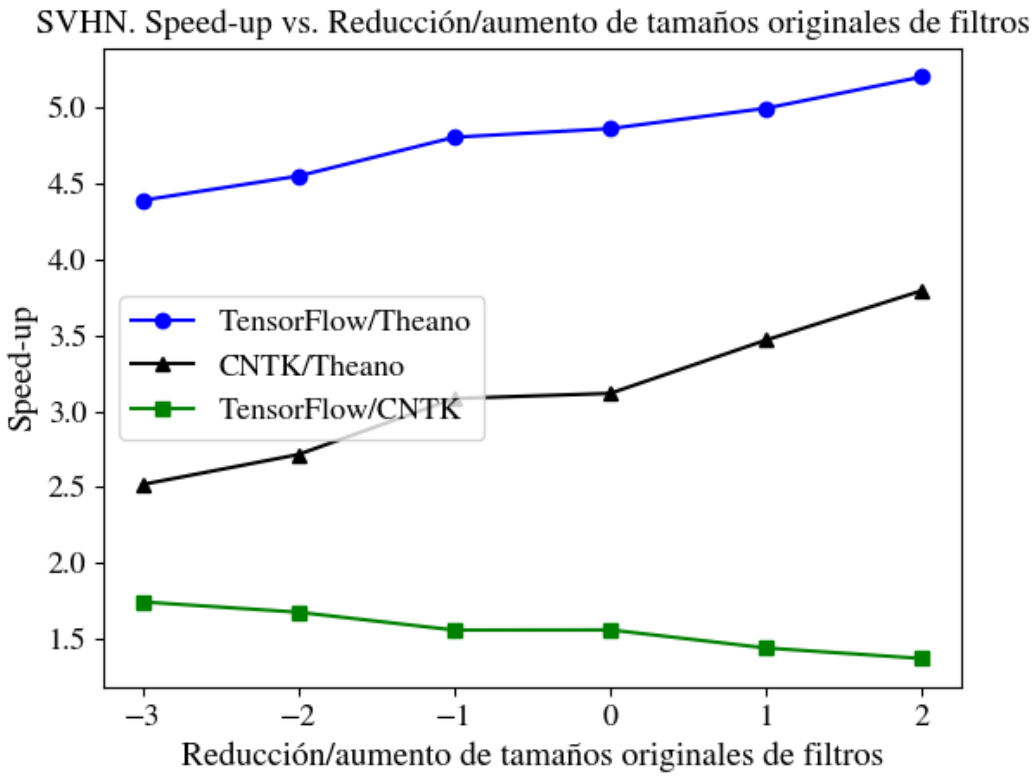


Figura 4.56: Speed-up relativo entre los tiempos de entrenamiento de basic-SVHN de cada par de frameworks en función de la variación en los tamaños de los filtros. Dataset: SVHN.

Según muestra la figura 4.57, el consumo de memoria es prácticamente invariable en TensorFlow. Por otro lado, el consumo decrece rápidamente en CNTK cuando los tamaños de los filtros resultantes aumentan. En Theano, en cambio, la utilización de memoria GPU decrece cuasi-linealmente al aumentar las dimensiones resultantes. Notar, además, que al reducir los tamaños en dos y en tres unidades, CNTK utiliza una mayor cantidad de memoria en comparación con TensorFlow.

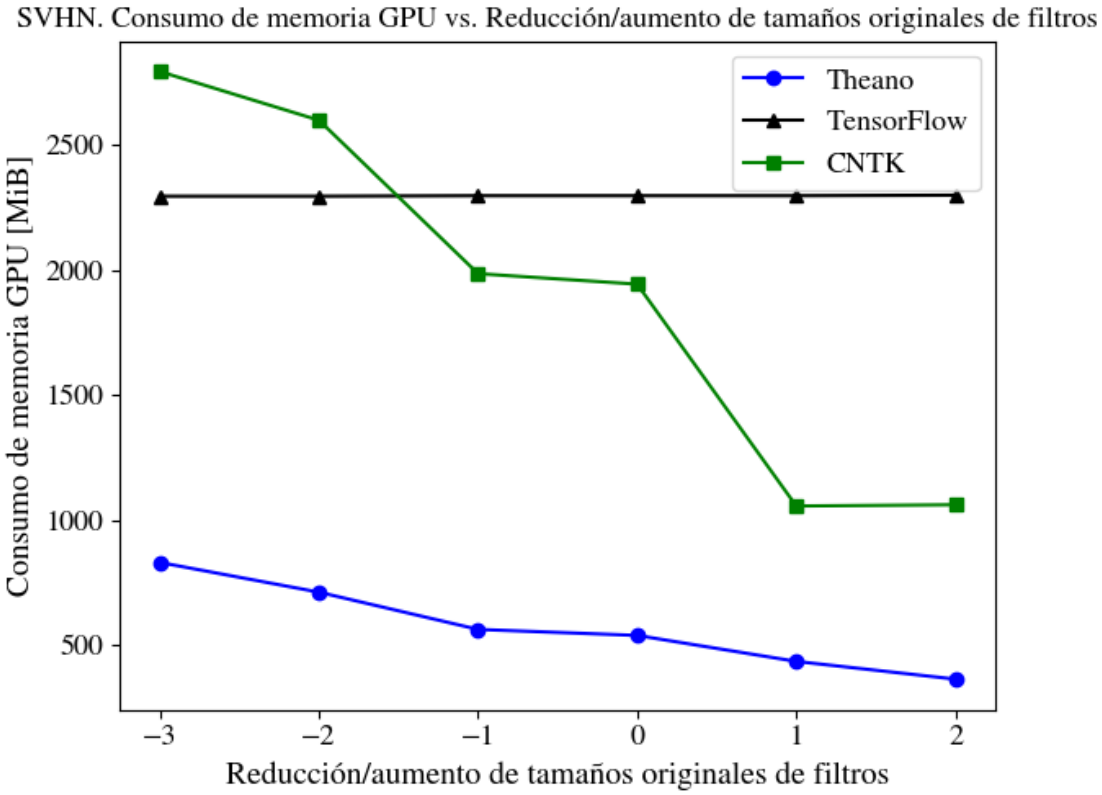


Figura 4.57: Variación de los tamaños de los filtros convolucionales de basic-SVHN y su impacto en el consumo de memoria GPU. Dataset: SVHN.

La figura 4.58 muestra que TensorFlow consume entre 2,8 y 6,3 veces más memoria que Theano. Además, CNTK consume entre 2,4 y 3,7 veces más memoria que Theano, donde el valor mínimo es alcanzado al aumentar las dimensiones de los filtros en una unidad, mientras que el máximo al aplicar una reducción de dos unidades. Por último, el uso relativo de memoria entre TensorFlow y CNTK varía entre 0,8 y 2,2; el valor máximo es obtenido al aumentar los tamaños de los filtros en una unidad.

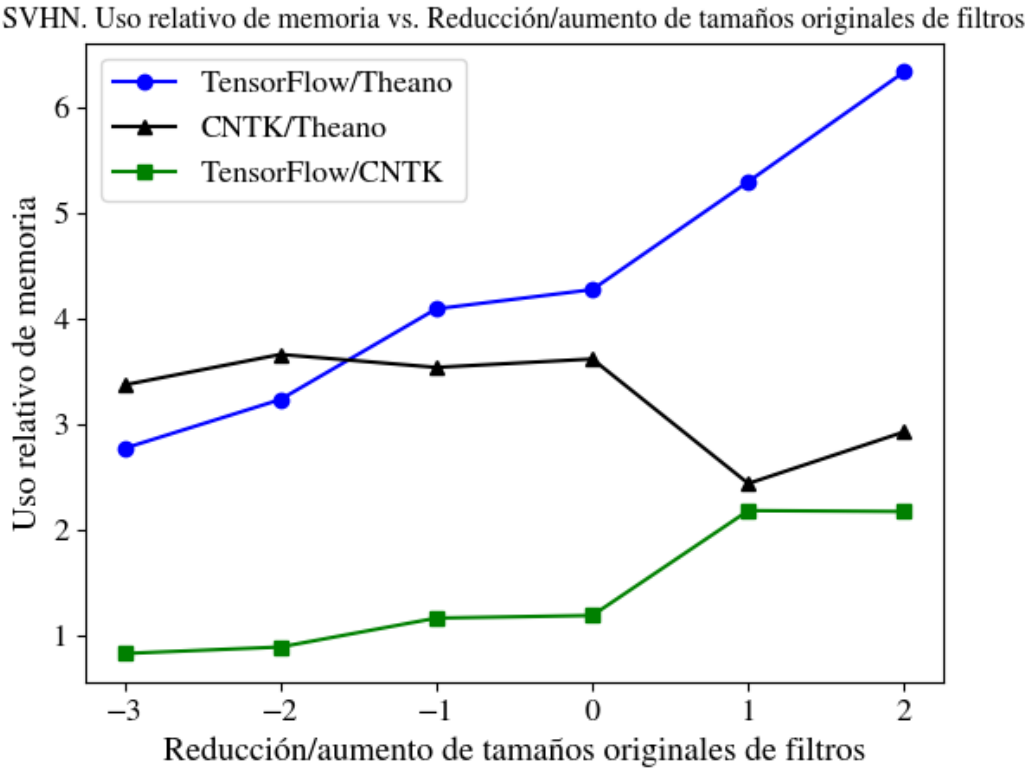


Figura 4.58: Razón entre los consumos de memoria GPU para cada par de frameworks en el entrenamiento de basic-SVHN como función de la variación en los tamaños de los filtros. Dataset: SVHN.

4.6.2.4. NORB

4.6.2.4.1. Arquitectura maestra

Se considera la red propuesta en [47] que hace uso de average pooling y ReLU. Para efectos prácticos, la red es denominada AP-ReLU. Los detalles de sus capas se entregan a continuación, en orden de aparición:

- Una capa convolucional con 64 filtros de 5 x 5.
- Una capa de average pooling con pooling size de 3 x 3.
- Una capa convolucional con 64 filtros de 5 x 5.
- Una capa de average pooling con pooling size de 3 x 3.

- Una capa convolucional con 64 filtros de 5 x 5.
- Una capa de average pooling con pooling size de 3 x 3.
- Una capa totalmente conectada con 500 neuronas y dropout de 0,5.
- Una capa totalmente conectada con 6 neuronas y función de activación softmax (capa de salida).

La arquitectura está conformada por 243.698 parámetros entrenables. Además, AP-ReLU obtiene un error de prueba del 5,30 %.

Si bien no se lleva a cabo pre-procesamiento, las imágenes son escaladas desde 108 x 108 x 2 píxeles a 48 x 48 x 2 píxeles. A continuación de cada una de las dos primeras capas de pooling hay una capa de LRN. Para toda capa oculta, se utiliza ReLU como función de activación. El modelo es entrenado con SGD como método de optimización durante 250 epochs, con tamaño de mini-batch 128, weight decay de 0,01 y momentum de 0,9. Se utiliza una tasa de aprendizaje inicial de 0,001 que es multiplicada por un factor de 0,1 cada 100 epochs.

En la tabla 4.25 se muestran los resultados obtenidos para esta configuración. TensorFlow logra el tiempo de entrenamiento total más bajo de todos, de manera que CNTK emplea 1,1 veces más tiempo que TensorFlow. Theano, por su parte, tardó 2,2 veces más que TensorFlow y 1,9 veces más que CNTK en cumplir la misma tarea. El consumo de memoria GPU es notablemente superior en TensorFlow, siendo 1,5 veces mayor al de CNTK y 2,7 veces mayor al de Theano. En lo que respecta a los errores, tanto en entrenamiento como en evaluación la diferencia más significativa se da entre TensorFlow y CNTK (0,21 % y 0,72 %, respectivamente).

Cuadro 4.25: Rendimiento sobre arquitectura AP-ReLU. Dataset: NORB.

Framework	**Tiempo de entrenamiento total [s]**	**Consumo de memoria GPU [MiB]**	**Error de entrenamiento [%]**	**Error de prueba [%]**
Theano	8573,7	475	3,20	9,85
TensorFlow	3875,8	1283	3,21	9,98
CNTK	4445,7	849	3,00	9,26

4.6.2.4.2. Arquitectura básica

Nuevamente se considera uno de los modelos estudiados en [47]. En este caso, aquel que hace uso de max pooling y de ReLU, y que será denominado MP-ReLU. La estructura de esta arquitectura es prácticamente idéntica a la de la red maestra, excepto por las capas de average pooling, que son reemplazadas por capas de max pooling. Las consideraciones tomadas respecto al pre-procesamiento de los datos, funciones de activación utilizadas, método de optimización, tamaño de mini-batch, cantidad de epochs de entrenamiento, tasa de aprendizaje, weight decay y momentum se mantienen con respecto a la arquitectura maestra. Este modelo alcanza un error de prueba del 5,88 % en la citada publicación.

Los resultados obtenidos por cada framework se muestran en la tabla 4.26. Dado a que los tiempos totales de entrenamiento y los consumos de memoria GPU sólo presentan pequeñas variaciones con respecto a la arquitectura maestra, los análisis en torno a estas variables son análogos a los efectuados sobre dicha arquitectura. En lo que respecta a los errores, la diferencia existente entre TensorFlow y Theano/CNTK es de un 0,02 % en entrenamiento. En la fase de evaluación, en cambio, la diferencia más significativa se produce entre Theano y CNTK (0,65 %).

Cuadro 4.26: Rendimiento sobre arquitectura MP-ReLU. Dataset: NORB.

Framework	**Tiempo de entrenamiento total [s]**	**Consumo de memoria GPU [MiB]**	**Error de entrenamiento [%]**	**Error de prueba [%]**
Theano	8584,8	475	0,08	7,84
TensorFlow	3721,3	1283	0,10	7,19
CNTK	4488,2	858	0,08	7,31

4.6.2.4.3. Variación de cantidad de ejemplos de entrenamiento

Como en secciones anteriores, se estudia el impacto de variar la cantidad de ejemplos de entrenamiento disponibles. Las fracciones y cantidades de datos consideradas se muestran en la tabla 4.27.

Cuadro 4.27: Fracciones de ejemplos usadas para el entrenamiento de MP-ReLU. Dataset: NORB.

Fracción de ejemplos [%]	**Número de ejemplos**
33	19.245
40	23.328
57	33.242
64	37.324
83	48.405
100	58.320

La figura 4.59 muestra que, por tercera ocasión, es TensorFlow quien alcanza el

menor tiempo total de entrenamiento en todo escenario posible, seguido de cerca por CNTK. Al igual que para datasets anteriores, TensorFlow y CNTK alcanzan tiempos comparables para fracciones pequeñas de ejemplos, pero las diferencias de tiempo son cada vez mayores conforme aumenta la fracción considerada. Como siempre, Theano alcanza el peor rendimiento y no escala bien con el aumento de la cantidad de datos de entrenamiento.

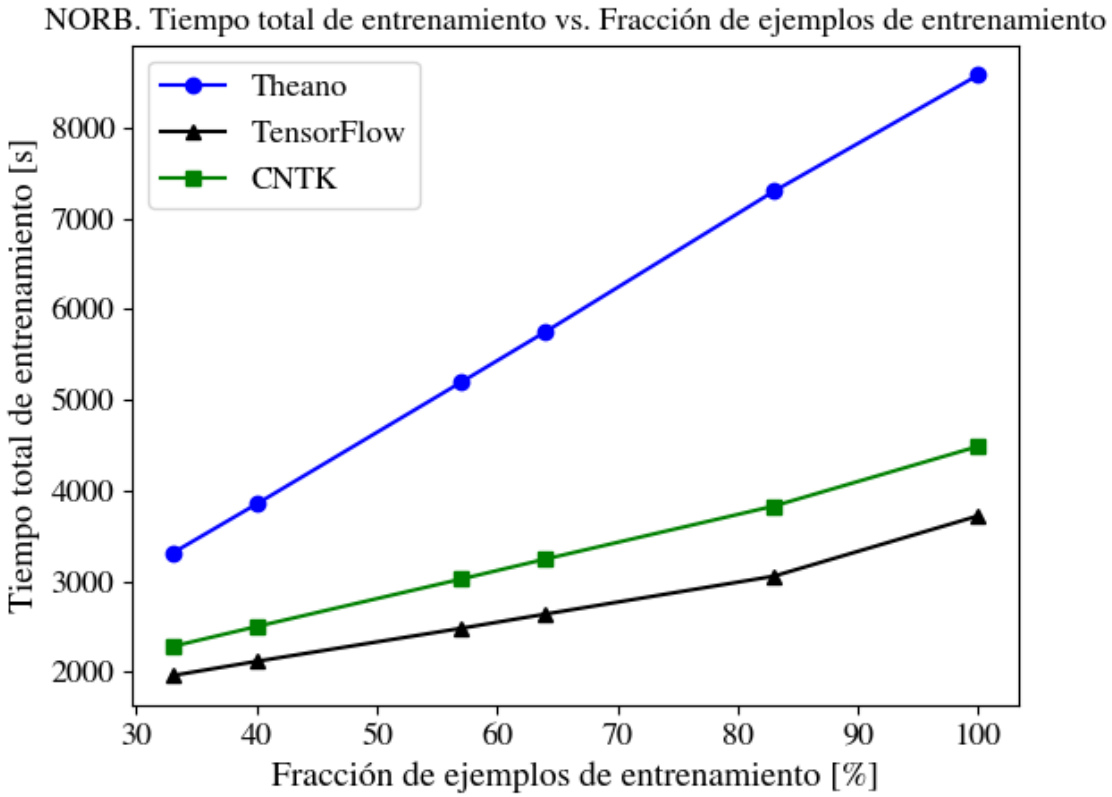


Figura 4.59: Variación de la fracción de ejemplos provistos para el entrenamiento de MP-ReLU y su impacto en el tiempo total de entrenamiento. Dataset: NORB.

La figura 4.60 muestra que TensorFlow y CNTK consiguen un aumento moderado de la cantidad de ejemplos de entrenamiento procesados por unidad de tiempo mientras mayor es la cantidad de ejemplos disponibles, siendo TensorFlow quien logra el mayor aumento. Theano, por otra parte, es quien procesa la menor cantidad de datos de entrenamiento por unidad de tiempo para toda fracción posible. Además, la evolución de esta métrica es prácticamente nula con el mencionado framework.

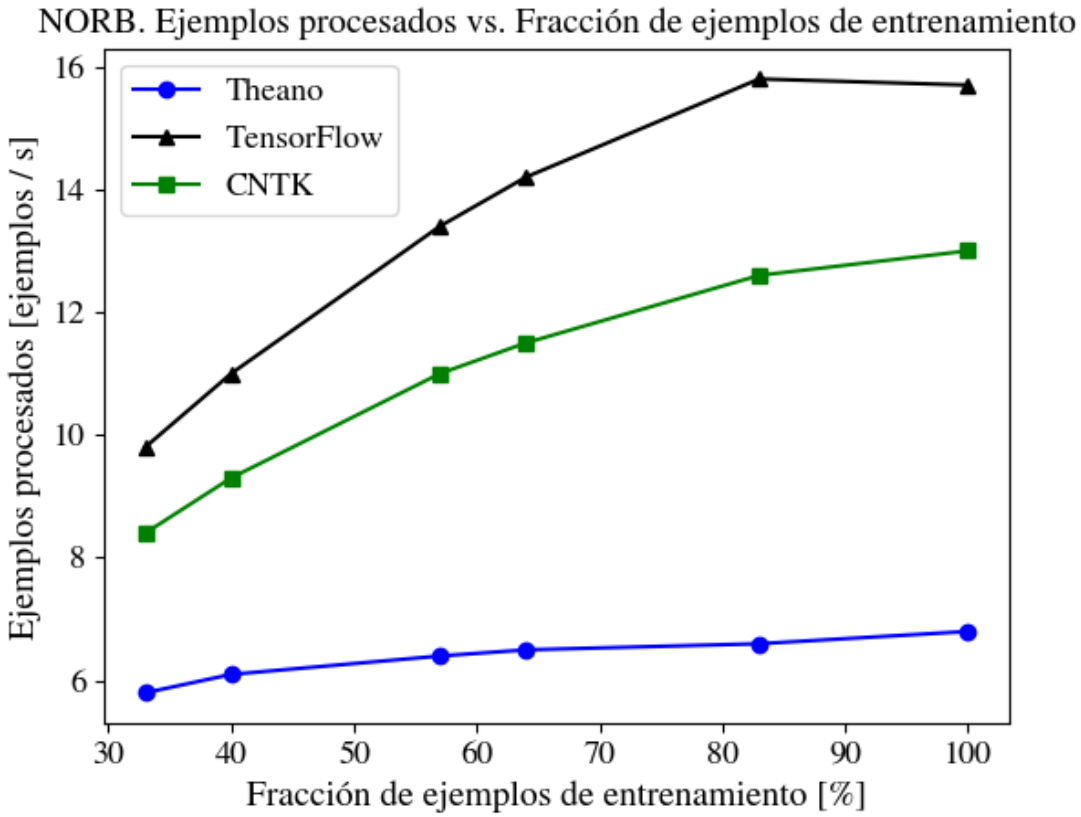


Figura 4.60: Variación de la fracción de ejemplos provistos para el entrenamiento de MP-ReLU y su impacto en la cantidad promedio de ejemplos procesados. Dataset: NORB.

De acuerdo a la figura 4.61, el speed-up relativo entre los tiempos de TensorFlow y CNTK varía entre 1,2 y 1,3, lo que implica que las proporciones existentes entre ambos frameworks se mantienen relativamente constantes. A diferencia de casos anteriores, las curvas TensorFlow/Theano y CNTK/Theano se comportan en forma similar a la función raíz cuadrada, de tal manera que en el primer caso las proporciones entre tiempos varían entre 1,5 y 1,9, mientras que en el segundo caso entre 1,7 y 2,4.

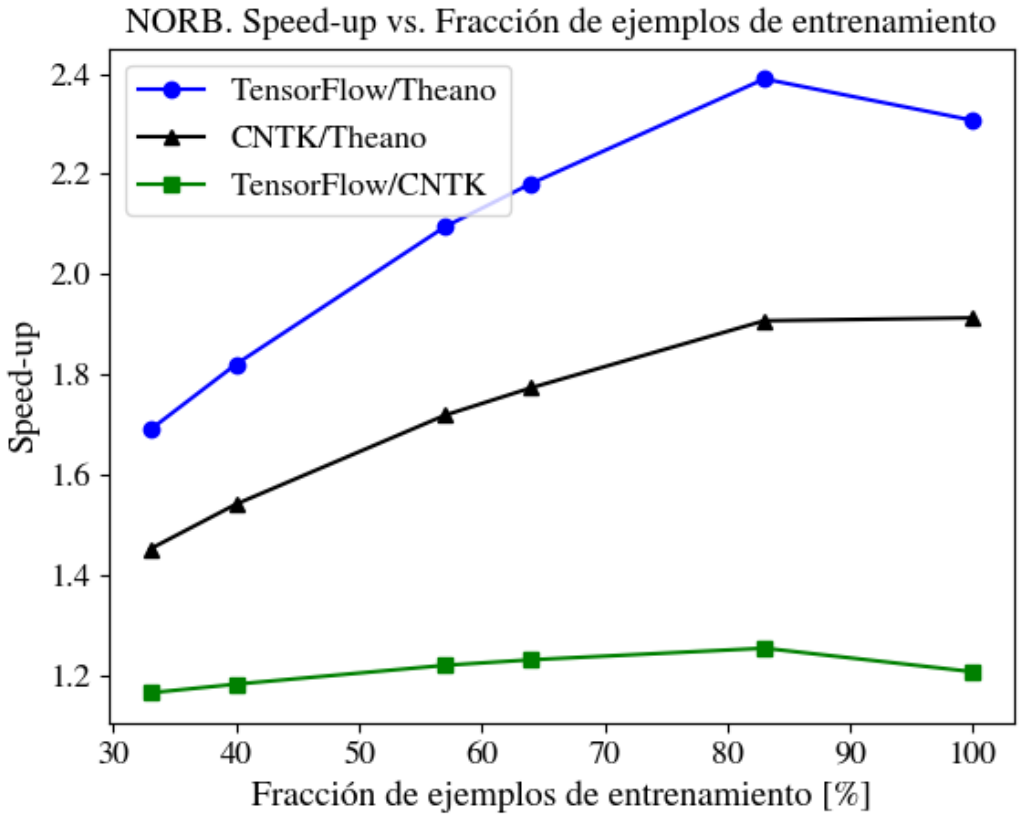


Figura 4.61: Speed-up relativo entre los tiempos de entrenamiento de MP-ReLU para cada par de frameworks en función de la fracción de ejemplos provistos. Dataset: NORB.

4.6.2.4.4. Variación de la cantidad de capas convolucionales

Tal como en secciones anteriores, se estudia el impacto de incrementar el número de capas convolucionales en MP-ReLU. En la tabla 4.28 se muestran algunas características de los modelos considerados para el análisis.

Cuadro 4.28: Aumento en la cantidad de capas convolucionales en MP-ReLU, junto con el número total de parámetros de los modelos resultantes. Dataset: NORB.

Factor de aumento	**Cantidad total de capas convolucionales**	**Cantidad total de parámetros**
1	3	243.698
2	6	551.090
3	9	858.482
4	12	1.165.874
5	15	1.473.266

La figura 4.62 da indicios de que, así como para MNIST y CIFAR-10, los tiempos totales de entrenamiento varían de forma lineal en cada framework, de tal manera que Theano presenta el peor rendimiento, mientras que TensorFlow el mejor, siendo muy cercano al de CNTK, si bien las curvas se distancian cada vez más al aumentar el factor de variación.

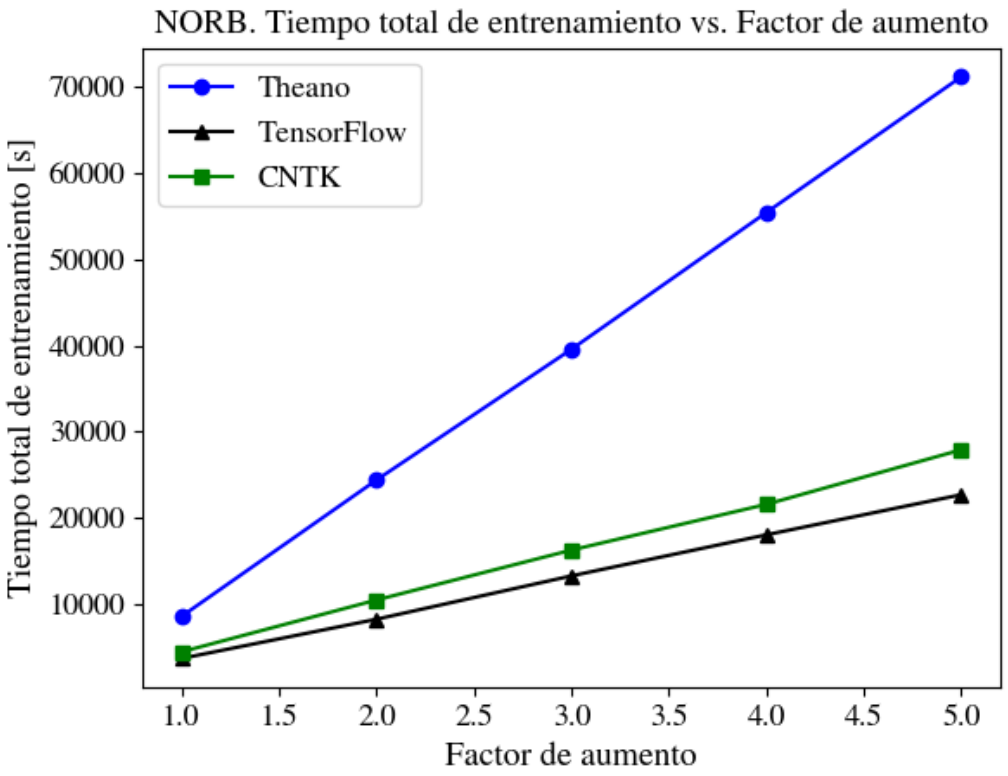


Figura 4.62: Incremento de la cantidad de capas convolucionales en MP-ReLU y su impacto en el tiempo total de entrenamiento. Dataset: NORB.

La figura 4.63 muestra que el speed-up relativo entre TensorFlow y CNTK varía entre 1,2 y 1,3. Por otra parte, CNTK presenta tiempos entre 1,9 y 2,6 veces más bajos que los de Theano, mientras que TensorFlow obtiene tiempos entre 2,3 y 3,1 veces menores que los de TensorFlow.

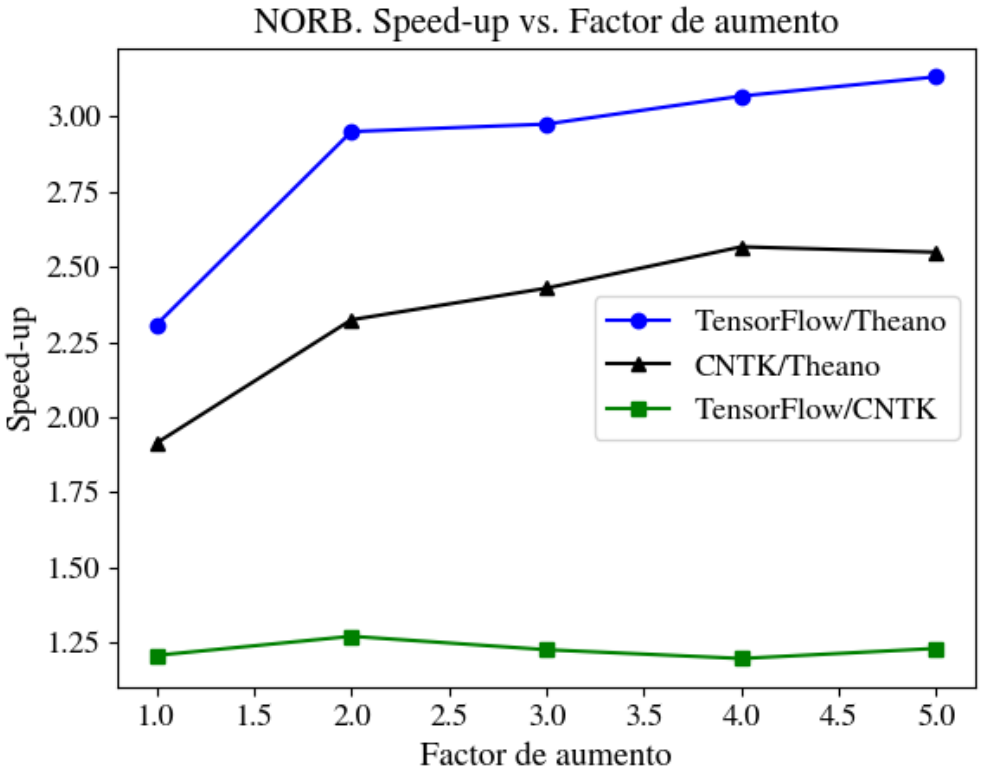


Figura 4.63: Speed-up relativo entre los tiempos de entrenamiento de MP-ReLU para cada par de frameworks en función del factor de aumento

. Dataset: NORB.

La figura 4.64 muestra que el consumo de memoria varía linealmente en Theano, mientras que en CNTK lo hace de forma cuasi-lineal y en TensorFlow se mantiene constante para todos los factores de aumento posibles. Además, la figura da cuenta de que este último framework hace uso de una cantidad de memoria ampliamente superior a la de los demás.

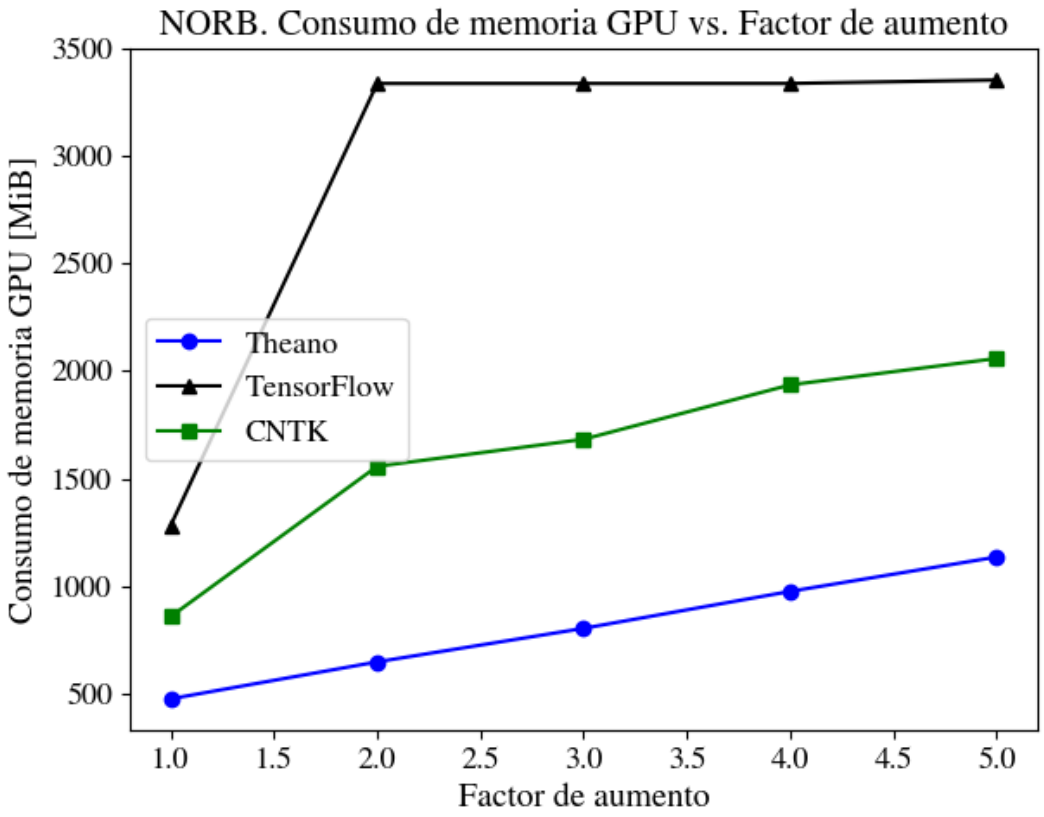


Figura 4.64: Incremento de la cantidad de capas convolucionales en MP-ReLU y su impacto en el consumo de memoria GPU. Dataset: NORB.

La figura 4.65 muestra que TensorFlow llega a utilizar 5,2 veces más memoria GPU que Theano con un factor de 2. Sin embargo, el consumo relativo entre ambos frameworks decrece con cada conjunto adicional de capas que se agrega. Además, el valor mínimo de dicho cociente, que es de 2,7, se alcanza con la arquitectura original. Por otra parte, TensorFlow sobrepasa entre 1,5 y 2,1 veces el consumo de CNTK, así como también este último sobrepasa entre 3,4 y 5,3 veces el de Theano.

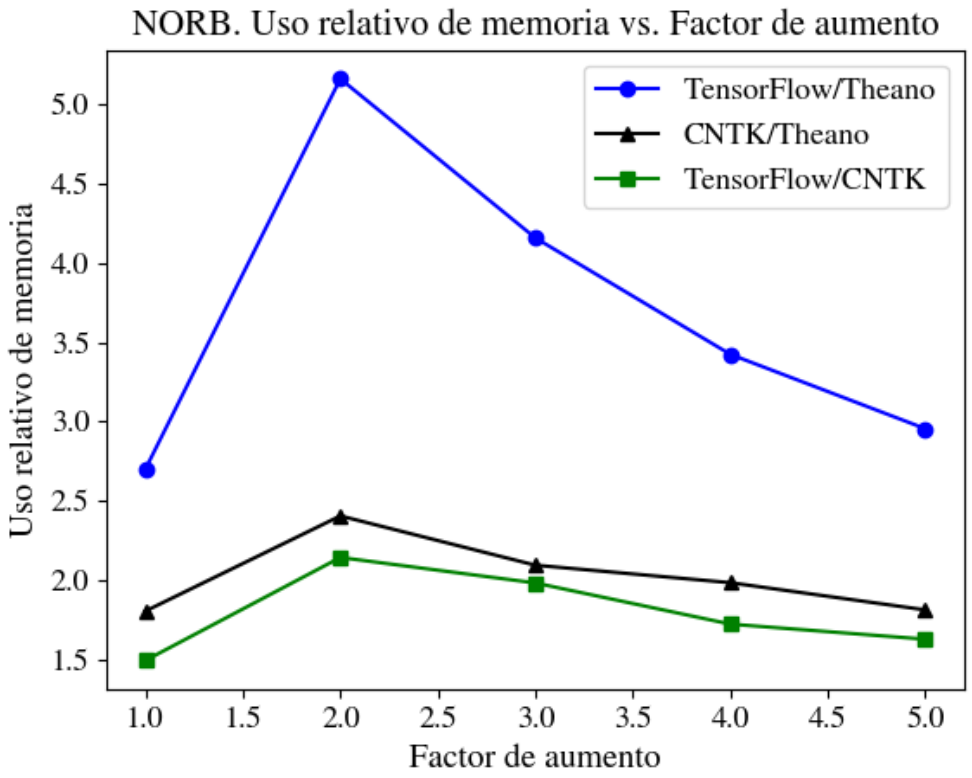


Figura 4.65: Uso relativo de memoria GPU entre cada par de frameworks en el entrenamiento de MP-ReLU como función del factor de aumento. Dataset: NORB.

4.6.2.4.5. Variación de la cantidad de filtros convolucionales

Como en secciones anteriores, se varía la cantidad de filtros convolucionales de MP-ReLU. Los detalles de cada variación son visibles en la tabla 4.29.

Cuadro 4.29: Cantidad de filtros y parámetros para cada variación estudiada de MP-ReLU, donde C_i corresponde a la capa convolucional i de la red. Dataset: NORB.

Factor de variación	**Número de filtros C1**	**Número de filtros C2**	**Número de filtros C3**	**Cantidad total de parámetros**
$\frac{1}{4}$	16	16	16	25.154
$\frac{1}{2}$	32	32	32	72.402
1	64	64	64	243.698
2	128	128	128	893.490
4	256	256	256	3.421.874

En la figura 4.66 se aprecia que el tiempo total de entrenamiento de la red cambia linealmente en función de la variación para TensorFlow y CNTK, mientras que para Theano el cambio se desarrolla en forma súper lineal. Los rendimientos de TensorFlow y CNTK son muy similares, si bien la brecha entre ambos comienza a acrecentarse con el aumento de filtros en la red, siendo el primero el que alcanza el mejor rendimiento en todos los casos, excepto cuando la cantidad de filtros es reducida a la cuarta parte en cada capa convolucional.

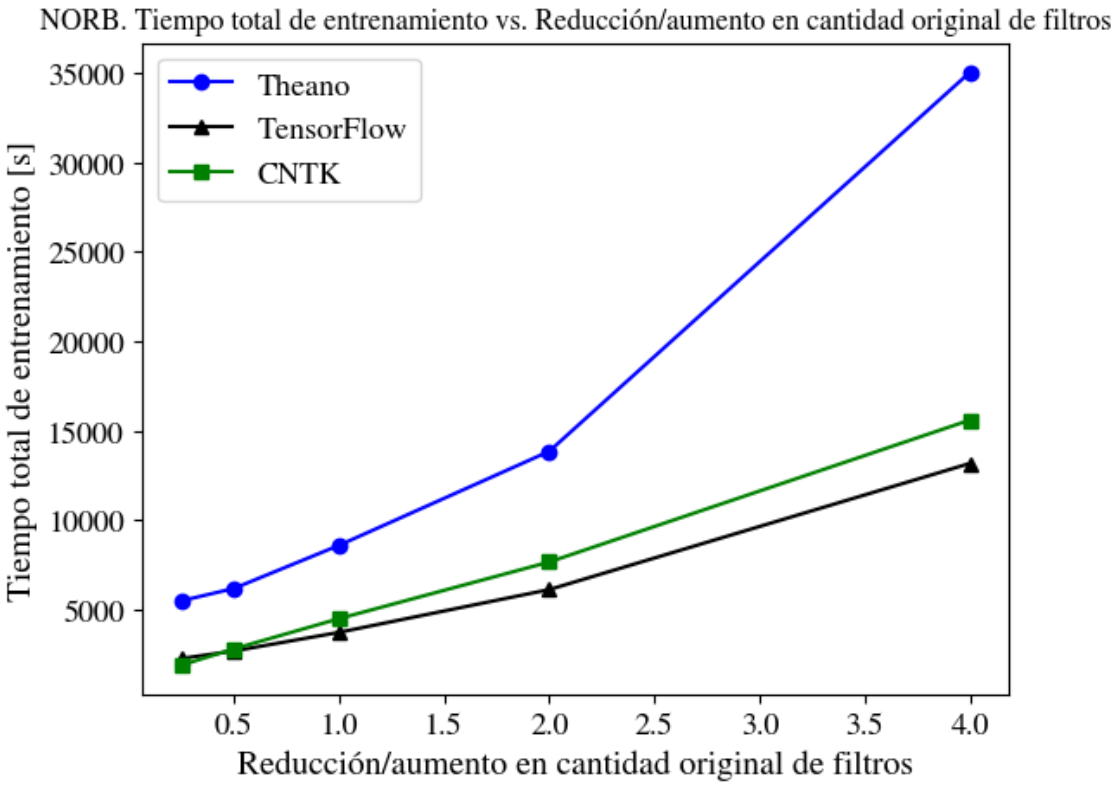


Figura 4.66: Variación de la cantidad de filtros convolucionales en MP-ReLU y su impacto en el tiempo total de entrenamiento. Dataset: NORB.

En la figura 4.67 se aprecia que el speed-up relativo entre TensorFlow y Theano tiende a mantenerse constante, si bien aumenta levemente al cuadriplicar el número de filtros, de tal manera que el primero es entre 2,3 y 2,6 veces más rápido que el segundo con factores de aumento de 2 y 4, respectivamente. La relación entre los tiempos de CNTK y Theano, en cambio, desciende hasta alcanzar su valor mínimo con el doble de filtros, para luego aumentar y alcanzar su valor máximo con el cuádruple de ellos; así, CNTK es entre 1,8 y 2,9 más rápido que Theano. Finalmente, el speed-up relativo entre TensorFlow y CNTK crece hasta que se alcanza la cantidad original de filtros, para luego presentar variaciones mínimas en los experimentos restantes; el valor mínimo de la relación es de 0,8, mientras que el máximo de 1,25.

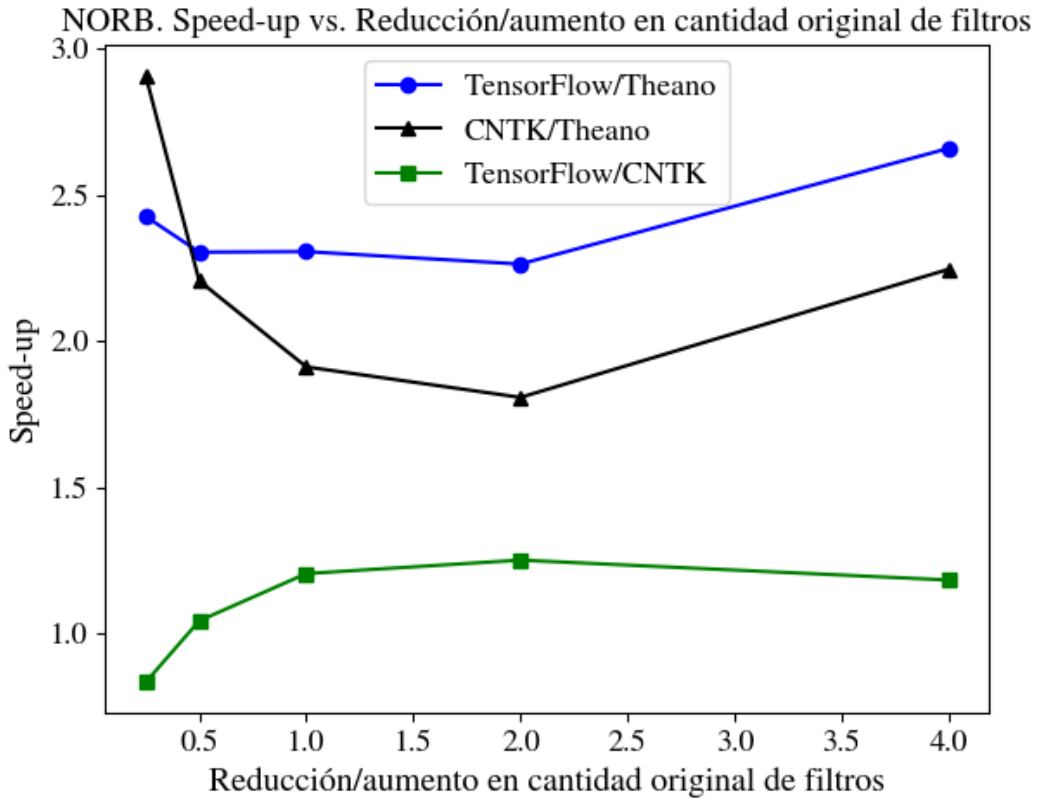


Figura 4.67: Speed-up relativo entre los tiempos de entrenamiento de MP-ReLU de cada par de frameworks en función de la variación en la cantidad de filtros. Dataset: NORB.

La figura 4.68 muestra que el consumo de memoria GPU cambia en forma lineal con el factor de variación en cada framework. Notar que es la primera vez que este comportamiento es observado en TensorFlow y en CNTK. Al igual que en datasets anteriores, los consumos son muy similares al reducir el número original de filtros, pero son muy diferentes entre sí al aumentarlo.

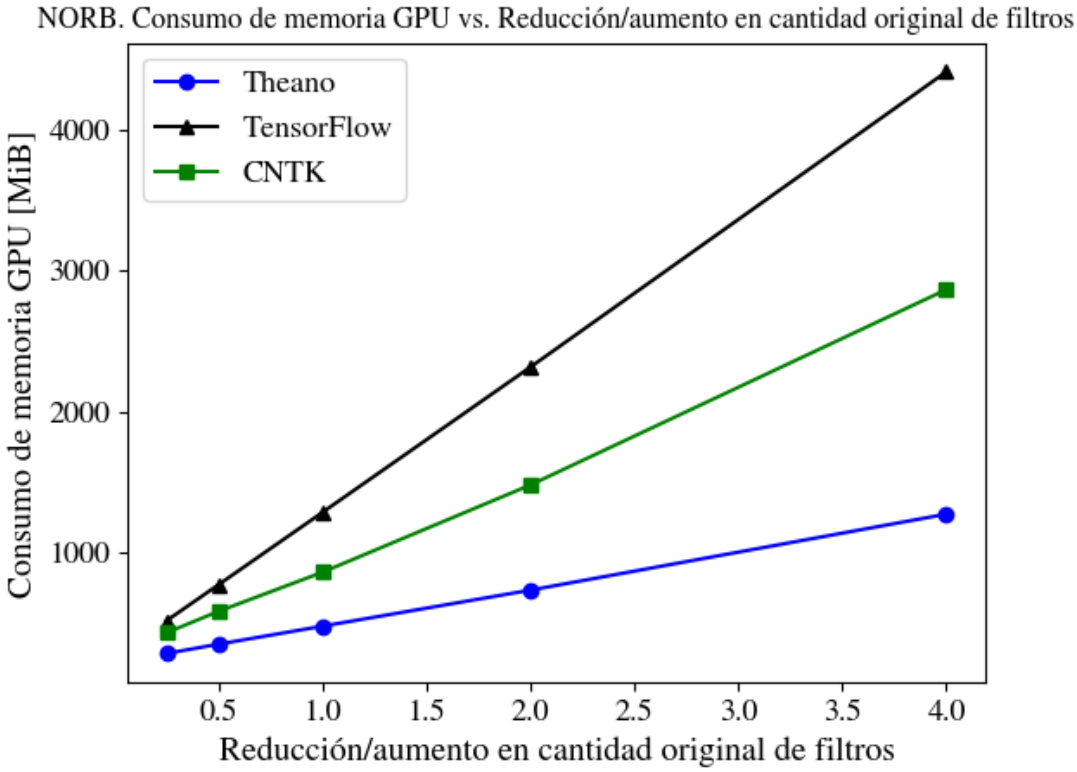


Figura 4.68: Variación de la cantidad de filtros convolucionales de MP-ReLU y su impacto en el consumo de memoria GPU. Dataset: NORB.

En la figura 4.69 es posible apreciar que TensorFlow consume entre 1,8 y 3,5 veces más memoria que Theano. CNTK, por su parte, utiliza entre 1,5 y 2,3 veces más memoria que Theano. Finalmente, TensorFlow ocupa entre 1,2 y 1,5 veces más memoria que CNTK.

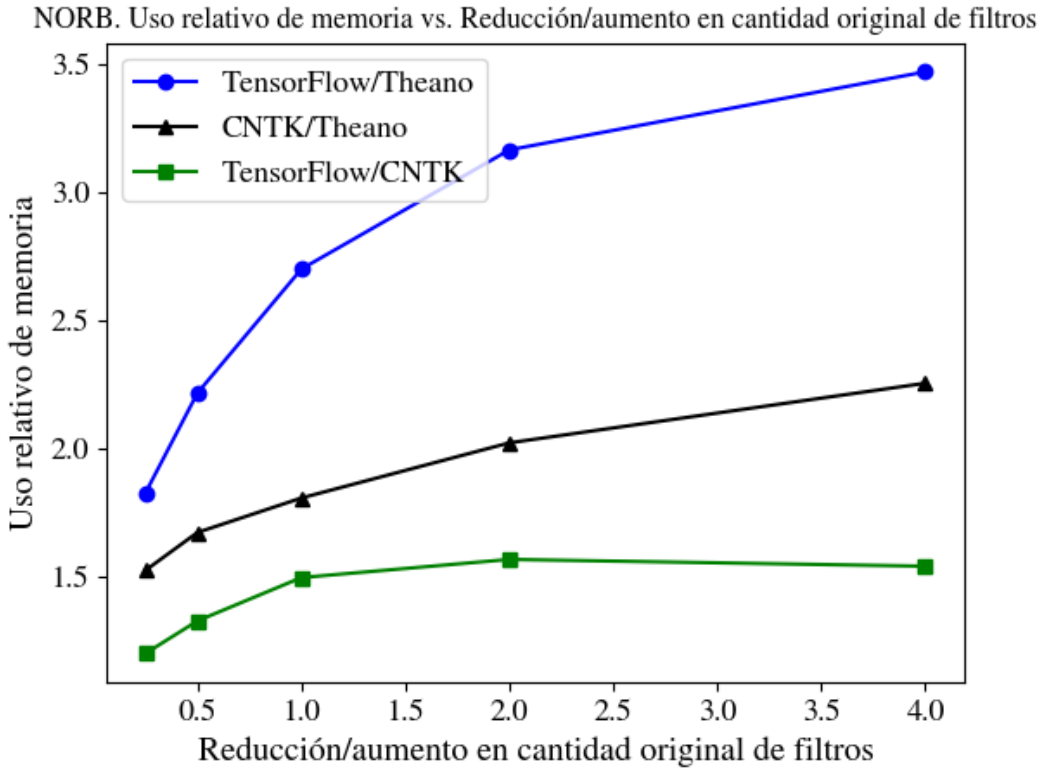


Figura 4.69: Uso relativo de memoria GPU entre cada par de frameworks en el entrenamiento de MP-ReLU como función de la variación en la cantidad de filtros. Dataset: NORB.

4.6.2.4.6. Variación de tamaños de filtros convolucionales

Como en secciones previas, se altera el tamaño de los filtros convolucionales en MP-ReLU. La tabla 4.30 muestra las configuraciones que se estudian. Notar que para evitar tanto la reducción prematura de las dimensiones de los inputs a procesar como la necesidad de aplicar padding asimétrico, se han duplicado las dimensiones de las imágenes originales, se ha reducido el pooling size de las capas de pooling de 3 x 3 a 2 x 2 y se ha removido el uso de padding.

Cuadro 4.30: Disminución / incremento en el tamaño de los filtros de las capas convolucionales de MP-ReLU, junto con la cantidad de parámetros resultantes. C_i representa la capa convolucional i. Dataset: NORB.

Cambio en tamaños de filtros	**Tamaño filtros C1**	**Tamaño filtros C2**	**Tamaño filtros C3**	**Cantidad total de parámetros**
-3	2 x 2	2 x 2	2 x 2	1.604.978
-2	3 x 3	3 x 3	3 x 3	1.230.578
-1	4 x 4	4 x 4	4 x 4	936.818
0	5 x 5	5 x 5	5 x 5	723.698
1	6 x 6	6 x 6	6 x 6	591.218
2	7 x 7	7 x 7	7 x 7	699.378

En la figura 4.66 se muestra que el tiempo total de entrenamiento se mantiene relativamente constante en todos los frameworks, excepto en Theano, donde el tiempo aumenta considerablemente al incrementar los tamaños de los filtros. No obstante, la tendencia en CNTK es a disminuir los tiempos, mientras que en TensorFlow a aumentarlos. Se observa, además, que TensorFlow obtiene los tiempo más bajos en todo escenario.

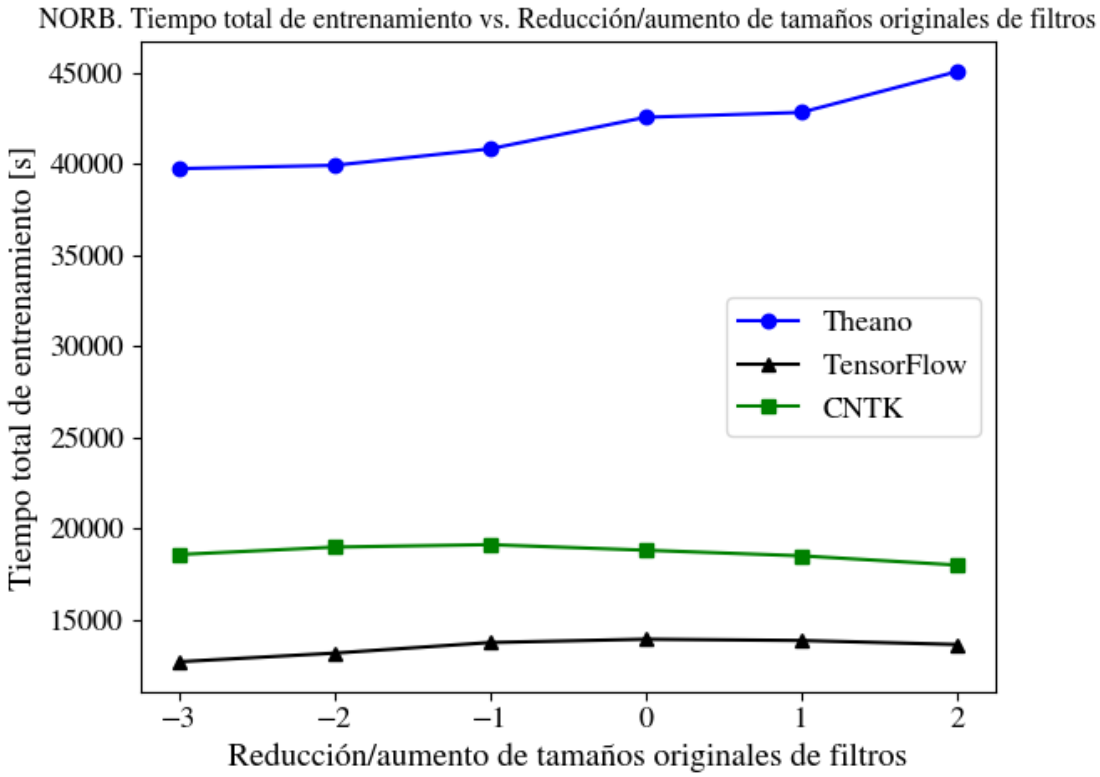


Figura 4.70: Variación de los tamaños de los filtros convolucionales de MP-ReLU y su impacto en el tiempo total de entrenamiento. Dataset: NORB.

De acuerdo a la figura 4.71, el speed-up relativo entre Theano y TensorFlow varía entre 3 y 3,3, donde el valor mínimo es alcanzado al reducir las dimensiones de los filtros en dos unidades, mientras que el máximo al aumentar los tamaños en tres unidades. Además, el speed-up relativo entre CNTK y Theano varía entre 2,1 y 2,5, donde los valores mínimo y máximo se producen en las mismas circunstancias que para la relación TensorFlow-Theano. Por otra parte, el speed-up relativo entre TensorFlow y CNTK varía entre 1,3 y 1,5.

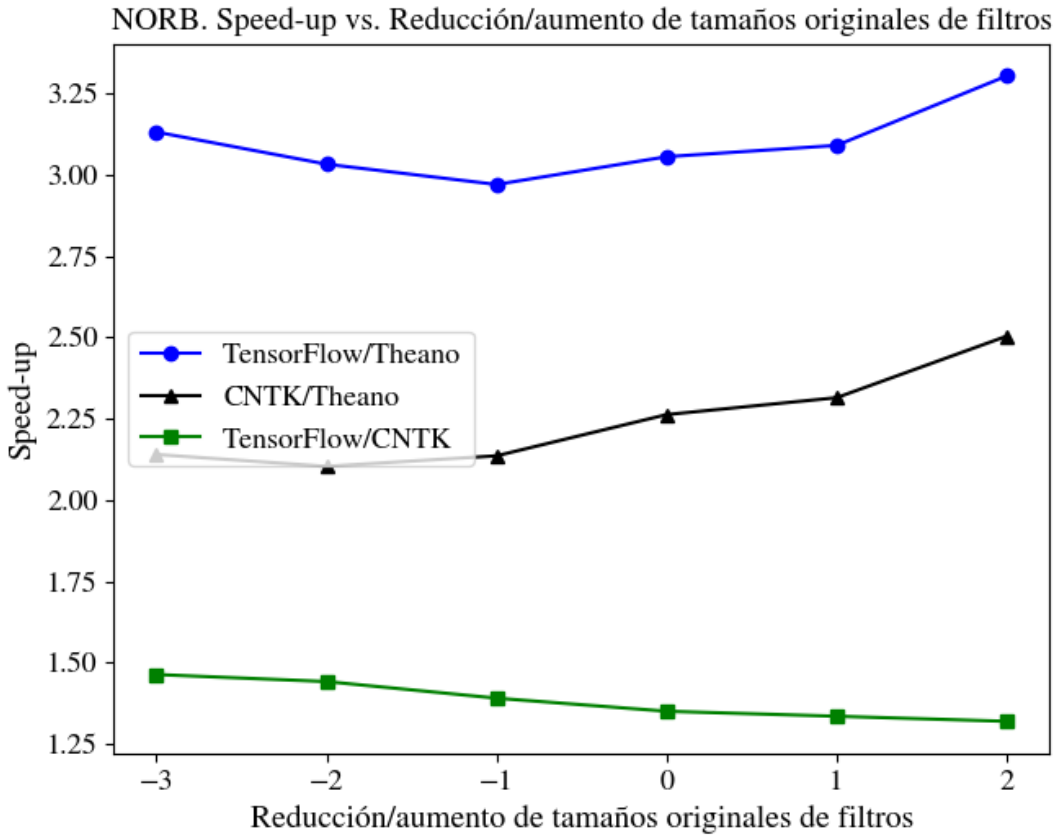


Figura 4.71: Speed-up relativo entre los tiempos de entrenamiento de MP-ReLU de cada par de frameworks en función de la variación en el tamaño de los filtros. Dataset: NORB.

La figura 4.72 muestra que el uso de memoria GPU es prácticamente constante en TensorFlow, excepto al aumentar las dimensiones de los filtros en dos unidades, donde se produce un abrupto aumento en el consumo. En CNTK, por otro lado, el consumo varía en forma bastante irregular, casi oscilatoria, alcanzando su valor máximo al aumentar las dimensiones de los filtros en una unidad. Por otra parte, el nivel de utilización del recurso decrece linealmente en Theano, aunque a un ritmo mucho más lento que en datasets anteriores.

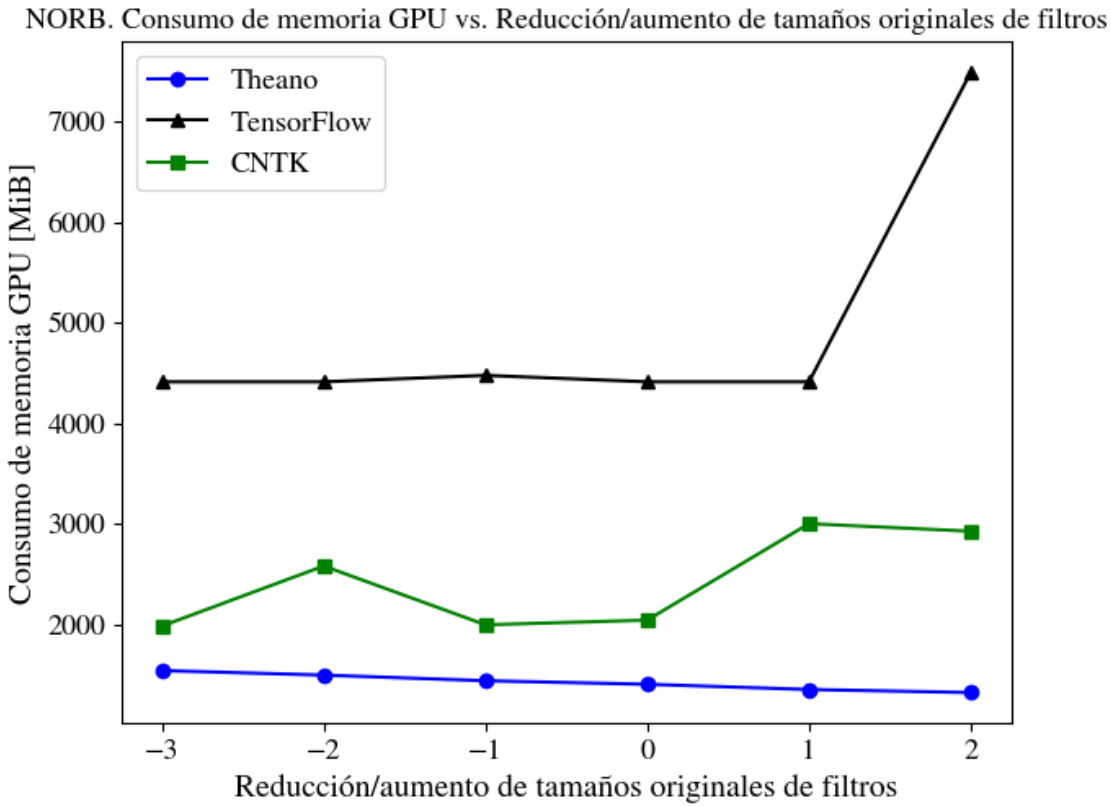


Figura 4.72: Variación de los tamaños de los filtros convolucionales de MP-ReLU y su impacto en el consumo de memoria GPU. Dataset: NORB.

En la figura 4.73 se muestra que TensorFlow utiliza entre 2,9 y 5,6 veces más memoria que Theano. Además, el consumo relativo entre CNTK y Theano varía entre 1,3, al reducir las dimensiones de los filtros en tres unidades, y 2,2, al aumentar dichas dimensiones en una unidad. Finalmente, el uso relativo de memoria entre TensorFlow y CNTK varía entre 1,5 y 2,5, valores que se presentan al aumentar las dimensiones en una y dos unidades, respectivamente.

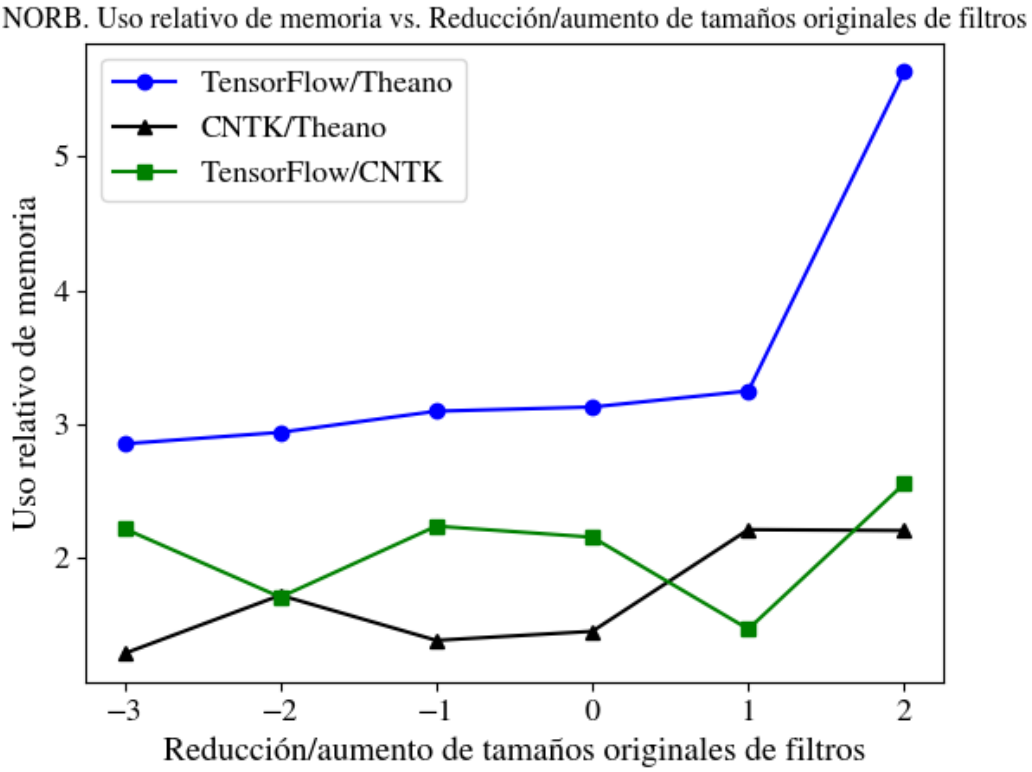


Figura 4.73: Razón entre los consumos de memoria GPU para cada par de frameworks en el entrenamiento de MP-ReLU como función de la variación de la cantidad de filtros. Dataset: NORB.

4.6.3. Redes recurrentes

Para el estudio de redes recurrentes, se han considerado los datasets *How Much Did It Rain? II* e *IMDB Movie reviews*, ya descritos anteriormente. Por consistencia, los criterios aplicados para la selección de los modelos corresponden a los mismos considerados para la elección de las arquitecturas convolucionales.

4.6.3.1. IMDB Movie reviews

Dada la imposibilidad de encontrar publicaciones científicas que entregaran todas las características necesarias para el diseño y posterior entrenamiento de los modelos descritos en ellas, se ha considerado la arquitectura propuesta por Jason Brownlee, co-

rrespondiente a una red LSTM y que para efectos prácticos será denominada IMDB-LSTM.

Cada reseña ha sido reemplazada por una lista que indica la frecuencia de aparición de cada palabra que la integra en el dataset completo. Por ejemplo, si la reseña *"great movie"* se convierte en la lista [7, 5] quiere decir que las palabras *great* y *movie* corresponden a la séptima y quinta palabras más frecuentes del dataset, respectivamente. A continuación, cada palabra es convertida en un vector real de tamaño 32, técnica conocida como *word embedding*. Se han considerado sólo las 5.000 palabras más frecuentes del dataset, de manera que los términos presentes en una reseña, pero que no se encuentran dentro de este subconjunto son reemplazadas por el valor cero; esta cifra equivale al tamaño del vocabulario considerado. Además, el tamaño de cada reseña es limitado a 500 palabras, de manera que las que exceden esta cifra son "truncadas", mientras que las que poseen una cantidad inferior son rellenadas con tantos ceros como sean necesarios para cumplir con el límite establecido.

De esta manera, la arquitectura implementada se detalla a continuación:

- Una capa de embedding, con tamaño de vocabulario 5.000, largo de secuencia 500 y tamaño de embedding 32.
- Una capa LSTM de 100 celdas.
- Una capa totalmente conectada de 1 neurona (capa de salida).

Gracias al modelo anteriormente descrito, el autor obtuvo un error de clasificación de prueba de 12,21 %.

Los pesos de la capa de embedding han sido inicializados por medio de una distribución uniforme, mientras que los pesos no recurrentes y recurrentes de la capa LSTM a través de una distribución uniforme de Glorot y una matriz ortogonal aleatoria, respectivamente. Los pesos de la capa totalmente conectada han sido inicializados por medio de una distribución uniforme de Glorot. Además, en la capa LSTM se ha aplicado la función tanh como función de activación no recurrente y hard sigmoid como función de activación recurrente, mientras que en la capa de salida la función sigmoide. En lo que respecta al entrenamiento del modelo, se ha considerado binary crossentropy como función de pérdida, Adam como método de optimización, tasa de aprendizaje 0,001, tamaño de mini-batch 64 y sólo 3 epochs, ya que el entrenamiento converge rápidamente.

4.6.3.1.1. Variación de la cantidad de capas LSTM

De forma análoga al caso de las redes convolucionales, se procede a variar la profundidad de IMDB-LSTM aumentando el número de capas LSTM entre 1 y 5 con el fin de determinar el impacto sobre el rendimiento de los frameworks. La tabla 4.31 exhibe las características de las distintas arquitecturas estudiadas.

Cuadro 4.31: Variación del número de capas LSTM en IMDB-LSTM, junto a la cantidad total de parámetros de la arquitectura resultante. Dataset: IMDB Movie Reviews.

Número de capas LSTM	Número total de parámetros en la red
1	213.301
2	293.701
3	374.101
4	454.501
5	534.901

La figura 4.74 muestra que los tiempos de Theano y CNTK son muy similares entre sí, de tal manera que el segundo resulta ser más rápido que el primero en todos los casos, excepto al triplicar el número de capas, donde Theano entrena el modelo en un tiempo levemente inferior al de CNTK. No obstante, es posible observar que los tiempos de ambos comienzan a distanciarse cada vez más a partir del mencionado experimento. Por otra parte, los tiempos de TensorFlow son notablemente superiores a los de los frameworks restantes en todos los casos estudiados. En general, puede decirse que los tiempos aumentan linealmente al añadir más capas LSTM.

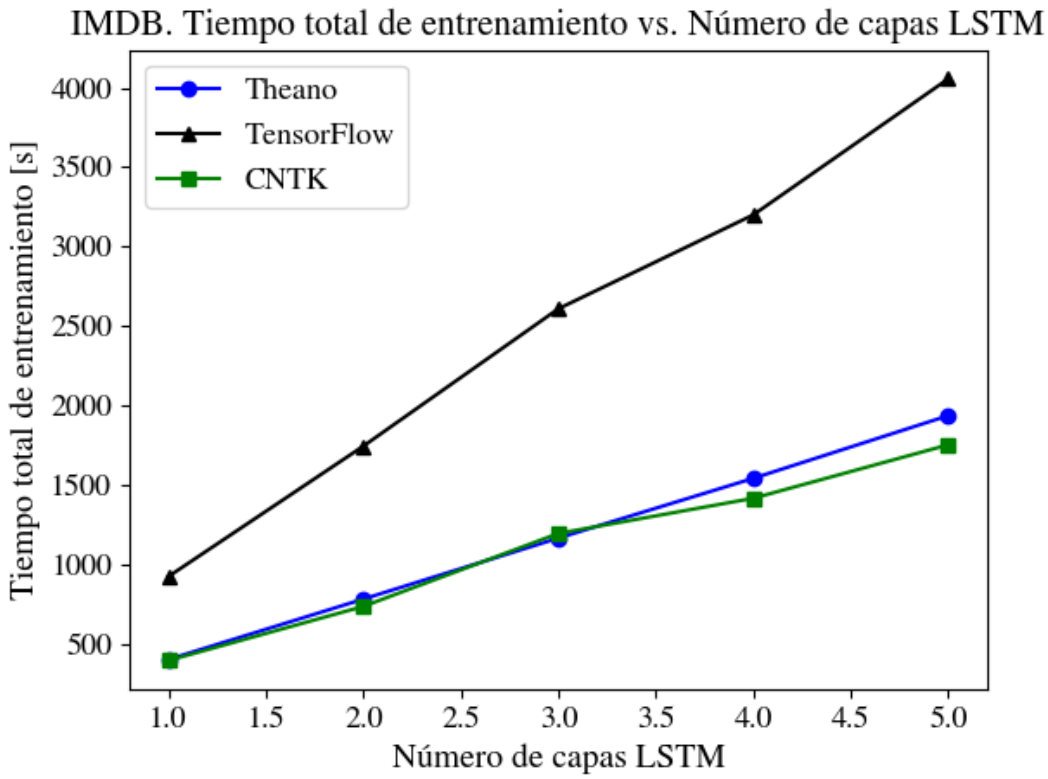


Figura 4.74: Aumento de la cantidad de capas LSTM en IMDB-LSTM y su impacto en el tiempo total de entrenamiento. Dataset: IMDB Movie Reviews.

Según muestra la figura 4.75, el speed-up relativo entre Theano y CNTK varía entre entre 0,98 y 1,1, reafirmando la cercanía entre los tiempos de ambos. Tal como se señaló antes, el valor mínimo de dicho speed-up se da al triplicar el número de capas LSTM, mientras que el máximo, al quintuplicar dicho número. Por otro lado, el speed-up relativo entre Theano y TensorFlow varía entre 2,1 (al cuadruplicar y al quintuplicar el número de capas) y 2,3 (al mantener la cantidad de capas intacta), mientras que el speed-up relativo entre CNTK y TensorFlow varía entre 2,2 (al triplicar las capas) y 2,4 (al duplicar las capas).

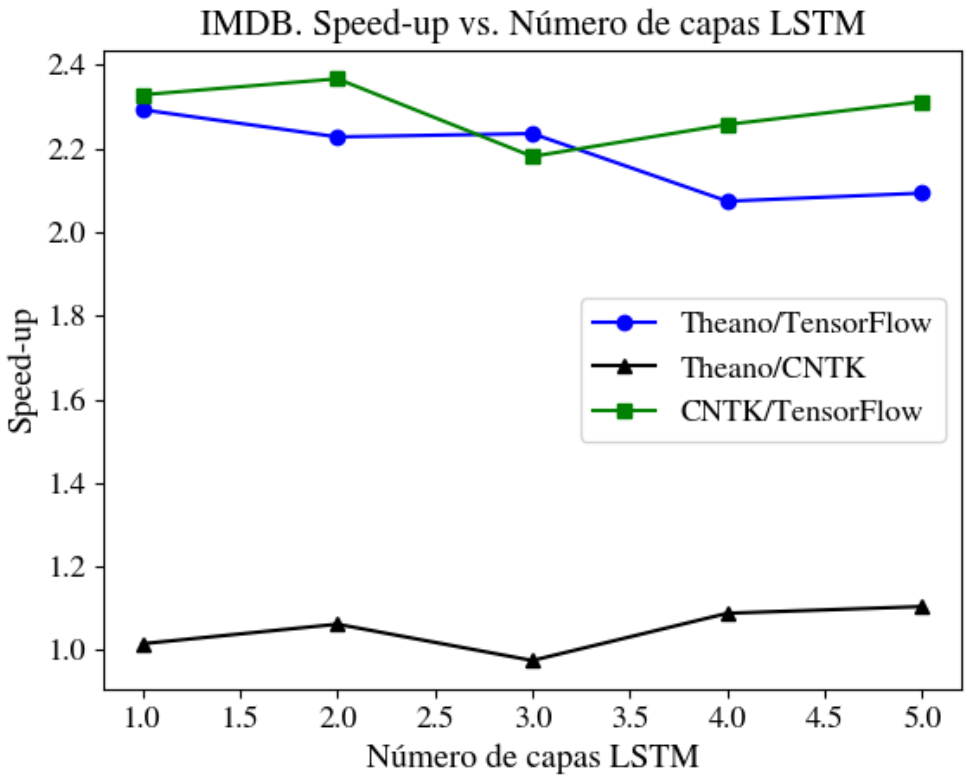


Figura 4.75: Speed-up relativo entre los tiempos de entrenamiento de IMDB-LSTM de cada par de frameworks en función del número de capas LSTM en la red. Dataset: IMDB Movie Reviews.

De acuerdo a la figura 4.76, el consumo de memoria GPU varía de diferentes formas en cada framework. En CNTK, quien en todo escenario hace la mayor utilización del recurso, el consumo aumenta linealmente con el número de capas LSTM. En TensorFlow, el consumo aumenta linealmente hasta que se duplican las capas, para luego mantenerse constante al triplicarlas y luego repetir el comportamiento anterior al cuadruplicar y quintuplicar la cantidad de capas. En Theano, el consumo también aumenta linealmente, aunque de forma mucho más lenta que en CNTK.

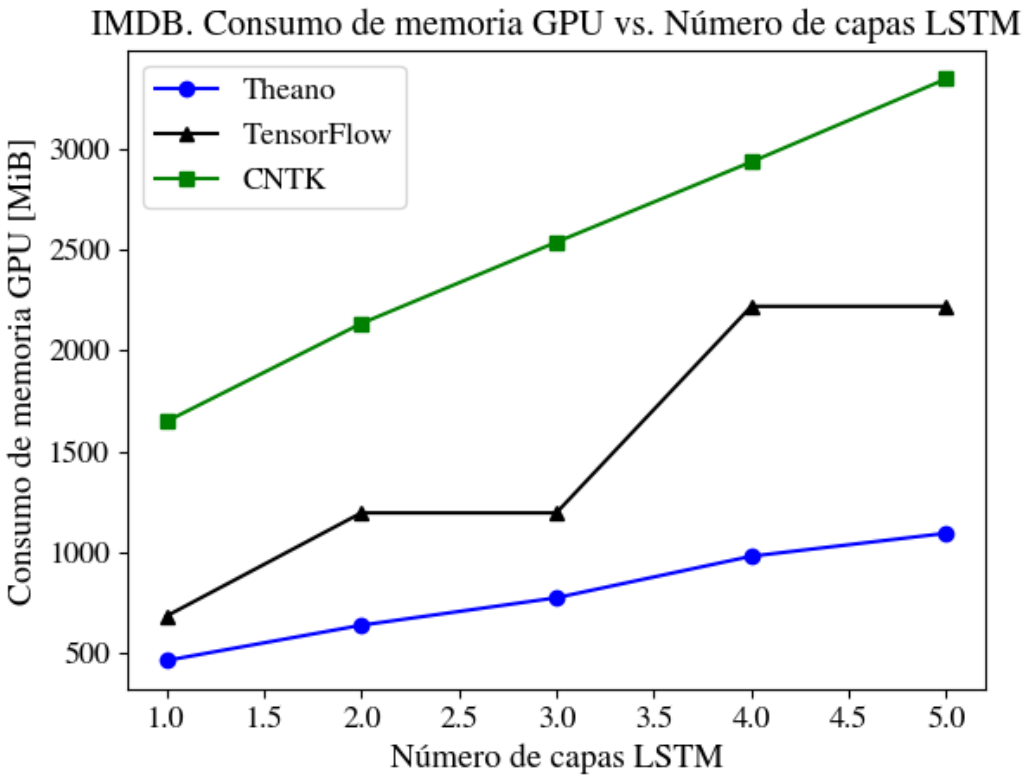


Figura 4.76: Aumento de la cantidad de capas LSTM en IMDB-LSTM y su impacto en el consumo de memoria GPU. Dataset: IMDB Movie Reviews.

La figura 4.77 muestra que la utilización de memoria GPU en CNTK es entre 3 y 3,6 veces superior a la de Theano. Por otra parte, CNTK consume entre 1,3 y 2,4 veces más memoria que TensorFlow, siendo el uso relativo mínimo al cuadruplicar el número de capas y máxima al no hacer cambios en la red. Finalmente, la razón entre los consumos de TensorFlow y Theano varía entre 1,5 (al no alterar el número de capas) y 2,3 (al cuadruplicarlo).

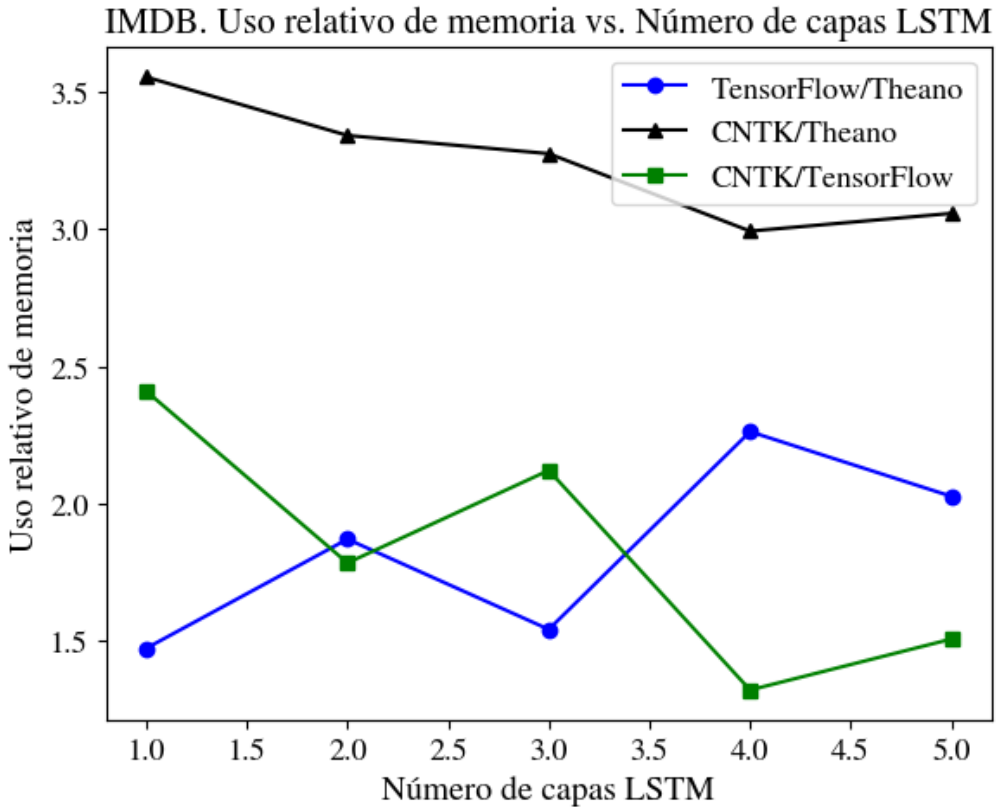


Figura 4.77: Uso relativo de memoria GPU entre cada par de frameworks en el entrenamiento de IMDB-LSTM como función del número de capas LSTM en la red. Dataset: IMDB Movie Reviews.

4.6.3.1.2. Variación de la cantidad de celdas LSTM

En esta sección se varía el número de celdas LSTM en IMDB-LSTM manteniendo la cantidad de capas ocultas intacta. El número de celdas es reducido a la cuarta parte y a la mitad, así como también es aumentando al doble, al cuádruple y al óctuple. La tabla 4.32 expone los detalles de cada arquitectura.

Cuadro 4.32: Variación del número de celdas LSTM en IMDB-LSTM, junto a la cantidad total de parámetros de la arquitectura resultante. Dataset: IMDB Movie Reviews.

Número de celdas LSTM	**Número total de parámetros en la red**
25	165.826
50	176.651
100	213.301
200	346.601
400	853.201
800	2.826.401

La variación del número de celdas LSTM provoca que los tiempos de entrenamiento cambien de formas muy disímiles entre frameworks. Según muestra la figura 4.78, estos aumentan súper linealmente en Theano y CNTK. Se observa que al reducir el número de celdas con respecto a la cantidad original, Theano obtiene tiempos más bajos que CNTK, pero al aumentarlo, los roles se invierten en forma notable, de manera que los tiempos del primero son cada vez más altos que los del segundo, si bien en CNTK no fue posible aumentar la cantidad de celdas en un factor de ocho, nuevamente por requerir más memoria de la disponible. Por otro lado, los tiempos de TensorFlow aumentan muy poco con el número de celdas, siendo superiores en todos los experimentos, excepto en el último.

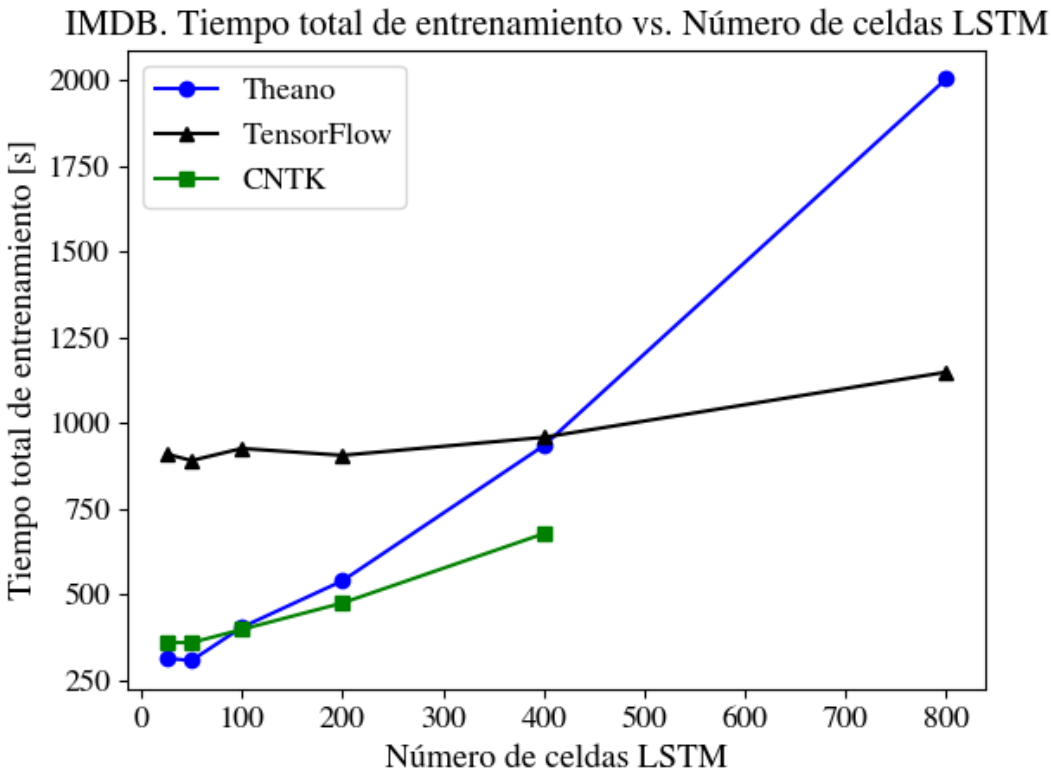


Figura 4.78: Aumento de la cantidad de celdas LSTM en IMDB-LSTM y su impacto en el tiempo total de entrenamiento. Dataset: IMDB Movie Reviews.

De acuerdo a la figura 4.79, el speed-up relativo entre Theano y CNTK varía entre 0,9 y 1,4. Por otra parte, el speed-up entre Theano y TensorFlow decrece rápidamente desde 2,9 hasta 0,6. Análogamente, el speed-up relativo entre CNTK y TensorFlow también decrece, pero desde 2,5 hasta 1,4.

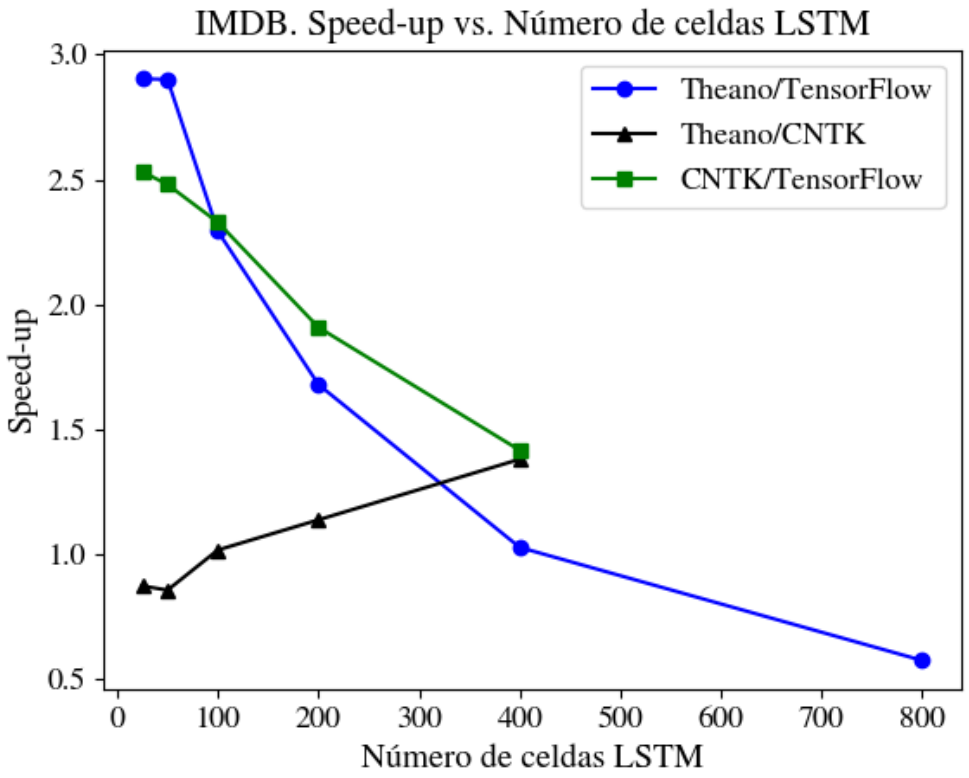


Figura 4.79: Speed-up relativo entre los tiempos de entrenamiento de IMDB-LSTM de cada par de frameworks en función del número de celdas LSTM en la red. Dataset: IMDB Movie Reviews.

Según muestra la figura 4.80, el aumento del número de celdas LSTM lleva a que el consumo de memoria GPU crezca linealmente en cada framework. Al igual que al aumentar la cantidad de capas ocultas, CNTK es quien hace una mayor utilización de memoria GPU, seguido por TensorFlow y Theano. Sin embargo, el rápido aumento en el consumo en CNTK permite comprender la imposibilidad de llevar a cabo el último experimento en este framework. Notar que al reducir el número inicial de celdas, los consumos son muy similares entre frameworks (más aún entre TensorFlow y Theano).

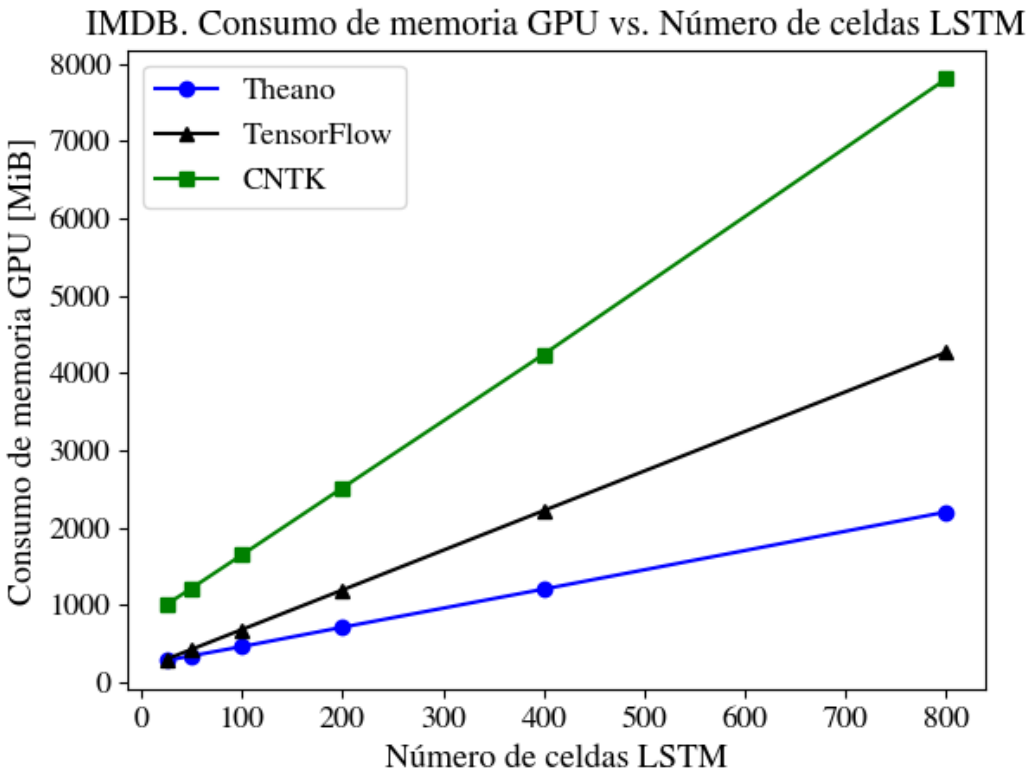


Figura 4.80: Aumento de la cantidad de celdas LSTM en IMDB-LSTM y su impacto en el consumo de memoria GPU. Dataset: IMDB Movie Reviews.

La figura 4.81 muestra que CNTK utiliza entre 3,5 y 3,6 veces más memoria que Theano, a la vez que hace una utilización entre 1,8 y 3,3 veces mayor a la de TensorFlow, dejando nuevamente en evidencia la sobre explotación del recurso por parte de CNTK en este caso particular. Por otro lado, TensorFlow consume entre 1,1 y 1,9 veces más memoria que Theano.

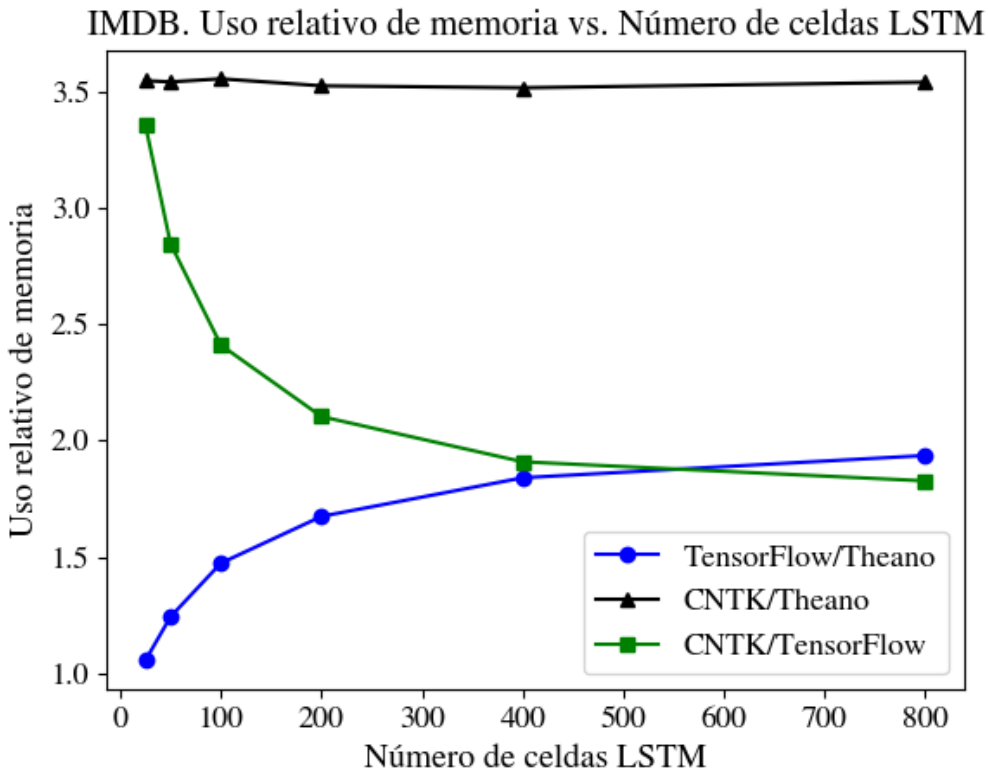


Figura 4.81: Razón entre los consumos de memoria GPU de cada par de frameworks en el entrenamiento de IMDB-LSTM como función del número de celdas LSTM en la red. Dataset: IMDB Movie Reviews.

4.6.3.2. How Much Did It Rain? II

Se ha escogido el modelo que consiguió el segundo lugar en la competencia, correspondiente a una red LSTM y que fue propuesto por Luis Andre Dutra e Silva [6]. El modelo, que para efectos prácticos será denominado HMDIR-LSTM, está estructurado como se muestra a continuación:

- Una capa LSTM con 35 celdas.
- Una capa totalmente conectada de 1 neurona (capa de salida).

Esta arquitectura consigue un error absoluto promedio de 24,70999.

Para el pre-procesamiento de los datos, se utilizó la transformación de Marshall-

[6] http://blog.kaggle.com/2015/12/17/how-much-did-it-rain-ii-2nd-place-luis-andre-dutra-e-silva/

Palmer sobre las cuatro columnas cuyos valores se miden en [dBZ] y se linealizaron las columnas cuyos valores se miden en [dB]. Además, se agregaron 2 atributos adicionales en cada registro, aumentando a 24 la dimensionalidad de los datos: El largo de la secuencia a la cual pertenece dicho registro y el promedio de los valores de los 22 atributos originales. Los atributos ausentes han sido reemplazados por ceros. La capa LSTM hace uso de la función sigmoide como función de activación no recurrente y de hard sigmoid como función de activación recurrente, mientras que la capa de salida, de la función identidad. Además, en la capa LSTM se inicializan los pesos recurrentes por medio de una distribución uniforme de Glorot, mientras que los pesos no recurrentes por medio de una distribución uniforme en el rango [-0,05, 0,05]. Los pesos de la capa de salida también son inicializados mediante una distribución uniforme de Glorot. Para el entrenamiento, se usó RMSprop como método de optimización con tasa de aprendizaje fija de 0,001, tamaño de mini-batch 256, durante 100 epochs.

4.6.3.2.1. Variación de la cantidad de capas LSTM

Así como en la sección 4.6.3.1.1, se varía el número de capas LSTM en HMDIR-LSTM y se estudia su efecto sobre las métricas de rendimiento consideradas. La tabla 4.33 muestra los detalles de las arquitecturas implementadas.

Cuadro 4.33: Variación del número de capas LSTM en HMDIR-LSTM, junto a la cantidad total de parámetros de la arquitectura resultante. Dataset: HMDIR-II.

Número de capas LSTM	Número total de parámetros en la red
1	8.436
2	18.376
3	28.316
4	38.256
5	48.196

De acuerdo a la figura 4.82, los tiempos de Theano y CNTK vuelven a ser similares,

si bien las diferencias entre ellos son mayores respecto a lo observado en IMDB Movie reviews. Además, dichas diferencias se incrementan a medida que se añaden más capas a la red; junto con lo anterior, se observa que los tiempos de ambos frameworks aumentan linealmente con el número de capas LSTM. Por otra parte, los tiempos de TensorFlow vuelven a ser los más altos y siguen un comportamiento que se asemeja levemente al de la función raíz cuadrada.

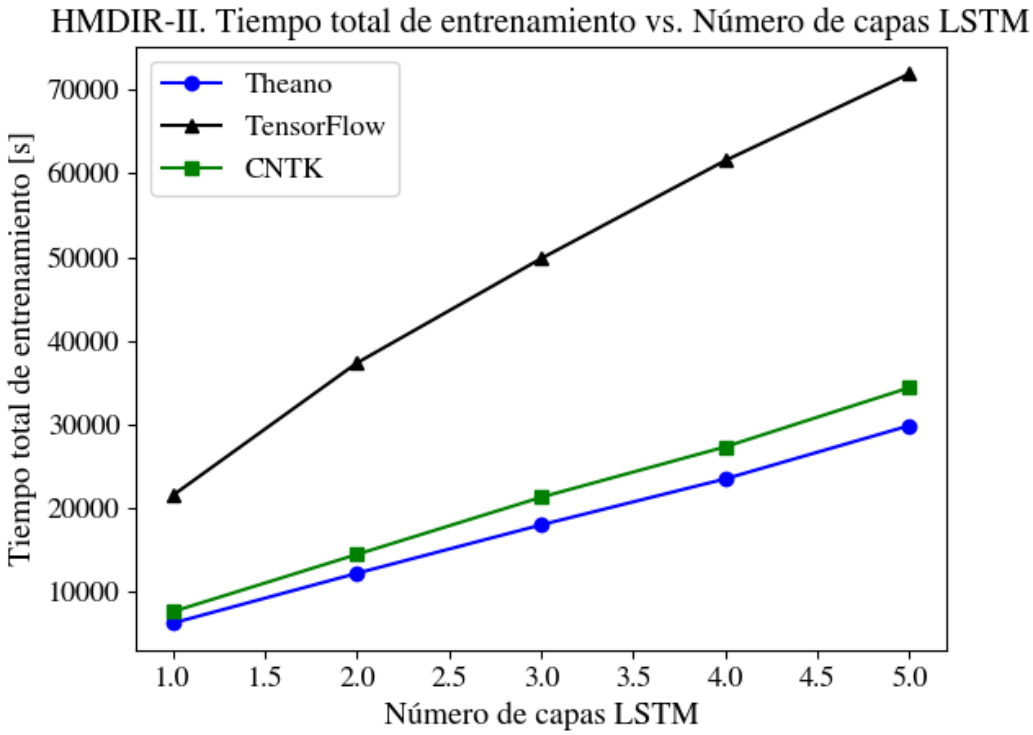


Figura 4.82: Aumento de la cantidad de capas LSTM en HMDIR-LSTM y su impacto en el tiempo total de entrenamiento. Dataset: HMDIR-II.

La figura 4.83 muestra que el speed-up relativo entre Theano y CNTK varía entre 0,82 y 0,87, lo que implica que los tiempos de CNTK son entre 1,1 y 1,2 veces más altos que los de Theano. Por otro lado, el speed-up relativo entre Theano y TensorFlow decrece con el número de capas LSTM, variando entre 2,4 y 3,4. En forma análoga, el speed-up relativo entre CNTK y TensorFlow también decrece, variando entre 2,1 y 2,8.

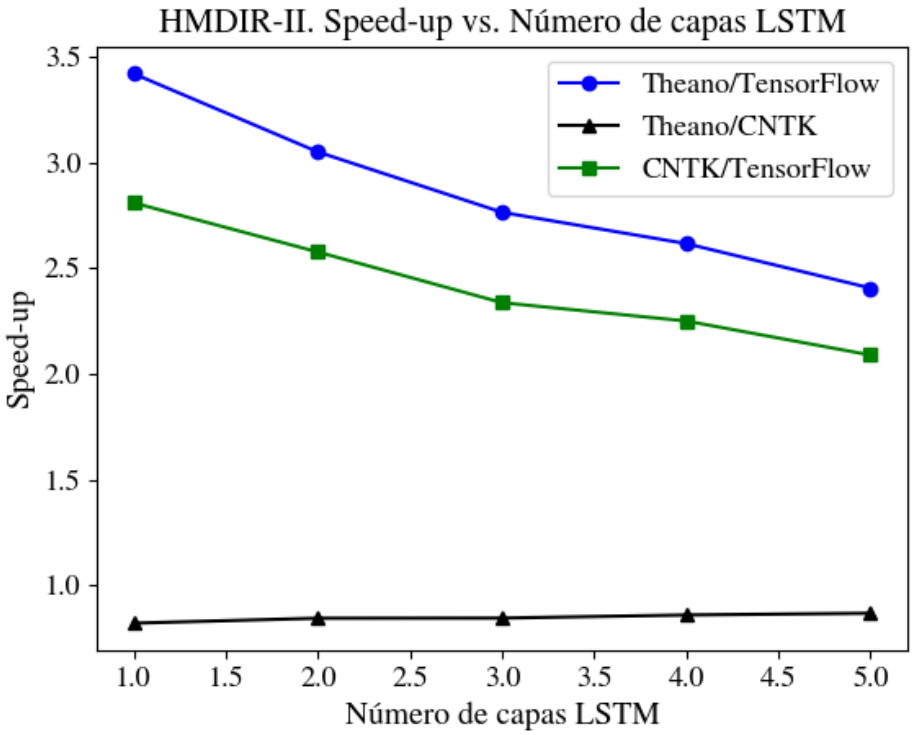


Figura 4.83: Speed-up relativo entre los tiempos de entrenamiento de HMDIR-LSTM de cada par de frameworks en función del número de capas LSTM en la red. Dataset: HMDIR-II.

En la figura 4.84 se muestra que, a diferencia de lo ocurrido con IMDB Movie reviews, es Theano quien consume la mayor cantidad de memoria en todos los experimentos realizados, seguido por CNTK y TensorFlow, si bien las diferencias son muy pequeñas entre sí. Se aprecia que el crecimiento en el consumo de Theano y CNTK es cuasi-lineal, mientras que en TensorFlow la utilización de memoria GPU aumenta sólo cuando la red está integrada por un número par de capas LSTM, mientras que al contar un número impar, el uso de memoria permanece inmutable.

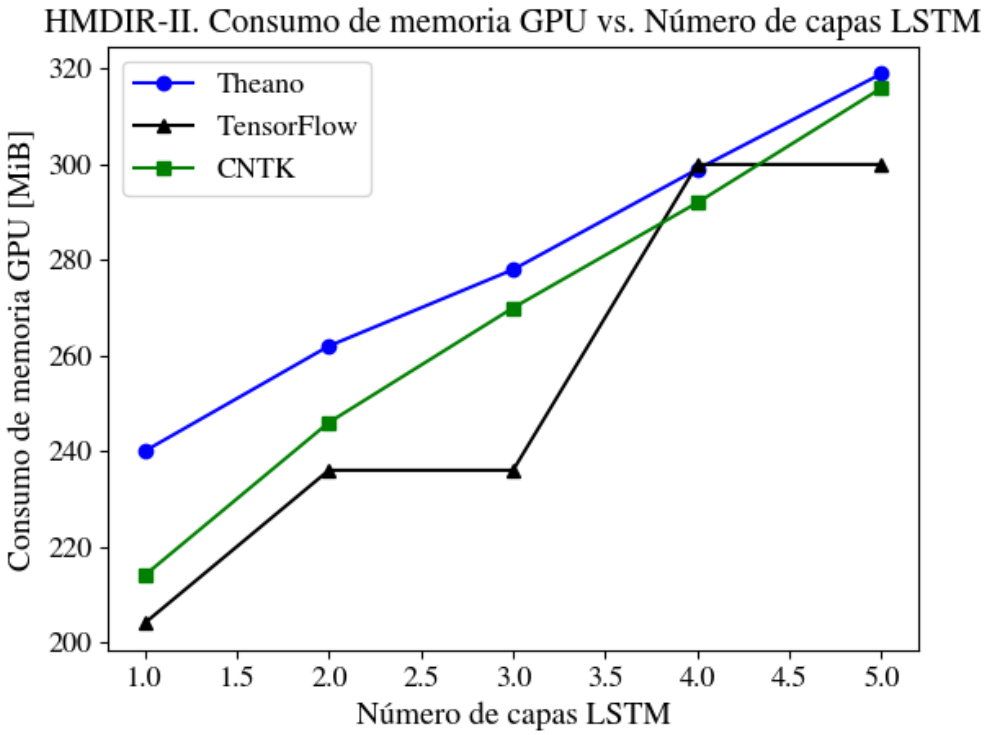


Figura 4.84: Aumento de la cantidad de capas LSTM en HMDIR-LSTM y su impacto en el consumo de memoria GPU. Dataset: HMDIR-II.

Según muestra la figura 4.85, el consumo relativo entre CNTK y Theano varía entre 0,9 y 1. Por otra parte, la razón entre los consumos de CNTK y TensorFlow alcanza su valor mínimo, 1, al cuadruplicar el número de capas, y su valor máximo, 1,1, al triplicarlo. Por último, el consumo relativo entre TensorFlow y Theano varía entre 0,8, al triplicar el número de capas, y 1, al cuadruplicarlo.

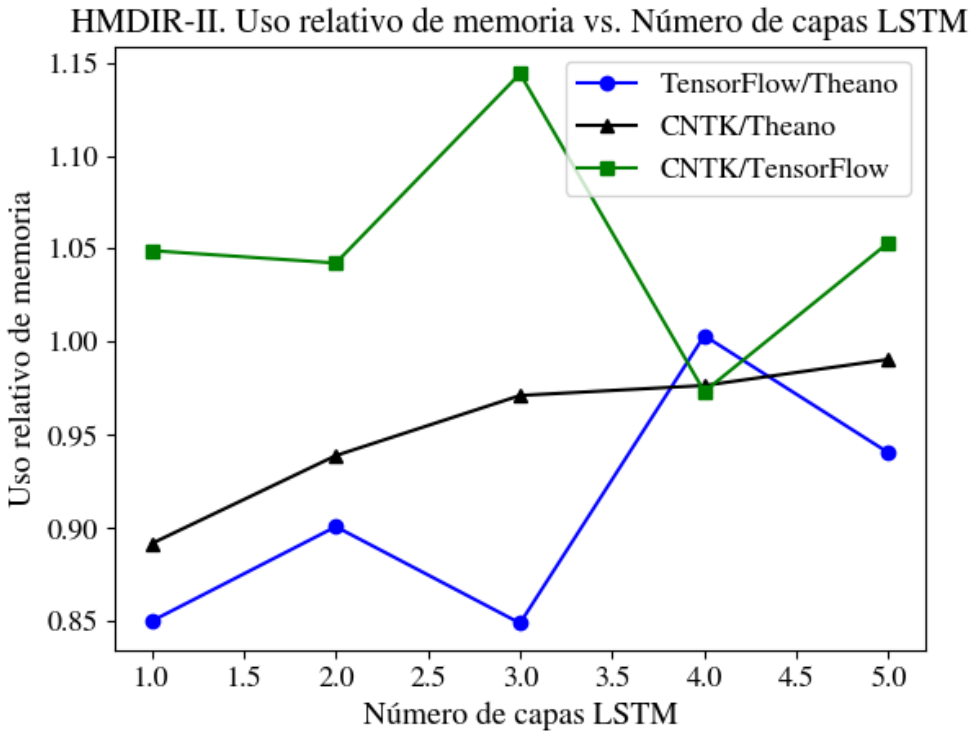


Figura 4.85: Uso relativo de memoria GPU entre cada par de frameworks en el entrenamiento de HMDIR-LSTM como función del número de capas LSTM en la red. Dataset: HMDIR-II.

4.6.3.2.2. Variación de la cantidad de celdas LSTM

Tal como en la sección 4.6.3.1.2, se varía el número de celdas LSTM en HMDIR-LSTM y se estudia el rendimiento de los frameworks en cada caso. Las características de las arquitecturas implementadas se muestran en la la tabla 4.33.

Cuadro 4.34: Variación del número de celdas LSTM en HMDIR-LSTM, junto a la cantidad total de parámetros de la arquitectura resultante. Dataset: HMDIR-II.

Número de celdas LSTM	**Número total de parámetros en la red**
9	1.234
18	3.115
35	8.436
70	26.671
140	92.541
280	341.881

De acuerdo a la figura 4.86, Theano emplea tiempos menores a los de CNTK al reducir el número original de celdas, sin embargo, en los experimentos restantes, los papeles se invierten. De todas maneras, los rendimientos de ambos frameworks son bastantes similares entre sí.Tanto en Theano como en CNTK, el aumento de los tiempos es muy lento al comienzo, pero al aumentar el número de celdas, los tiempos crecen linealmente y en forma mucho más rápida. Por otra parte, los tiempos de TensorFlow son muy superiores y tienden a mantenerse relativamente constantes.

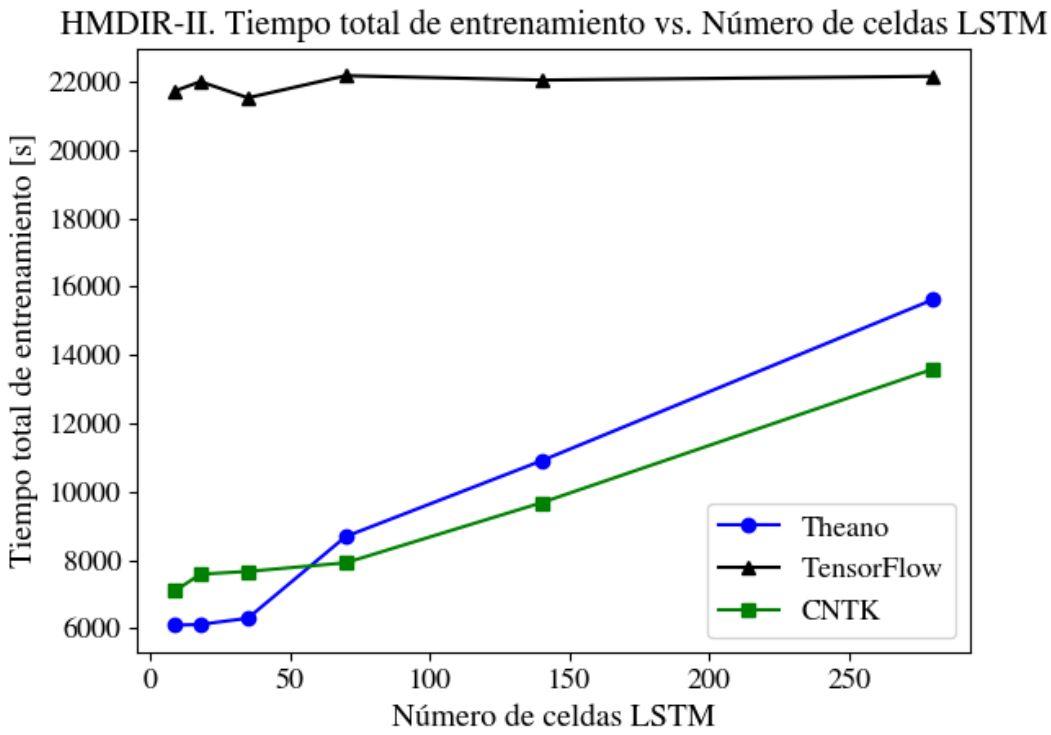


Figura 4.86: Aumento de la cantidad de celdas LSTM en HMDIR-LSTM y su impacto en el tiempo total de entrenamiento. Dataset: HMDIR-II.

Según muestra la figura 4.87, el speed-up relativo entre Theano y CNTK varía entre entre 0,8 y 1,1. Por otra parte, Theano emplea tiempos entre 1,4 y 3,6 veces más bajos que los de TensorFlow y CNTK obtiene tiempos entre 1,6 y 3,1 veces menores que los de TensorFlow.

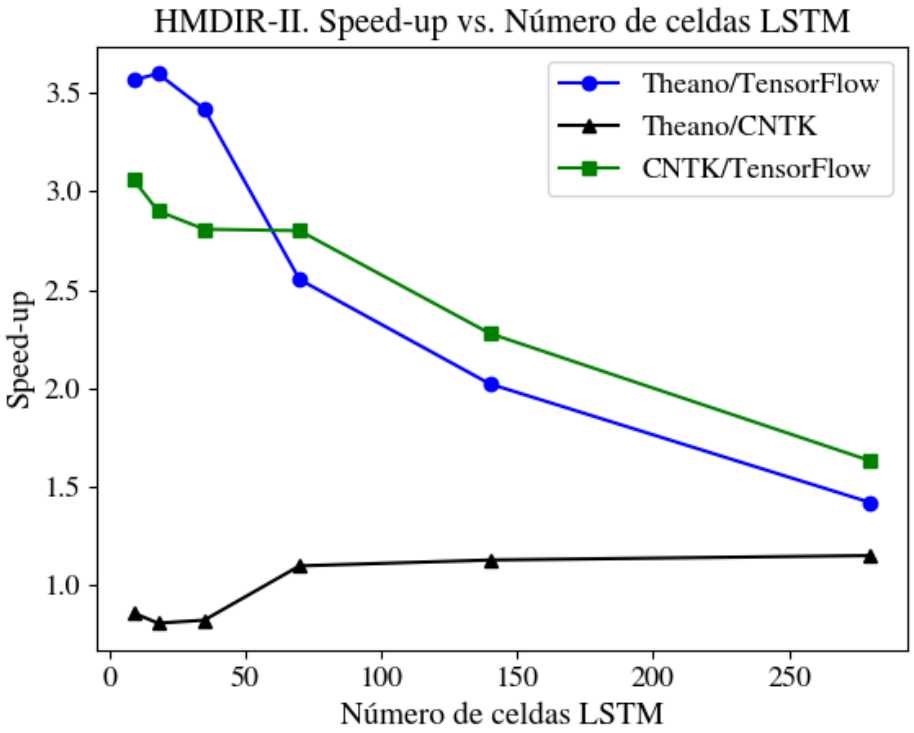


Figura 4.87: Speed-up relativo entre los tiempos de entrenamiento de HMDIR-LSTM obtenidos por cada par de frameworks en función del número de celdas LSTM en la red. Dataset: HMDIR-II.

La figura 4.88 muestra que el consumo de memoria GPU aumenta linealmente con el número de celdas, para todo framework. A diferencia de lo ocurrido con IMDB Movie reviews, Theano hace una mayor utilización del recurso con respecto a TensorFlow en todos los experimentos, excepto al octuplicar dicho número; además, mientras mayor es la cantidad de celdas, más se asemejan los consumos de ambos frameworks. Notar que al igual que en el dataset anterior, el consumo aumenta rápidamente en CNTK, aunque sin llegar a sobre-utilizar el recurso.

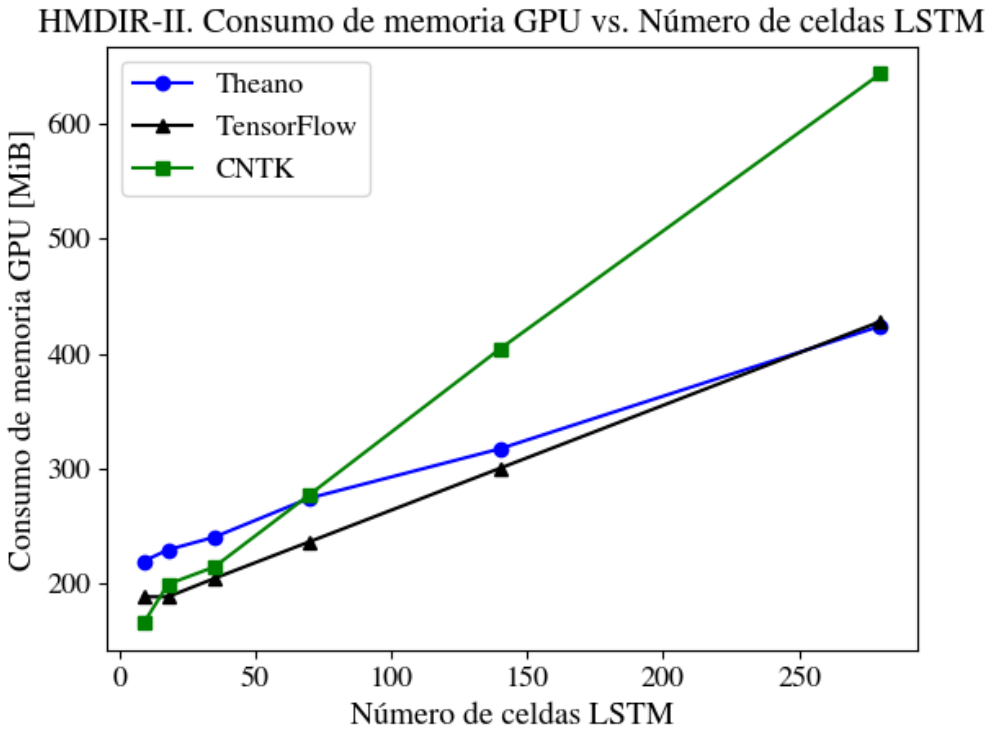


Figura 4.88: Aumento de la cantidad de celdas LSTM en HMDIR-LSTM y su impacto en el consumo de memoria GPU. Dataset: HMDIR-II.

De acuerdo a la figura 4.89, el consumo relativo entre CNTK y Theano varía entre 0,8 y 1,5, de manera que el segundo consigue una mayor utilización del recurso al reducir el número de celdas, mientras que al aumentar dicho número, los papeles se invierten. Por otro lado, la relación entre los consumos de CNTK y TensorFlow varía en el mismo rango, pero este último sólo hace una mayor utilización al reducir la cantidad de celdas a un cuarto del original. Por último, el consumo relativo entre TensorFlow y Theano cambia entre 0,8 y 1.

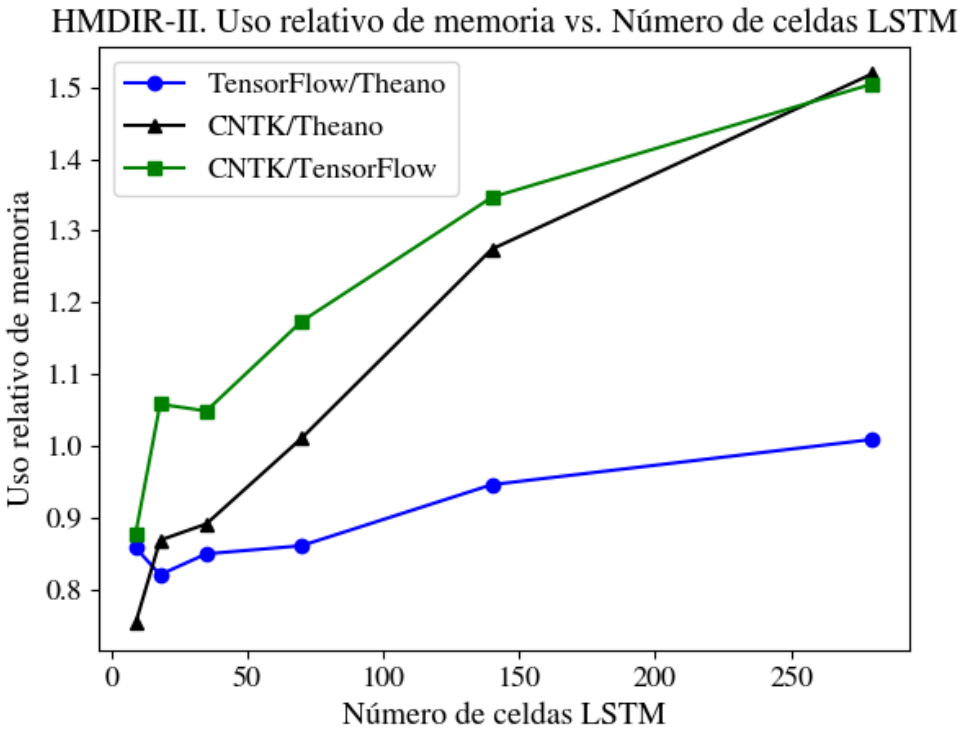


Figura 4.89: Razón entre los consumos de memoria GPU de cada par de frameworks en el entrenamiento de HMDIR-LSTM como función del número de celdas LSTM en la red. Dataset: HMDIR-II.

Capítulo 5

Conclusiones

Los experimentos realizados han demostrado que no existe el framework perfecto.

Theano, por su parte, ha presentado un rendimiento sobresaliente en la implementación de redes feedforward, empleando tiempos de entrenamiento muy inferiores a los de sus competidores, al mismo tiempo que hizo una mejor utilización de la memoria GPU. También mostró un buen rendimiento en la implementación de redes recurrentes. Por otro lado, su rendimiento en la implementación de redes convolucionales fue muy pobre, al obtener tiempos de entrenamiento muy superiores a los de TensorFlow y CNTK, a la vez que sub-utilizó la memoria GPU disponible.

Por otra parte, TensorFlow logró un rendimiento excepcional en la implementación de redes convolucionales, tanto en términos de tiempo de entrenamiento como en el uso eficaz dado a la memoria GPU al variar la cantidad de ejemplos de entrenamiento, al aumentar la cantidad de capas convolucionales, al variar la cantidad de filtros y al variar los tamaños de dichos filtros. Sin embargo, fue el framework que peor rindió en la implementación de redes feedforward y redes LSTM, donde sus tiempos de entrenamiento fueron los más altos y donde la capacidad de cómputo de la GPU fue aprovechada sólo en algunos casos.

CNTK, por otra parte, presentó buenos rendimientos en la implementación de todos los tipos de redes estudiados, siendo, por lo general, comparables con los de sus competidores. En el caso de las redes feedforward, el framework obtuvo tiempos comparables con los de Theano, así como sus tiempos también fueron comparables con los de TensorFlow en la implementación de redes convolucionales. Lamentablemente, un

aumento brusco en la cantidad de parámetros de una red puede llevar a la imposibilidad de que esta sea entrenada en CNTK, dejando en evidencia que, en este escenario, su manejo de memoria GPU es deficiente.

En cuanto a Keras, se destaca que las diferencias de rendimiento en términos de error son mínimas entre los distintos backends. También, llama la atención la sencillez con que es posible llevar a cabo la implementación y el entrenamiento de las redes, gracias a su sintaxis de alto nivel, en contraste con Theano y TensorFlow, API's de bajo nivel que requieren de códigos muchos más extensos y complejos para los mismos fines. Como desventaja, se pueden señalar las dificultades experimentadas al intentar implementar arquitecturas que obtienen resultados cercanos al estado del arte sobre problemas de visión computacional, dada la ausencia de ciertos tipos de capas y/o métodos de entrenamiento sólo posibles de ser utilizados mediante la implementación manual por parte del usuario, requiriendo en algunos casos incluso modificar el código fuente de la herramienta. Otro punto débil a destacar es la generación de mensajes de error muy extensos y poco explicativos.

Para todo tipo de red, se ha detectado que mientras menor es la cantidad de parámetros de una arquitectura determinada, mayor es la similitud que existe entre los rendimientos de los frameworks, situación que puede llegar a ser radicalmente distinta con modelos más complejos.

Se ha observado que tanto la variación de la profundidad de una red feedforward como del tamaño de mini-batch utilizado no produce variaciones significativas en el consumo de memoria GPU. No obstante, es Theano quien logra, en todo escenario, una utilización más alta de la GPU, seguido por TensorFlow y CNTK. En lo que respecta a tiempos de entrenamientos, se puede decir que, para todo framework, estos aumentan linealmente con la profundidad de la red, aunque en forma más bien lenta. Análogamente, para todo framework, el tiempo total de entrenamiento es inversamente proporcional al tamaño de mini-batch considerado, a la vez que decrece rápidamente con dicho tamaño. En todo experimento realizado, Theano obtiene los tiempos más bajos, seguido por CNTK y luego por TensorFlow. Al variar la profundidad de las redes, Theano resulta ser entre 3,4 y 5 veces más rápido que TensorFlow y entre 2,2 y 3 veces más rápido que CNTK, mientras que CNTK es entre 1,4 y 1,8 veces más que rápido que TensorFlow. Al variar el tamaño de mini-batch, Theano resulta ser entre 2,2 y 5,1 veces más rápido que TensorFlow y entre 1,5 y 3,1 veces más rápido que CNTK, mientras que CNTK es

entre 1,4 y 1,8 veces más que rápido que TensorFlow.

Respecto a las redes convolucionales, se observó que el tiempo total de entrenamiento aumenta linealmente con la fracción de ejemplos de entrenamiento considerada, conclusión que es válida en todos los frameworks. TensorFlow logra ser entre 1,9 y 4 veces más rápido que Theano y entre 1,1 y 1,9 veces más rápido que CNTK, a la vez que CNTK es entre 1,7 y 2,4 más rápido que Theano. El consumo de GPU se mantuvo constante en todos los experimentos, dada la independencia existente entre el tamaño de mini-batch y el número de ejemplos de entrenamiento disponible.
Al igual que en el caso anterior, el tiempo total de entrenamiento aumenta linealmente al incrementar el número de capas convolucionales en la mayoría de los casos, idea que también es válida para todos los frameworks. En este contexto, TensorFlow es entre 1,6 y 5,5 veces más rápido que Theano y entre 1,1 y 3,3 veces más rápido que CNTK, mientras que CNTK es entre 1,6 y 3 veces más rápido que Theano. En lo que respecta al consumo de memoria GPU, se observaron aumentos cuasi-lineales en los casos de Theano y CNTK, mientras que en TensorFlow el consumo se mantuvo, por lo general, constante. Así, la utilización de la GPU en TensorFlow fue entre 1,6 y 5,2 veces superior a la de Theano y entre 1 y 2,1 veces superior a la de CNTK, mientras que el consumo de CNTK superó entre 1,7 y 5,3 veces al de Theano.
Por otra parte, el incremento de la cantidad de filtros en las capas convolucionales llevó a un crecimiento súper lineal en los tiempos totales de entrenamiento de los tres frameworks. Luego, TensorFlow resulta ser entre 1,4 y 6,8 veces más rápido que Theano y entre 1,1 y 2,1 veces más rápido que CNTK, a la vez que CNTK es entre 1,3 y 4,4 veces más rápido que Theano. Además, el consumo de memoria GPU varía en forma similar al tiempo total de entrenamiento en todos los frameworks. Así, la utilización del recurso en TensorFlow es entre 1,2 y 4,9 veces superior a la de Theano y entre 1,1 y 1,5 veces mayor a la de CNTK, mientras que este último consume entre 1,1 y 4,5 veces más memoria que Theano.
Por último, la variación de los tamaños de los filtros en las capas convolucionales llevó a que los tiempos totales de entrenamiento decrecieran linealmente en forma muy lenta, o bien, se mantuvieran relativamente constantes en los tres frameworks. TensorFlow es entre 1,7 y 5,2 veces más rápido que Theano y entre 1,2 y 1,7 veces más rápido que CNTK, al mismo tiempo que CNTK es entre 1,9 y 3,8 veces más rápido que Theano. El consumo de memoria GPU, por su parte, experimenta un comportamiento similar al del tiempo total de entrenamiento. Así, TensorFlow utiliza la GPU entre 1,4 y 6,3 veces

más que Theano y entre 1 y 2,2 veces más que CNTK, así como CNTK consume entre 1,2 y 3,7 veces más memoria GPU que Theano.

Respecto al entrenamiento de redes recurrentes, se observó que el tiempo total de entrenamiento aumenta linealmente al incrementar el número de capas LSTM en todos los frameworks. CNTK resultó ser entre 1 y 1,1 veces más rápido que Theano y entre 2,1 y 2,8 veces más rápido que TensorFlow, a la vez que Theano fue entre 1,1 y 1,3 veces más rápido que CNTK y entre 2,1 y 3,4 veces más rápido que TensorFlow. Por otro lado, el consumo de memoria GPU también aumenta en forma lineal (o bien, cuasi-lineal) en CNTK, quien presentó, en general, el consumo más alto, y en Theano, quien obtuvo el consumo más bajo, mientras que en TensorFlow la utilización de la GPU aumenta y se comporta como una función escalonada. Luego, CNTK consume entre 1 y 3,6 veces más memoria GPU que Theano y entre 1 y 2,4 veces más que TensorFlow; Theano puede llegar a utilizar hasta 1,1 veces más memoria que CNTK y entre 1,1 y 1,3 veces más memoria que TensorFlow, mientras que este último hace una utilización entre 1 y 2,3 veces superior a la de Theano.
Además, el aumento en la cantidad de celdas LSTM lleva a que el tiempo total de entrenamiento aumente ya sea en forma súper lineal o lineal en Theano y CNTK, dependiendo de la complejidad del dataset, mientras que en TensorFlow tiende a mantenerse constante. Notar que en el caso de redes con una capa LSTM y un número de celdas inferior a 100, Theano obtiene tiempos de entrenamiento más bajos que los de CNTK, mientras que los papeles se invierten al utilizar una cantidad de celdas superior a 100. TensorFlow, en casi la totalidad de los casos, obtiene los tiempos más altos. Desafortunadamente, un aumento exagerado en la cantidad de celdas puede llevar a una sobre-utilización de la GPU por parte de CNTK, generándose un error de memoria GPU insuficiente, tal como ocurrió al intentar entrenar una red integrada por una capa LSTM y 800 celdas sobre el dataset IMDB Movie reviews. Luego, CNTK es entre 1,1 y 1,4 veces más rápido que Theano y entre 1,4 y 3,1 veces más rápido que TensorFlow, así como Theano es entre 1,4 y 3,6 más rápido que TensorFlow. El consumo de memoria GPU, por otra parte, aumenta linealmente con el incremento del número de celdas en todos los frameworks, de manera que el mayor consumo se da en CNTK; al igual que al aumentar el número de capas LSTM, TensorFlow hace una mejor utilización del recurso respecto a Theano en algunos casos, mientras que los papeles se invierten en otros experimentos, llevando a la imposibilidad de generalizar el comportamiento de estos frameworks. Así, CNTK hace una utilización de la GPU entre 1,5 y 3,6 veces superior

a la de Theano y entre 1,5 y 3,3 veces más alta que TensorFlow, mientras que este último consume entre 1,1 y 1,9 veces más memoria que Theano y Theano entre 1 y 1,3 veces más que TensorFlow.

Bibliografía

[1] BURGER, Josef. A Basic Introduction To Neural Networks [en línea] <http://pages.cs.wisc.edu/~bolo/shipyard/neural/local.html> [consulta: 30 de agosto de 2017].

[2] MCCULLOCH, Warren y PITTS, Walter. A logical calculus of the ideas immanent in nervous activity. The bulletin of mathematical biophysics, 5(4):115-133, Diciembre, 1943.

[3] POKHARNA, Harsh. The best explanation of Convolutional Neural Networks on the Internet! [en línea] <https://medium.com/technologymadeeasy/the-best-explanation-of-convolutional-neural-networks-on-the-internet-fbb8b1ad5df8> [consulta: 1 de septiembre de 2017].

[4] BENGIO, Yoshua, SIMARD, Patrice y FRASCONI, Paolo. Learning long-term dependencies with gradient descent is difficult. IEEE Transactions on Neural Networks, 5(2):157-166, marzo de 1994.

[4] DESHPANDE, Adit. A Beginner's Guide To Understanding Convolutional Neural Networks Part 2 [en línea] <https://adeshpande3.github.io/adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks-Part-2/> [consulta: 6 de septiembre de 2017].

[5] DESHPANDE, Adit. A Beginner's Guide To Understanding Convolutional Neural Networks [en línea] <https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/> [consulta: 1 de septiembre de 2017].

[6] BRITZ, Denny. Recurrent Neural Networks Tutorial, Part 1 – Introduction to RNNs [en línea] <http://www.wildml.com/2015/09/recurrent-neural-networks-tutorial-part-1-introduction-to-rnns/> [consulta: 6 de septiembre de 2017].

[7] THEANO: A Python framework for fast computation of mathematical expressions por Rami Al-Rfou [et al] [en línea]. CoRR. 9 de mayo de 2016. abs/1605.02688v1 <https://arxiv.org/abs/1605.02688v1.pdf> [consulta: 22 de diciembre de 2017].

[8] OLAH, Christopher. Understanding LSTM Networks [en línea] <http://colah.github.io/posts/2015-08-Understanding-LSTMs/> [consulta: 7 de septiembre de 2017].

[9] BROWNLEE, Jason. What is Deep Learning? [en línea] <https://machinelearningmastery.com/what-is-deep-learning/> [consulta: 11 de septiembre de 2017].

[10] BENGIO, Yoshua. Deep Learning of Representations for Unsupervised and Transfer Learning. JMLR W & CP: Proc. Unsupervised and Transfer Learning, 27:17, 2011.

[11] SHAIKH, Faizan. Why are GPUs necessary for training Deep Learning models? [en línea] <https://www.analyticsvidhya.com/blog/2017/05/gpus-necessary-for-deep-learning/> [consulta: 11 de septiembre de 2017].

[12] PIETRO, Piscione, ALESSIO, Villardita. Power Consumption and Performance trends on GPU's [diapositivas]. Pisa, Italia. Università di Pisa, 2015. 25 diapositivas.

[13] GUIM, Francesc, RODERO, Ivan. CUDA. En su: Arquitecturas basadas en computación gráfica (GPU). Cataluña, Universitat Oberta de Catalunya, 2013, pp. 45-56.

[14] LABORATOIRE d'Informatique des Systèmes Adaptatifs. Theano at a Glance [en línea] <http://www.deeplearning.net/software/theano/introduction.html> [consulta: 25 de agosto de 2017].

[15] GOOGLE. About TensorFlow [en línea] <https://www.tensorflow.org/> [consulta: 25 de agosto de 2017].

[16] GOOGLE. API Documentation [en línea] <https://www.tensorflow.org/api_docs/> [consulta: 25 de agosto de 2017].

[17] MICROSOFT. The Microsoft Cognitive Toolkit [en línea] <https://www.microsoft.com/en-us/cognitive-toolkit/>[consulta: 28 de agosto de 2017].

[18] MASSACHUSETTS Institute of Technology. Keras: The Python Deep Learning Library [en línea] <https://keras.io/>[consulta: 28 de agosto de 2017].

[19] COMPARATIVE Study of Deep Learning Software Frameworks por Soheil Bahrampour [et al] [en línea]. CoRR. 30 de marzo de 2016. abs/1511.06435 <http://arxiv.org/abs/1511.06435> [consulta: 27 de diciembre de 2017].

[20] GRADIENT-BASED Learning Applied to Document Recognition por Yann LeCun [et al]. Proceedings of the IEEE, 86:2284-2286, noviembre de 1998.

[21] KRIZHEVSKY, Alex, SUTSKEVER Ilya y HINTON, Geoffrey. ImageNet Classification with Deep Convolutional Neural Networks. Proceedings of the 25th International Conference on Neural Information Processing Systems, 1:1100-1101, diciembre de 2012.

[22] LABORATOIRE d'Informatique des Systèmes Adaptatifs. LSTM Networks for Sentiment Analysis. En su: Deep Learning Tutorial. Versión 0.1. Montreal, Canadá, Laboratoire d'Informatique des Systèmes Adaptatifs, 2015, pp. 145.

[23] OVERFEAT: Integrated Recognition, Localization and Detection using Convolutional Networks por Pierre Sermanet [et al] [en línea]. CoRR. 24 de febrero de 2014. abs/1312.6229 <https://arxiv.org/abs/1312.6229> [consulta: 27 de diciembre de 2017].

[24] SIMONYAN, Karen y ZISSERMAN, Andrew. Very deep convolutional networks for large-scale image recognition [en línea]. CoRR. 10 de abril de 2015. abs/1409.1556 <https://arxiv.org/abs/1409.1556> [consulta: 27 de diciembre de 2017].

[25] GOING Deeper with Convolutions por Christian Szegedy [et al] [en línea]. CoRR. 17 de septiembre de 2014. abs/1409.4842 <http://arxiv.org/abs/1409.4842> [consulta: 27 de diciembre de 2017].

[26] MARCUS, Mitchell, SANTORINI, Beatrice y MARCINKIEWICZ, Mary. Building a Large Annotated Corpus of English: The Penn Treebank. Comput. Linguist., 19(2):313-330, junio de 1993.

[27] DESCRIBING Videos by Exploiting Temporal Structure por Li Yao [et al] [en línea]. CoRR. 1 de octubre de 2015. abs/1502.08029 <https://arxiv.org/abs/1502.08029> [consulta: 27 de diciembre de 2017].

[28] TENSORFLOW: A system for large-scale machine learning por Martín Abadi [et al] [en línea]. CoRR. 31 de mayo de 2016. abs/1605.08695 <http://arxiv.org/abs/1605.08695> [consulta: 27 de diciembre de 2017].

[29] IOFFE, Sergey, SZEGEDY, Christian. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift [en línea]. CoRR. 2 de marzo de 2015. abs/1502.03167 <http://arxiv.org/abs/1502.03167> [consulta: 27 de diciembre de 2017].

[30] RETHINKING the Inception Architecture for Computer Vision por Christian Szegedy [et al] [en línea]. CoRR. 11 de diciembre de 2015. abs/1512.00567 <http://arxiv.org/abs/1512.00567> [consulta: 27 de diciembre de 2017].

[31] BENCHMARKING State-of-the-Art Deep Learning Software Tools por Shaohuai Shi [et al] [en línea]. CoRR. 19 de septiembre de 2016. abs/1608.07249 <http://arxiv.org/abs/1608.07249> [consulta: 27 de diciembre de 2017].

[32] DEEP Residual Learning for Image Recognition por Kaiming He [et al] [en línea]. CoRR. 10 de diciembre de 2015. abs/1512.03385 <http://arxiv.org/abs/1512.03385> [consulta: 27 de diciembre de 2017].

[33] GRADIENT-BASED Learning Applied to Document Recognition por Yann LeCun [et al]. Proceedings of the IEEE, 86:2286-2287, noviembre de 1998.

[34] KRIZHEVSKY, Alex. Learning a generative model of images. En su: Learning multiple layers of features from tiny images. Toronto, Canadá, Department of Computer Science, University of Toronto, 2009, pp. 32.

[35] READING Digits in Natural Images with Unsupervised Feature Learning por Yuval Netzer [et al]. NIPS Workshop on Deep Learning and Unsupervised Feature Learning, :1-9, 2011.

[36] LECUN, Yann, HUANG, Fu y BOTTOU, Léon. Learning methods for generic object recognition with invariance to pose and lightning. Proceedings of the 2004 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2:97-104, julio de 2004.

[37] LEARNING word vectors for sentiment analysis por Andrew L. Maas [et al]. Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies, 1:142-150, 2011.

[38] PROBABILISTIC precipitation rate estimates with ground-based radar networks por Pierre-Emmanuel Kirstetter [et al]. Water Resources Research, 51:1422-1442, marzo de 2015.

[39] LIN, Zhouhan, MEMISEVIC, Roland y KONDA, Kishore. How far can we go without convolution: Improving fully-connected networks [en línea]. CoRR. 9 de noviembre de 2015. abs/1511.02580 <http://arxiv.org/abs/1511.02580> [consulta: 28 de diciembre de 2017].

[40] OLIVA, Aude y TORRALBA, Antonio. Modeling the Shape of the Scene: A Holistic Representation of the Spatial Envelope. Int. J. Comput. Vision, 52(3):145-175, mayo de 2001.

[41] CIRESAN, Dan, MEIER, Ueli y SCHMIDHUBER, Juergen. Multi-column Deep Neural Networks for Image Classification [en línea]. CoRR. 13 de febrero de 2012. abs/1202.2745 <http://arxiv.org/abs/1202.2745> [consulta: 28 de diciembre de 2017].

[42] Deeply-Supervised Nets por Chen-Yu Lee [et al] [en línea]. CoRR. 25 de septiembre de 2014. abs/1409.5185 <https://arxiv.org/abs/1409.5185> [consulta: 28 de diciembre de 2017].

[43] STRIVING For Simplicity: The All Convolutional Net por Jost Springenberg [et al] [en línea]. CoRR. 13 de abril de 2015. abs/1412.6806 <http://arxiv.org/abs/1412.6806> [consulta: 28 de diciembre de 2017].

[44] LEARNING Activation Functions To Improve Deep Neural Networks por Forest Agostinelli [et] [en línea]. CoRR. 21 de abril de 2015. abs/1412.6830 <http://arxiv.org/abs/1412.6830> [consulta: 28 de diciembre de 2017].

[45] CLEVERT, Djork-Arné, UNTERTHINER, Thomas y HOCHREITER, Sepp. Fast And Accurate Deep Network Learning By Exponential Linear Units (ELUs) [en línea]. CoRR. 22 de febrero de 2016. abs/1511.07289 <http://arxiv.org/abs/1511.07289> [consulta: 28 de diciembre de 2017].

[46] ZAGORUYKO, Sergey y KOMODAKIS, Nikos. Wide Residual Networks [en línea]. CoRR. 14 de junio de 2017. abs/1605.07146 <http://arxiv.org/abs/1605.07146> [consulta: 28 de diciembre de 2017].

[47] SHI, Zenglin, YE, Yangdong y WU, Yunpeng. Rank-based pooling for deep convolutional neural networks. Neural Netw., 83(C):21-31, noviembre de 2016.

[48] BROWNLEE, Jason. Sequence Classification with LSTM Recurrent Neural Networks in Python with Keras [en línea] <https://machinelearningmastery.com/sequence-classification-lstm-recurrent-neural-networks-python-keras/> [consulta: 28 de abril de 2018].

Anexos

A. Módulo inception

Un módulo inception corresponde a la aplicación en forma paralela de múltiples capas convolucionales sobre un mismo input, donde cada una de ellas está integrada por filtros de dimensiones diferentes a los de las demás capas. La figura A.1 muestra la estructura de un módulo inception.

De acuerdo a dicha figura, inception está integrado por los siguientes sub-módulos:

- Una capa convolucional con filtros de tamaño 1 x 1.
- Una capa convolucional con filtros de tamaño 1 x 1, seguida por una capa convolucional con filtros de tamaño 3 x 3.
- Una capa convolucional con filtros de tamaño 1 x 1, seguida por una capa convolucional con filtros de tamaño 5 x 5.
- Una capa de max pooling con pooling size de tamaño 3 x 3, seguida por una capa convolucional con filtros de tamaño 1 x 1.

Los feature maps obtenidos a partir de la aplicación de cada sub-módulo son concatenados en la dimensión de profundidad y dan forma al input recibido por la siguiente capa de la red.

Teóricamente, un módulo inception busca la extracción simultánea de características de diferente complejidad en un mismo nivel de la red, desde atributos más bien específicos por medio de la aplicación de filtros de tamaño 1 x 1, o bien, atributos de carácter general a través del uso de filtro de 5 x 5.

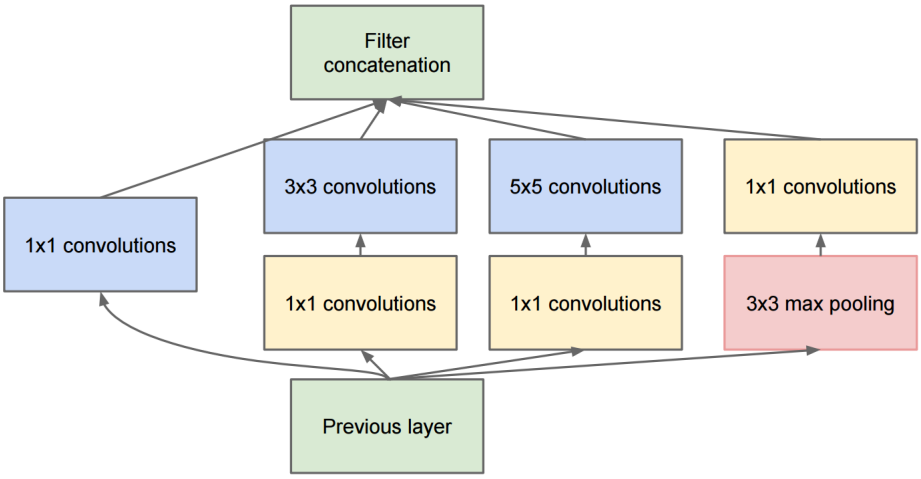


Figura A.1: Estructura de un módulo inception.