Algoritmo de aprendizaje incremental basado en RMSProp y representantes para reducir el tiempo de reentrenamiento de redes convolucionales

Camilo Narváez Rivas
José David Muñoz Gómez

Director: PhD. Carlos Alberto Cobos Lozada
Codirectora: PhD. Martha Eliana Mendoza Becerra

Universidad del Cauca
Facultad de Ingeniería Electrónica y Telecomunicaciones
Departamento de Sistemas
Grupo de I+D en Tecnologías de la Información
Línea Investigación: Sistemas Inteligentes
Popayán, julio de 2019

TABLA DE CONTENIDO

LISTA DE FIGURAS

LISTA DE TABLAS

RESUMEN

En Deep Learning, entrenar apropiadamente un modelo con datos en gran cantidad y de alta calidad es crucial para alcanzar un buen resultado. Sin embargo, en algunas tareas los datos necesarios no se encuentran disponibles en un momento único y sólo se pueden obtener a lo largo del tiempo. En el último caso, el Aprendizaje Incremental es una alternativa a usar para entrenar un modelo apropiadamente, sin embargo, se presenta un problema en la forma del dilema de la estabilidad-plasticidad: cómo entrenar incrementalmente un modelo que pueda responder bien a nuevos datos (plasticidad), a la vez que se retiene el conocimiento previo (estabilidad). El presente trabajo de investigación propone un modelo de aprendizaje incremental inspirado en Rehearsal (recuerdo de memorias pasadas basada en un subconjunto de datos) el cual se ha denominado CRIF, y que ha sido construido sobre un framework de soporte de experimentación de algoritmos de aprendizaje incremental llamado DILF, el cual también ha sido propuesto en este trabajo. Adicionalmente se proponen dos algoritmos que instancian el marco establecido en CRIF: uno que usa una selección aleatoria de muestras representativas (NIL) y otro que usa las métricas de *Best vs. Second Best* y distancia de *Crowding* en conjunto para esta tarea (RILBC).

El rendimiento de los algoritmos propuestos fue evaluado usando tres métricas, a saber: exactitud, tiempo y pérdida (*loss*). Los experimentos fueron realizados sobre cuatro datasets, MNIST, Fashion-MNIST, CIFAR-10, y Caltech 101; y en dos escenarios incrementales diferentes: un escenario con clases estrictamente incrementales, y un escenario con clases pseudo incrementales y datos desbalanceados. En Caltech 101 se usó Transfer Learning, y en este escenario, así como en los otros tres datasets, el método propuesto NIL alcanza mejores resultados en exactitud que los algoritmos comparados, tales como RMSProp Inc (línea base) e iCaRL (propuesta del estado del arte), y sobre el otro método propuesto RILBC. NIL obtiene resultados que son estadísticamente comparables con un proceso de entrenamiento acumulativo (todos los datos disponibles) y también demuestra poder alcanzar estos resultados en menos tiempo.

CAPÍTULO 1

1 INTRODUCCIÓN

1.1 PLANTEAMIENTO DEL PROBLEMA

Machine Learning es una rama de la Inteligencia Artificial cuyo objetivo es crear sistemas capaces de aprender y generalizar comportamientos. En términos generales una máquina aprende cuando cambia su estructura, programa o datos de forma que mejore su rendimiento [1]. Estos algoritmos pueden ser usados en diversas tareas de clasificación y regresión, las cuales se enmarcan dentro de los llamados modelos supervisados [2] y tienen diversas aplicaciones, como: visión artificial, predicción de enfermedades, reconocimiento de voz, entre otras [3]. La tarea de clasificación consiste en predecir y asignar a un registro o instancia nueva, una categoría o clase que corresponda con el valor real [2][4][5], mientras que la tarea de regresión consiste en mapear una función a partir de los datos suministrados para predecir su comportamiento [4].

Una familia muy importante de algoritmos de Machine Learning son las Redes Neuronales Artificiales (RNA), que son un modelo computacional basado en elementos del funcionamiento del cerebro humano, en el cual se conectan muchas unidades pequeñas de procesamiento [6]. Recientemente se ha visto la necesidad de crear RNA con arquitecturas muy grandes para resolver problemas complejos como la conducción autónoma de automóviles, análisis de proteínas, reconocimiento de escritura o la detección de voz. Estas arquitecturas reciben el nombre colectivo de Deep Learning [7].

El proceso de entrenamiento en aprendizaje supervisado funciona de la siguiente manera [2][5]: se toma un dataset con datos etiquetados (dataset de entrenamiento) para ser suministrado a un algoritmo clasificador o de regresión. El algoritmo de Machine Learning plantea una hipótesis o función generalizadora y = f(x) que busca predecir con la mayor exactitud posible, el valor y dado un x del dataset de entrenamiento. Finalmente, se usa un dataset independiente para pruebas, con respecto al cual se comprueba la precisión de la hipótesis obtenida.

En el caso específico de las RNA se usan diferentes algoritmos de entrenamiento, entre ellos el Back Propagation (BP) y RMSProp. BP es un método simple basado en el gradiente descendente y la propagación del error entre las capas de la red neuronal [6], pero RMSProp que es una mejora de BP, funciona con mini lotes y

divide el gradiente por un promedio de sus valores anteriores [8]. Este último algoritmo es ampliamente usado en Deep Learning [9], tanto en Redes Convolucionales [10] como en otras arquitecturas de red [11] y la literatura reporta que supera en rendimiento a los otros métodos de entrenamiento del estado del arte [12].

En algunas aplicaciones de Machine Learning es necesario hacer un reentrenamiento constante debido a que sus datos surgen de entornos dinámicos y son adquiridos a lo largo del tiempo. Si no se realiza el reentrenamiento, los modelos se desactualizan y reducen su precisión, debido a que se pierde la oportunidad de entrenar con nuevos datos para obtener un mejor modelo [13][14]. Sin embargo, reentrenar y construir nuevamente el modelo cada vez que un nuevo lote de datos esté disponible no es una buena opción e incluso podría ser inviable debido al gran volumen de datos que se opera y a que algunos modelos (como los de la familia Deep Learning) son sumamente complejos y requieren gran cantidad de recursos computacionales y de tiempo para su entrenamiento [15].

El aprendizaje incremental ha surgido como una alternativa para solucionar los problemas de un reentrenamiento completo, de forma que los modelos se adapten constantemente con base en un flujo de datos que cambia en el tiempo [13]. El enfoque de entrenamiento incremental es deseable en aplicaciones como procesamiento de Big Data, robótica, etiquetado automático, procesamiento de imágenes y detección de anomalías [13]. Sin embargo, con este enfoque se presenta la dificultad fundamental de mantener la capacidad de aprender al tiempo que se conserva el conocimiento previamente obtenido, el llamado dilema de la estabilidad-plasticidad [16].

En la literatura se encuentran propuestas de aprendizaje incremental basadas en el ensamble de modelos tipo boosting, entre los que se encuentran árboles de decisión con múltiples clasificadores [17], redes convolucionales colaborativas [18] [19] y el algoritmo Learn++, el cual construye varias redes neuronales para hacer la clasificación [20], de esta manera cada red se enfoca en clasificar unos datos específicos, sin embargo, este tipo de solución debe gestionar cada uno de los modelos creados, por lo cual su complejidad se hace bastante alta, además poseen problemas cuando los datos poseen ruido y si los clasificadores son complejos pueden llevar a un sobre entrenamiento (overfitting).

También se encuentran propuestas con un enfoque centrado en el cambio dinámico de la estructura del modelo, como redes Fuzzy con reglas dinámicas [14][21] y clustering semi-supervisado [22][23]. La limitación de este enfoque es que se cambia la estructura misma de los clasificadores, por lo que las propuestas mencionadas no pueden extenderse a otros clasificadores ya existentes, como las Redes Neuronales Profundas, entre ellas las Redes Convolucionales.

Otras propuestas de solución se enmarcan en un método llamado aprendizaje activo, en el cual el algoritmo selecciona de forma continua muestras no

etiquetadas para que se etiqueten manualmente y sean usadas luego en la fase de entrenamiento [24]. De esta forma, el problema clave consiste en seleccionar las muestras que aporten la mayor cantidad de conocimiento posible. En el estado del arte se encontraron estrategias de selección de muestras con base en clústers tradicionales [25] y elípticos [26], y según la incertidumbre que se tiene durante la clasificación [27][28] o usando una combinación de medidas de representatividad e incertidumbre de las muestras [29][30]. Un inconveniente de esta técnica es que no balancea el impacto del nuevo conocimiento respecto al conocimiento previo.

Un aspecto crucial para lograr un buen rendimiento en procesos de aprendizaje incremental involucra el adecuado manejo del desbalanceo de clases (i.e. cuando las clases tienen una cantidad muy diferente de muestras). Para ello se ha usado la creación de muestras sintéticas del algoritmo SMOTE, variando desde una interpolación simple [31][32] hasta el uso del algoritmo de clustering con múltiples centroides CURE [33]. Por otro lado, en [34] se hace uso de representantes obtenidos a partir de una Red Neuronal para realizar un enfoque de clasificación Nearest-Mean-of-Exemplars. También se ha explorado el uso de redes DCGAN para generar representantes que son usados juntos con los nuevos datos para realizar el entrenamiento. En [35] se han generado muestras artificiales usando distribuciones gaussianas y también se ha propuesto usar un esquema de pesos para penalizar la clasificación incorrecta de las clases minoritarias [36][37].

Por todo lo anterior en este trabajo de grado se planteó la siguiente pregunta de investigación: ¿Es posible reducir el tiempo de reentrenamiento, manteniendo la calidad de los resultados en problemas de clasificación mediante la integración de un método de selección de muestras representativas en el entrenamiento de redes convolucionales con RMSProp?

Al observar las técnicas de aprendizaje activo y desbalanceo de clases, se pensó en definir una estrategia para resolver el dilema de la estabilidad-plasticidad, basada en mantener muestras (prototipos o representantes) de entrenamientos previos a las que se les asigne un peso de acuerdo con su representatividad y usarlos junto con los nuevos lotes de datos para realizar un proceso de aprendizaje incremental. A diferencia de los trabajos previos, en este proyecto se aprovechó la información de la clasificación de los datos que se hace en el entrenamiento de una red neuronal convolucional.

Por lo anterior, en este trabajo se tomaron dos enfoques de solución, el primero denominado RILBC, se basa en modificar RMSProp para el entrenamiento de redes neuronales convolucionales, de manera que los prototipos se escojan incrementalmente buscando que se encuentren en zonas cercanas a los hiperplanos de separación de las clases (basado en la incertidumbre de la capa de salida) y que cubran la mayor cantidad de fronteras de separación evitando que el conjunto de prototipos crezca descontroladamente. Esta propuesta hace uso de una métrica de selección de candidatos basada en Best vs Second Best junto a

una estrategia de competencia de representantes con una medida de Crowding Distance y un regularizador que penaliza la mala clasificación de los representantes.

El segundo enfoque denominado NIL, se basa en la introducción de aleatoriedad realizando una selección aleatoria de representantes por clase, junto con un mecanismo de regularización que penaliza la mala clasificación de los representantes agregando un peso adicional para el cálculo del *loss*.

1.2 APORTES DEL PROYECTO

Desde una perspectiva investigativa, la principal contribución de este proyecto estuvo orientada a la generación de nuevo conocimiento en el área de aprendizaje incremental supervisado en Deep Learning, específicamente en redes convolucionales, teniendo en cuenta que en la búsqueda realizada en las bases de datos Scopus y ScienceDirect se encontraron pocas propuestas de solución de este problema. Además, en la búsqueda no se encontró ningún artículo que realice una selección de prototipos usando las etiquetas de clase del aprendizaje supervisado como el que se propone en este trabajo (i.e. selección de prototipos definidos por competencia en las fronteras de clase). Por otro lado, se tiene como aporte adicional el análisis comparativo que se realizó entre el algoritmo propuesto y el algoritmo del estado del arte.

Desde la perspectiva de innovación, se desarrolló un módulo que implementó la estrategia de aprendizaje incremental basada en la selección y competencia de prototipos de entrenamiento (con los dos enfoques). Este módulo se construyó en TensorFlow y está disponible para la comunidad académica, científica e industrial, de modo que pueda ser usado en investigaciones futuras y en aplicaciones del mundo real.

Esta investigación se puede aplicar en áreas como el seguimiento visual y la detección de objetos, en donde la propuesta de entrenamiento incremental permite incorporar nuevos ejemplos al modelo a lo largo del tiempo según el entorno y necesidades requeridas por el sistema inteligente. Esto puede ser usado en ámbitos como la agroindustria, eje de desarrollo del Cauca y de Colombia en general, donde puede ser de gran ayuda para apoyar actividades relacionadas con el control de calidad, la clasificación de productos, entre otras actividades.

1.3 OBJETIVOS

A continuación, se presentan los objetivos como fueron aprobados en el Anteproyecto por parte del Consejo de Facultad de la Facultad de Ingeniería Electrónica y Telecomunicaciones.

1.3.1 Objetivo General

Proponer un algoritmo de aprendizaje incremental basado en RMSProp y registros representantes (prototipos) de los datos de entrenamiento, definidos por competencia y agrupamiento en las fronteras de clases, buscando reducir el tiempo de reentrenamiento de redes convolucionales manteniendo la mayor cantidad posible de calidad de clasificación de la red.

1.3.2 Objetivos Específicos

- Establecer la línea base de comparación del trabajo investigativo basado en TensorFlow, contemplando uno de los mejores algoritmos del estado del arte en aprendizaje incremental de redes convolucionales y RMSProp, diversas métricas de comparación (exactitud, precisión, recuerdo, medida F, AOC y tiempo), cuatro problemas de clasificación reconocidos por la comunidad científica como MNIST, SVHN, CIFAR10 y CIFAR100 y la arquitectura de la red convolucional que será usada para cada problema.
- Definir una nueva versión del algoritmo RMSProp que incluya la selección de muestras representativas (prototipos) de las fronteras de los hiperplanos, definidos por datos de entrenamientos de redes convolucionales, basado en un proceso de competencia y agrupamiento, buscando soportar el aprendizaje incremental en subsiguientes etapas de reentrenamiento, implementándolo en TensorFlow para facilitar su integración con la línea base previamente definida.
- Evaluar el desempeño del algoritmo de aprendizaje incremental propuesto, basándose en las medidas de la línea base (por ejemplo, tiempo de entrenamiento y exactitud) y realizar un análisis comparativo de los resultados obtenidos frente a los algoritmos seleccionados en línea base, soportando dicho análisis con las pruebas estadísticas no paramétricas de Friedman y Wilcoxon.

1.4 RESULTADOS OBTENIDOS

A continuación, se resumen los principales resultados del presente trabajo de grado:

1. **Monografía de trabajo de grado**: Se refiere al presente documento el cual presenta el estado del arte en el campo de aprendizaje incremental, la descripción del marco de trabajo para realizar un entrenamiento incremental y la definición e implementación de dos instanciaciones del marco propuesto, por último, se exponen los resultados obtenidos por los dos algoritmos propuestos y se comparan con la línea base y un algoritmo del estado del arte
2. **Framework de pruebas:** Se refiere al código fuente del framework (denominado DILF) construido para realizar las pruebas e implementar los algoritmos de aprendizaje incremental, y al código de las dos instanciaciones del marco de aprendizaje incremental propuesto, denominados NIL y RILBC

que se encuentra disponible en https://github.com/camilonar/incremental-eureka y en un anexo digital (**Anexo A**) de esta monografía. La documentación del código se presenta en el **Anexo B**.

3. **Artículo del framework de pruebas**: artículo titulado **"DILF: Deep Incremental Learning Framework over TensorFlow"** con la definición del framework el cual se encuentra en proceso de evaluación en la revista **SoftwareX** (ISSN 2352-7110), la cual es catalogada como **A1** por Colciencias y clasificada como **Q1** en **SJR**. Ver **Anexo C**.
4. **Artículo final de la investigación**: artículo con la descripción y resultados del marco de aprendizaje propuesto en este trabajo, denominado CRIF, y sus dos instanciaciones, NIL y RILBC, el cual ha sido titulado **"Incremental Learning Model Inspired in Rehearsal for Deep Neural Networks"** y que se encuentra en proceso de envío a una revista JCR Q1, como Knowledge-Based Systems o Information Sciences. Ver **Anexo D**.

1.5 ESTRUCTURA DE LA MONOGRAFÍA

A continuación, se describe de manera general el contenido y organización de la presente monografía:

CAPITULO 1: INTRODUCCIÓN: Hace referencia al presente capitulo que introduce el tema de investigación, presenta la pregunta de investigación que origino el trabajo, los aportes al problema, también los objetivos (general y específicos) definidos en el anteproyecto, un breve resumen de los resultados obtenidos y finalmente la organización de la monografía.

CAPITULO 2: CONTEXTO TEÓRICO Y ESTADO DEL ARTE: En este capítulo se presentan conceptos teóricos relacionados con las redes neuronales convolucionales, su proceso de entrenamiento, el aprendizaje incremental y el framework Tensorflow. Además, se presentan propuestas relevantes del estado del arte que han abordado el problema del aprendizaje incremental.

CAPITULO 3: FRAMEWORK PARA SOPORTAR LA EXPERIMENTACIÓN: En este capítulo se explica la estructura y funcionamiento general del framework desarrollado para la ejecución de los experimentos con los algoritmos, así como un ejemplo de construcción y ejecución de un experimento.

CAPITULO 4: PROPUESTA: En este capítulo se presenta CRIF, un marco de aprendizaje incremental basado en selección de representantes. También se presentan dos instanciaciones de CRIF, denominadas NIL y RILBC.

CAPITULO 5: EXPERIMENTOS Y RESULTADOS: Se presenta el protocolo de experimentación y se presentan los resultados de los algoritmos de la línea base y del algoritmo propuesto en los distintos escenarios de prueba. También se presenta un análisis comparativo de los resultados soportado con las pruebas

estadísticas no paramétricas de Friedman y Wilcoxon y el post Hoc de Holm para la prueba de Friedman.

CAPITULO 6: CONCLUSIONES Y TRABAJOS FUTUROS: En este capítulo se presentan las conclusiones obtenidas al finalizar el trabajo de grado e ideas que el grupo de investigación espera realizar en un trabajo futuro.

CAPITULO 7: BIBLIOGRAFIA: Este último capítulo contiene las referencias bibliográficas de los artículos y libros consultados para la realización del proyecto.

CAPÍTULO 2

2 CONTEXTO TEÓRICO Y ESTADO DEL ARTE

2.1 CONTEXTO TEÓRICO

2.1.1 Redes Neuronales Artificiales (RNA)

Las Redes Neuronales Artificiales (RNA) son una serie de modelos computacionales basados en elementos del funcionamiento del cerebro humano, en los cuales se conectan muchas unidades pequeñas de procesamiento [6].

Las RNA tienen 2 componentes básicos: las unidades de procesamiento (neuronas), y las conexiones entre ellas. Cada neurona recibe estímulos de las neuronas conectadas a ella, procesa la información y genera una salida [38].

Las neuronas se conectan entre sí, y se apilan formando capas, de forma que las salidas producidas por las neuronas de la capa *N* son usadas como estímulos de las neuronas de la capa *N+1*, aunque existen otras arquitecturas más complejas que permiten conexiones hacia las capas de atrás o hacia capas de adelante pero que no son necesariamente la siguiente [38]. En general, las capas se distribuyen en 3 grupos: la capa de entrada, que recibe un vector de entradas (denotado como *X*), una serie de capas ocultas, y una capa de salida que produce un vector con la salida de la red (denotado como *Y*) [1][38]. Esta estructura de redes con propagación hacia adelante (feed-forward) se puede observar en la **Figura 1**.

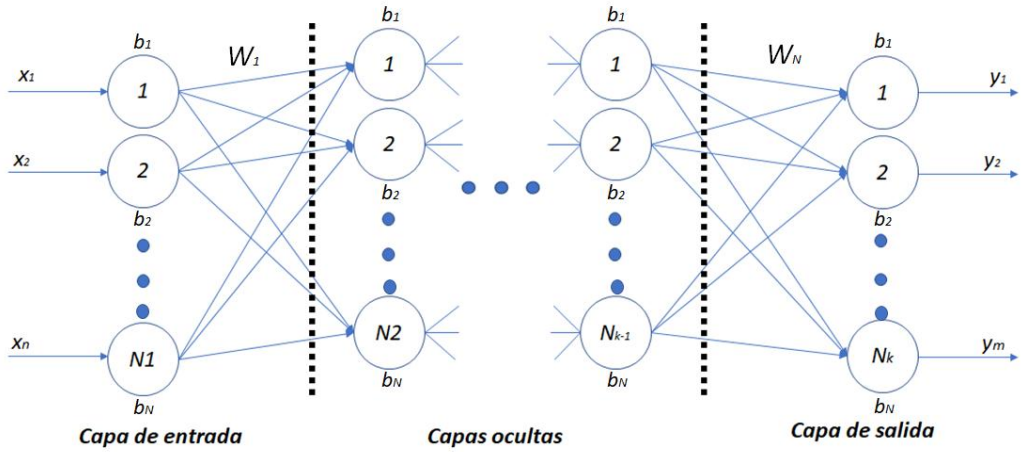

Figura 1 *Estructura básica de una RNA (fuente propia).*

Cada conexión entre neuronas tiene asociado un factor de peso, el cual multiplica el valor de cada entrada. Además, cada neurona tiene asociado un sesgo, que es un valor numérico independiente de la entrada. Para obtener el valor de salida de una neurona se suman las entradas ponderadas por su peso junto con el sesgo, y el resultado pasa a través de una función no lineal de activación [1][38]. De esta forma, la salida de una capa se puede calcular con la Ecuación (1).

$$h(x) = s(W * x + b) \tag{1}$$

donde *b* es el vector de sesgos, W es la matriz de pesos y *s* la función de activación neuronal.

Los valores de los pesos sinápticos y del vector de sesgos son parámetros que la red neuronal puede aprender por medio de un proceso de entrenamiento a través del cual la red logra calcular la salida correcta para cada entrada haciendo generalizaciones a partir de un conjunto de datos de entrenamiento [39]. El problema de entrenamiento entonces se limita a modificar los pesos **W** y sesgos **b** hasta encontrar los valores óptimos que reduzcan el error en la clasificación.

2.1.1.1 Redes Convolucionales

Las redes neuronales convolucionales (CNN) son un tipo de red neuronal de propagación hacia adelante, diseñada para reconocer patrones visuales directamente desde los pixeles de una imagen de una forma eficiente [40]. La forma en que las neuronas de cada capa están conectadas se inspira en la organización de la corteza visual de los animales, donde la salida de cada neurona puede representarse como una operación de convolución [41].

Su arquitectura tiene muchas variaciones; pero por lo general, están conformadas por capas convolucionales y capas de submuestreo (pooling) agrupadas en módulos, seguidos de una o más capas completamente conectadas [42], como se puede observar en la **Figura 2**.

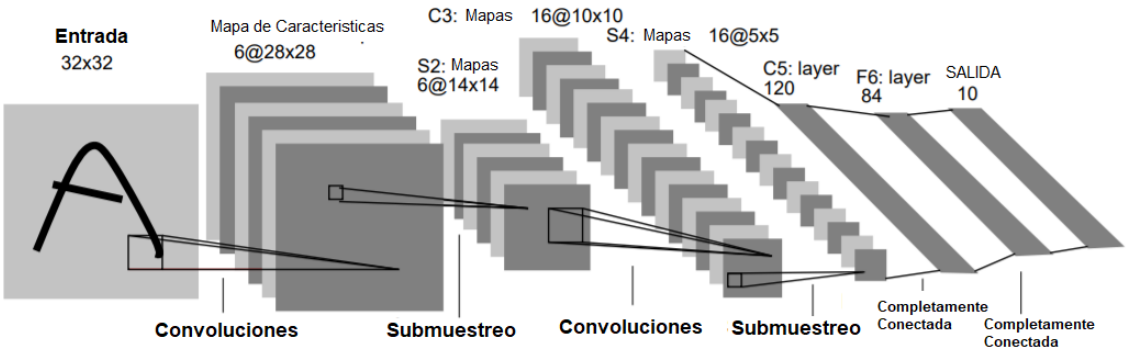

Figura 2 *Estructura Básica de una CNN (adaptada de* [40]*)*

2.1.1.2 Capas Convolucionales

Las capas convolucionales se encargan de extraer características de la imagen por medio de filtros que la red aprende. Cada filtro genera una abstracción de la imagen de entrada e identifica una característica en particular, como bordes, esquinas o figuras más complejas, de esta manera, la salida de la capa convolucional es una imagen más pequeña pero más profunda, donde la profundidad depende de la cantidad de filtros de la capa convolucional [40].

En las capas convolucionales las neuronas comparten los mismos pesos y no están totalmente conectadas entre sí, sino que únicamente se conectan con una región de la capa anterior [40].

En la **Figura 3** se puede observar un ejemplo de la operación de convolución donde el valor resultante se obtiene multiplicando el valor de entrada con el valor correspondiente en el filtro y sumando los resultados. Este proceso se repite desplazando el filtro por toda la entrada. Cabe destacar que la distancia vertical y horizontal al centro del filtro será la cantidad de filas y columnas que van a desaparecer en la imagen de salida [43].

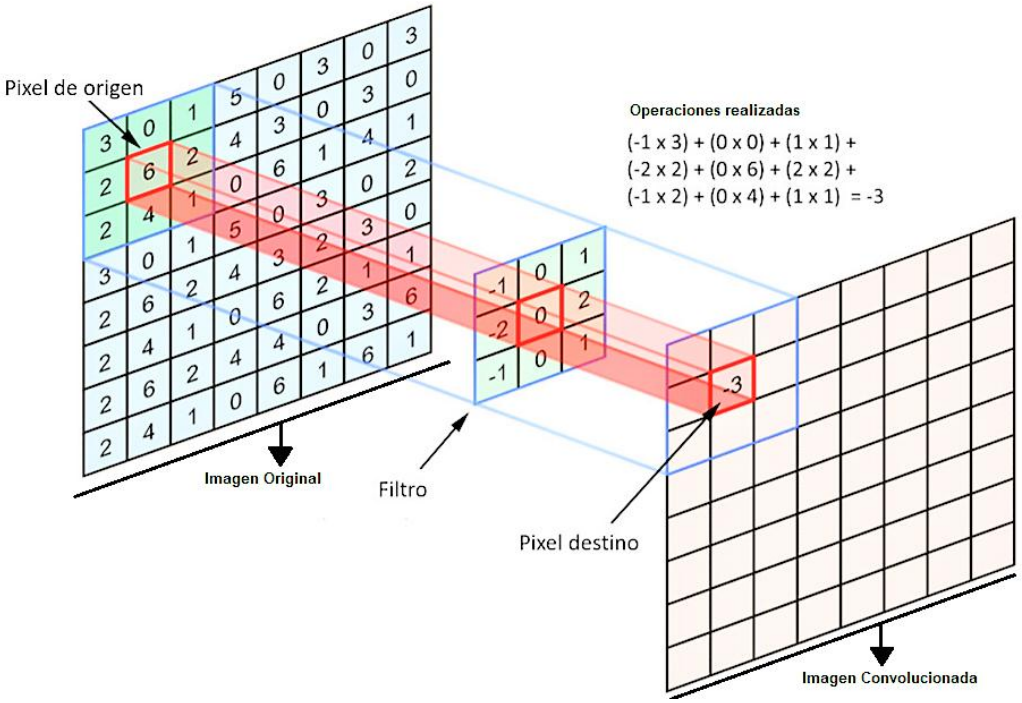

Figura 3 *Aplicación de filtro Convolucional (tomada de* [44]*)*

2.1.1.3 Capa Pooling

El proceso de pooling se lleva a cabo para reducir el tamaño de la entrada, y se realiza por medio de un filtro que se encarga de agrupar varios valores de entrada en uno. Por ejemplo, se puede transformar una entrada de 10x10 en una de 5x5 al hacer uso de una capa de Pooling. Esta operación se realiza principalmente para

evitar el sobre ajuste y reducir la carga computacional. Los dos métodos de agrupamiento más usados son Max Pooling (elegir el valor más grande) y Average Pooling (obtener el valor promedio) [43].

2.1.2 RMSProp

RMSProp es un algoritmo de entrenamiento que es ampliamente usado en Deep Learning [9], tanto en CNN [10] como en otras arquitecturas de red [11]. En esencia, este algoritmo es una variación de Back Propagation [6] que funciona con mini lotes y que, para la actualización de pesos de la red, hace uso de un gradiente que es dividido por un promedio de sus valores anteriores [8].

El algoritmo RMSProp tiene como propósito evitar grandes variaciones del gradiente, debido a que en un entrenamiento normal puede tomar valores radicalmente diferentes para diferentes pesos. Para solucionar esto, se divide el gradiente de un mini-lote entre un promedio de sus valores anteriores, de modo que mini-lotes adyacentes sean divididos por un valor similar [8]. Este valor es llamado **media cuadrada**, y se calcula por medio de la Ecuación (2).

$$MediaCuadrada(w,t) = 0.9\, MediaCuadrada(w,t-1) + 0.1(\frac{\partial E}{\partial w}(t))^2 \qquad (2)$$

Donde t es el mini-lote actual, y $\frac{\partial E}{\partial w}(t)$ es el error de la salida de la red respecto a las etiquetas correctas de los datos del mini-lote. Este último valor es conocido como **gradiente**.

La Ecuación (3) se usa para actualizar los valores de los pesos durante el entrenamiento. En ella, el gradiente se divide entre la media cuadrada para calcular la velocidad actual (V), que es equivalente al valor en el que deben variar los pesos (Δw). Adicionalmente, se usa una tasa de aprendizaje (α) para regular la velocidad a la cuál varían los pesos durante el entrenamiento [8].

$$\Delta w(t) = V_t = \alpha\, \Delta w(t-1) - \frac{1}{\sqrt{MediaCuadrada(w,t)}} * \frac{\partial E}{\partial w}(t) \qquad (3)$$

2.1.3 Aprendizaje incremental

El aprendizaje incremental es una forma de entrenar modelos de Machine Learning para que se adapten constantemente a un flujo de datos que cambia en el tiempo, de modo que no sea necesario realizar un reentrenamiento completo cada vez que se obtienen nuevos datos [13]. Evitar un reentrenamiento es importante debido a que algunos modelos son sumamente complejos y requieren gran cantidad de tiempo y recursos computacionales para su entrenamiento [15].

En algunas situaciones, como el procesamiento de Big Data, la robótica y el etiquetado automático, los datos surgen de entornos dinámicos y se adquieren a lo largo del tiempo [13]. Si no se realiza un aprendizaje constante, los modelos se

desactualizan y reducen su precisión, debido a que se pierde la oportunidad de entrenar con nuevos datos y así obtener un mejor modelo [13][14].

En su núcleo, el aprendizaje incremental consiste en entrenar un modelo haciendo uso únicamente de los nuevos datos que llegan al modelo, junto a unos cuántos datos usados en entrenamientos previos o a una representación de estos (e.g. estadísticas sobre los datos previamente vistos) [13]. Este enfoque hace que surja el dilema de la estabilidad-plasticidad, que planeta la dificultad de balancear el conocimiento previo, a la vez que se mantiene la capacidad de aprender [16].

En la ***Figura 4*** se muestran los resultados pre-entrenamiento y post-entrenamiento de una red neuronal hipotética que aprende un único patrón. La red aprende correctamente los pesos necesarios para lograr la salida deseada para el patrón para el cuál ha sido entrenada, sin embargo, en un ambiente real la red debe ser capaz de adaptarse a múltiples nuevos patrones que son suministrados a lo largo del tiempo (plasticidad) a la vez que mantiene el reconocimiento de los patrones previamente aprendidos (estabilidad) [45]. Lograr este equilibrio es uno de los problemas fundamentales del aprendizaje incremental [20].

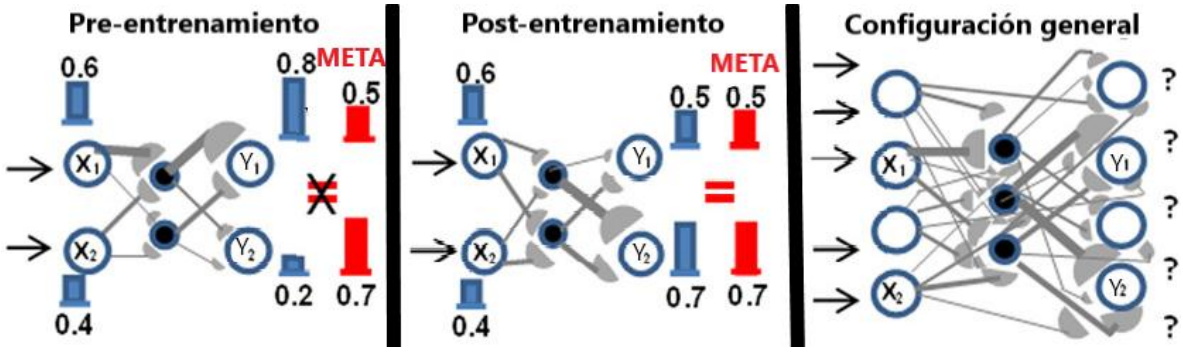

Figura 4 *Dilema de la estabilidad-plasticidad (adaptada de* [45]*)*

2.1.4 Transfer Learning

Transfer Learning, es una técnica de aprendizaje incremental que reutiliza el conocimiento adquirido al resolver un problema para facilitar la solución de un problema diferente pero relacionado, por ejemplo, el conocimiento adquirido para reconocer características de rostros puede ser aplicado para reconocer objetos. De esta forma se busca transferir tanto conocimiento como sea posible de una solución de origen a una solución de destino [46].

Para realizar *transfer learning* en una CNN se requiere conservar los pesos de las capas convolucionales que capturan la información general de cómo se compone una imagen. En las primeras capas de una CNN se capturan características de bajo nivel como bordes o esquinas y se detectan objetos complejos en capas posteriores, mientras que el proceso de clasificación se realiza en las capas completamente conectadas. De esta manera para realizar *transfer learning* se hace uso de las capas convolucionales de una red ya entrenada para construir el

nuevo modelo, que será entrenado congelando o ajustando ligeramente los parámetros de cada capa convolucional que se desee conservar para no perder los conocimientos adquiridos previamente [47].

2.1.5 TensorFlow

TensorFlow (TF) es una librería de código abierto usada ampliamente en el campo del aprendizaje automático y las redes neuronales profundas. Su estructura usa grafos para representar el flujo de datos, donde los nodos contienen variables u operaciones matemáticas y las aristas reflejan las matrices de datos multidimensionales, denominadas **tensores**. Un tutorial básico de TF se encuentra en: https://www.edx.org/es/course/deep-learning-with-tensorflow.

A continuación, se presentan diferentes conceptos de la librería, que son importantes para entender la solución propuesta en este trabajo.

2.1.5.1 TFRecord

TFRecord es un formato binario orientado a registros usado para guardar datos de entrenamiento. La clase ***tf.data.TFRecordDataset*** permite transmitir el contenido de uno o más archivos TFRecord como parte de un *Pipeline* de entrada.

2.1.5.2 Pipelines

Las tuberías (***Pipelines)*** en TF se encargan de entregar los datos del siguiente paso de entrenamiento antes de que el paso actual finalice. Para realizar un paso se requiere primero extraer y transformar los datos para alimentar el modelo que se ejecuta en un acelerador GPU (Modelo **ETL**).

Los Pipelines superponen el preprocesamiento y la ejecución de un paso de entrenamiento: mientras la GPU(s) está entrenando, la CPU prepara los datos para el próximo entrenamiento. TF provee la API *tf.data,* la cual ayuda a crear pipelines flexibles y eficientes. En la ***Figura 5*** se muestra un ejemplo de cómo funciona esta API.

E
```
files = tf.data.Dataset.list_files(file_pattern)
dataset = tf.data.TFRecordDataset(files)
```
T
```
dataset = dataset.shuffle(10000)
dataset = dataset.repeat(NUM_EPOCHS)
dataset = dataset.map(lambda x: tf.parse_single_example(x, features))
dataset = dataset.batch(BATCH_SIZE)
```
L
```
iterator = dataset.make_one_shot_iterator()
features = iterator.get_next()
```

Figura 5 *API tf.data*

En la fase de **transformación** los datos se desordenan y se aplica una función ***map*** que modifica cada uno de los elementos del dataset. Estas modificaciones incluyen descompresión de imágenes y aumento de datos (como recortes aleatorios, rotación de imagen y distorsiones de color). En el código de la ***Figura 5*** únicamente se realiza una descomprensión de cada imagen. Por último para realizar la fase de cargue de datos se debe indicar a TF cómo obtener los datos, para lo cual se crea un iterador de la clase ***tf.data.Iterato*****r** para tener un acceso secuencial a los elementos del dataset.

Un tutorial más detallado acerca del manejo de datasets y pipelines se puede encontrar en: https://towardsdatascience.com/how-to-use-dataset-in-tensorflow-c758ef9e4428

2.1.5.3 TensorBoard

TensorBoard (TB) es una herramienta web que permite visualizar parámetros de entrenamiento, métricas, hiperparámetros o estadísticas del modelo para inspeccionar y comprender las ejecuciones. TB opera leyendo archivos de evento, los cuales contienen un resumen de los datos que son generados cuando TF se está ejecutando. TB también permite visualizar el grafo computacional con el cual se está trabajando y actualmente proporciona cinco tipos de visualizaciones: escalares, imágenes, audio, histogramas y gráficos.

Para exportar las operaciones de TF a un archivo de eventos (Log file) se necesita crear un ***tf.summary.FileWriter***, el cual recibe dos parámetros: la carpeta donde se guardarán los archivos de eventos, y el grafo computacional con el cual se está trabajando (el grafo ya debe estar completamente creado).

TB proporciona una operación llamada ***summary*** para visualizar parámetros del modelo, como pueden ser los pesos o sesgos de una red, métricas como exactitud o pérdida (loss) y otros valores que ayuden a comprender el comportamiento del modelo que se está construyendo.

Hay 3 tipos de ***summaries***, ***tf.summary.scalar*** que almacena un solo valor, ***tf.summary.histogram*** usado para almacenar valores no escalares como pesos o el sesgo de una red neuronal y por último tf***.summary.image*** usado para almacenar imágenes.

Un tutorial más detallado de TensorBoard se puede encontrar en: https://www.datacamp.com/community/tutorials/tensorboard-tutorial

2.1.5.4 Checkpoints

Los puntos de control (***checkpoints***) son una forma de guardar el estado actual del entrenamiento de un modelo, de manera que se pueda reanudar su entrenamiento en el futuro. Estos checkpoints son útiles para reanudar entrenamientos después de fallos o para experimentar distintas estrategias con los modelos.

Los checkpoints contienen la información que se necesita para guardar el estado actual del experimento, como la arquitectura del modelo, los pesos, sesgos, configuración de entrenamiento y el estado del optimizador. En TF estos valores son llamados **Variables**, y pueden ser almacenados fácilmente mediante el uso de un ***Saver.***

La clase *Saver* permite guardar y restaurar las variables mediante las operaciones ***save*** y ***restore***, respectivamente. Para estas operaciones se debe especificar la ruta hacia el archivo de puntos de control en el que se desea escribir o leer.

2.2 ESTADO DEL ARTE

A continuación, se presentan trabajos previos relacionados con aprendizaje incremental, desde los tres enfoques principales encontrados en la literatura, a saber: aprendizaje activo, cambio dinámico de la estructura del modelo y ensamble de modelos. Además, se presentan trabajos previos relacionados con el desbalanceo de clases, tema que aporta ideas fundamentales para la selección apropiada de representantes o prototipos para el presente trabajo.

2.2.1 Aprendizaje activo

Aprendizaje activo basado en teoría de Scaffolding (gClass) [26]**:** propuesto en 2015 como una red de tipo Fuzzy que hace uso de las teorías de aprendizaje de Scaffolding y Schema y tiene un enfoque plug-and-play, lo que significa que no hay etapa de pre o pos entrenamiento. Utiliza un método de conflicto-ignorancia para decidir qué muestras son seleccionadas para el entrenamiento. Para esto, genera clústeres elipsoidales que agrupan a los datos calculando el conflicto con base en el hecho de que una instancia esté en múltiples clústeres. La ignorancia se mide de acuerdo a la distancia con respecto a los clústeres (si la distancia es muy grande, la ignorancia es grande).

Aprendizaje activo en redes de creencia profunda basado en Best vs. Second Best (BvSB-ADN) [27]**:** fue propuesto en 2016, y combina aprendizaje activo con entrenamiento semi-supervisado. En este algoritmo se realiza inicialmente una etapa de entrenamiento no supervisado, y posteriormente se agregan muestras etiquetadas a un pool de datos para entrenamiento supervisado basándose en su incertidumbre, la cual es obtenida mediante BvSB. En BySB se calcula la diferencia entre la probabilidad más alta, y la segunda más alta de pertenencia a una clase. Este proceso se repite, de modo que en cada iteración el pool de muestras etiquetadas crece.

Aprendizaje incremental de diccionario ponderado (WI-DL) [29]**:** propuesto en 2016 para la clasificación de imágenes hiper espectrales con Redes de Creencia Profunda (DBN). WI-DL selecciona las muestras para el aprendizaje activo mediante la combinación de medidas de incertidumbre y representatividad. La incertidumbre es medida mediante el cálculo de la entropía, mientras que la

representatividad se mide a través de la comparación de las muestras con un diccionario de muestras preseleccionadas. El diccionario se construye así: primero se escoge un conjunto de muestras iniciales basándose en su incertidumbre, luego se calcula la representatividad (similitud) de cada muestra del diccionario respecto a las demás, y finalmente, se reemplazan las muestras por nuevas con el menor puntaje. Es importante mencionar que el diccionario guarda la salida de la DBN de las muestras y no las muestras mismas.

WALI [30]**:** propuesto en 2016 para realizar reconocimiento visual sobre un flujo constante de datos. Se compone de 4 etapas: mirar, preguntar, aprender y mejorar. En el primer paso se obtienen imágenes sin categorizar a partir de videos. Luego se escogen muestras representativas que se etiquetan manualmente, para lo cual se usan medidas de incertidumbre y un clasificador binario para calcular la probabilidad de rechazo, de forma que, si un grupo de muestras ha sido rechazado previamente, se tenderá a rechazar muestras similares en el futuro. Finalmente, se entrenan clasificadores lineales por medio de regresión lineal. Este proceso se repite por un tiempo indefinido, de manera que el algoritmo mejore constantemente.

Aprendizaje activo y semisupervisado (ASSL) [25]**:** propuesto en 2017 para la tarea de detección de objetos mediante redes convolucionales. Se propuso un proceso incremental en el que en cada lote de datos se presenta al clasificador para asignar pseudo-etiquetas a las instancias, y se escogen las muestras con mayor incertidumbre. Posteriormente, se aplica el algoritmo k-means para formar clústers de las instancias y luego se escogen muestras de cada clúster para etiquetarlas de forma que las muestras tengan poca redundancia. Finalmente, se reentrena la red con los datos escogidos y el proceso se repite hasta lograr convergencia o cumplir un criterio de parada.

Aprendizaje activo y online para reconocimiento de objetos 3D [28]**:** propuesto en 2017, usa una red convolucional preentrenada para reconocer objetos mediante registro de colores y profundidad, la cual provee mapas de características que son usados por un clasificador externo: el *Mondrian Forest*. Utilizan 2 estrategias de aprendizaje activo para entrenar al clasificador. En la primera se escogen muestras cuya incertidumbre sea mayor que un valor previamente fijado, y en la segunda se dividen las muestras en lotes y se organizan según su incertidumbre, escogiendo un porcentaje de las muestras con mayor incertidumbre en cada lote.

En la **Tabla 1** se muestra un resumen de los algoritmos de aprendizaje activo y las principales diferencias con el algoritmo propuesto en este trabajo.

Tabla 1 *Algoritmos de aprendizaje activo*

Ref.	Nombre	Enfoque	Diferencias
[26]	gClass	Red Fuzzy que usa un	Usa conceptos de la teoría de

	(2015)	método de conflicto-ignorancia basado en clusters para aprendizaje activo	Scaffolding en un método diseñado exclusivamente para redes Fuzzy
[27]	BvSB-ADN (2016)	Incertidumbre calculada con BvSB	Selecciona los representantes directamente y no hace uso de candidatos
[29]	WI-DL (2016)	Uso de un diccionario para calcular incertidumbre y representatividad	Las métricas usadas para escoger representantes son diferentes a las usadas en este trabajo
[30]	WALI (2016)	Uso de medidas de incertidumbre y probabilidad de rechazo	WALI sólo se puede usar con clasificadores lineales
[25]	ASSL (2017)	Uso de K-Means sobre muestras con alta incertidumbre	ASSL usa un reentrenamiento constante con un conjunto de datos en constante crecimiento
[28]	Ap. Activo y online (2017)	Reconocimiento de objetos 3D con aprendizaje activo usando *Mondrian Forest*	Utiliza muestreo por incertidumbre, pero no por representatividad

2.2.2 Cambio dinámico de la estructura del modelo

Red neuronal incremental para clasificación y clustering (INNCC) [22]**:** fue propuesto en 2015, es una red capaz de aprender el número de nodos necesarios para representar cada clase. Utiliza aprendizaje semisupervisado, de modo que usa un pequeño conjunto de datos etiquetados para construir un clasificador inicial y luego usa los datos no etiquetados para terminar de construir la red, agregando o eliminando neuronas en el proceso. También usan Support Vector Machines (SVM) para mejorar y acelerar la clasificación de INNCC durante su entrenamiento.

Aprendizaje incremental online para series de tiempo (OILFTS) [23]**:** propuesto en 2015, utiliza una red autoorganizada incremental (SOINN) que inicialmente tiene sólo 2 nodos. Para entrenar la red, se leen series de tiempo y se inserta un par de nodos por cada serie de tiempo si la distancia de la serie de tiempo hacia los nodos ya existentes es mayor que un valor previamente fijado. Para eliminar el ruido, se utiliza la regla de Hebbian para enlazar cada par de nodos asociado a una serie de tiempo, eliminando los nodos que tengan menos de 2 vecinos.

Red Convolucional Adaptativa (ACNN) [48]**:** propuesto en 2016 para la tarea de reconocimiento de rostros. Comienza con una red convolucional básica de una rama, con 2 capas convolucionales y 2 capas de subsampling. Posteriormente el algoritmo determina si la red actual converge en cierto número de iteraciones, y en

caso de que no lo haga genera una nueva rama global. Cuando el error sea menor que un límite preestablecido, pero mayor que el error objetivo del entrenamiento entonces se realiza una fusión entre ramas. En el caso de aprendizaje incremental, se conservan los pesos de la red original y se generan ramas globales para cubrir el conocimiento de las nuevas muestras.

Redes Fuzzy dinámicas basadas en Incremental Similarity (IS) [14]**:** propuesto en 2016 para resolver problemas de clasificación, y específicamente en la clasificación de símbolos manuscritos. Aplican aprendizaje online (se aprende cada vez que llega una nueva muestra) con 4 modelos Fuzzy que tienen reglas del tipo SI *x* ES *a* ENTONCES *b*, de modo que *x* ES *a* se deriva usando una medida de similitud incremental, y se crean nuevas reglas dinámicamente. La similitud incremental mide la distancia de todos los puntos en una regla, y se usan parámetros y fórmulas de actualización de dichos parámetros, para así conservar el conocimiento previo.

Aprendizaje perpetuo usando una red Fuzzy Tipo 2 [21]**:** propuesto en 2016 para el aprendizaje perpetuo offline. Se parte de una estructura de reglas inicial que se expande incrementalmente. Adicionalmente se hace una poda de las reglas regularmente, para que la red no crezca demasiado. Cuando llegan nuevas instancias se mide la distancia a un clúster generado previamente para determinar si los datos son "nuevos" (i.e. no pertenecen a las categorías que se pueden identificar) o semi-nuevos (i.e. no estan completamente cubiertos) y con base en estos datos se generan las nuevas reglas.

Modelo evolutivo para aprendizaje online sobre flujos de datos sin almacenamiento [49]**:** propuesto en 2017 como un algoritmo incremental que usa como apoyo un tipo especial de red neuronal generativa llamada DCGAN, la cual tiene la capacidad de generar muestras artificiales de la clase con la cual se entrenó. El algoritmo construye una red DCGAN cada vez que aparece una nueva clase, y entrena a una red convolucional (clasificador) usando las nuevas muestras junto con las muestras generadas por todas las redes DCGAN previamente creadas. Debido a que se crea una red por cada clase, el algoritmo tiene problemas de escalabilidad cuando hay un gran número de clases.

En la **Tabla 2** se muestra un resumen de los algoritmos de cambio dinámico de la estructura del modelo y las principales diferencias con el algoritmo propuesto en este trabajo.

Tabla 2 *Algoritmos de cambio dinámico de la estructura del modelo*

Ref.	Nombre	Enfoque	Diferencias
[22]	INNCC (2015)	Aprendizaje semisupervisado en red que genera nodos automáticamente	Define el número de neuronas para generar un modelo, más no escoge representantes para el entrenamiento

[23]	OILFTS (2015)	Red autoorganizada para series de tiempo	Diseñado exclusivamente para análisis de series de tiempo
[48]	ACNN (2016)	CNN que genera y fusiona ramas globales	Gestiona una red CNN dinámica, pero no selecciona representantes para el entrenamiento
[14]	IS (2016)	Red Fuzzy con reglas generadas a partir de similitud incremental	Algoritmo centrado en la generación de reglas Fuzzy
[21]	Fuzzy Tipo-2 (2016)	Red Fuzzy con reglas generadas a partir de clusters	Algoritmo centrado en la generación y poda de reglas Fuzzy
[49]	DCGAN (2017)	Redes DCGAN que generan muestras sintéticas de cada clase	Requiere una red DCGAN por cada clase, por lo que su escalabilidad es menor

2.2.3 Ensamble de modelos

Learn++ [20]**:** fue propuesto en 2002, es un algoritmo de aprendizaje incremental que permite a redes neuronales supervisadas, como el Perceptrón multicapa, incorporar nueva información a su modelo y agregar nuevas clases. Esta propuesta no requiere de un acceso a la información anterior para los subsiguientes entrenamientos incrementales. Se basa en el algoritmo AdaBoost, donde se combinan los resultados de varios clasificadores débiles para obtener un clasificador robusto. En esencia Learn++ y AdaBoost generan un conjunto de clasificadores débiles. Las salidas de estos clasificadores se combinan utilizando el esquema de votación por mayoría para obtener la clasificación final.

Red Convolucional con Boosting Incremental (IB-CNN) [18]**:** propuesto en 2016 para la tarea de detección de expresiones faciales. Está formada por una red convolucional estándar y una capa de boosting que se conecta al final de la red para realizar la clasificación. La capa de boosting registra las neuronas (clasificadores débiles) que han sido activadas en una iteración del entrenamiento, y también las que se activaron en iteraciones previas. Para formar al clasificador fuerte se calcula el peso de clasificación de cada neurona de la capa de boosting y se combina la salida mediante AdaBoost estándar. Durante el entrenamiento se incrementan los pesos de las neuronas más usadas y se disminuyen los pesos de las neuronas poco usadas.

Redes Convolucionales Colaborativas (OCN) [19]**:** es una arquitectura de red propuesta en 2016 para la tarea de seguimiento de objetos en un video. Se divide en 2 partes, (i) el aprendizaje de características mediante K-Means y, (ii) la generación de la salida de la red mediante un algoritmo de regresión que genera un mapa de puntuaciones cuyo pico indica la posición del objeto en la imagen. Para tratar el dilema de la estabilidad-plasticidad se usan 2 OCN de forma

conjunta: una red tiene memoria de corto plazo y la otra se enfoca en el largo plazo.

Aprendizaje perpetuo basado en combinación dinámica de clasificadores [17]**:** en 2017 se propone un marco de aprendizaje vitalicio, donde el modelo de clasificación evoluciona de manera constante, aprendiendo nuevos conocimientos sin olvidar lo ya aprendido. Este marco se basa en la construcción de múltiples sub clasificadores cuando se procesan nuevos datos, los cuales se combinan dinámicamente mediante un árbol de decisión, el cual utiliza un método de poda para evitar el sobre entrenamiento y eliminar los modelos obsoletos.

iCaRL [34]**:** propuesto en 2017 como un algoritmo de clasificación incremental para problemas de visión artificial. Este algoritmo hace uso de una CNN para extraer características de los datos de entrenamiento y así seleccionar muestras representantes que son usadas en un enfoque de clasificación ***Nearest-Mean-of-Exemplars***, donde se calcula un vector prototipo equivalente a la media de cada clase y se clasifica las muestras según la cercanía a dichos vectores. Cuando llegan nuevos datos se realiza el proceso de entrenamiento mezclando los nuevos datos con los representantes.

En la **Tabla 3** se muestra un resumen de los algoritmos de ensamble de modelos y las principales diferencias con el algoritmo propuesto en este trabajo.

Tabla 3 *Algoritmos de ensamble de modelos*

Ref.	Nombre	Enfoque	Diferencias
[20]	Learn++(2002)	Boosting básico	Hace uso de múltiples clasificadores débiles, y no de un solo clasificador fuerte
[18]	IB-CNN (2016)	Capa de Boosting en una CNN	La asignación de pesos ponderados se usa en las neuronas de la capa de salida y no en las muestras de entrenamiento
[19]	OCN (2016)	CNN colaborativas: memoria de corto y largo plazo	La memoria de corto y largo plazo se guarda a través de modelos CNN y no de datos de entrenamiento
[17]	Ap. Perpetuo (2017)	Boosting con árbol de decisión	Hace uso de múltiples clasificadores débiles, y no de un solo clasificador fuerte
[34]	ICaRL (2017)	Representantes con clasificación *Nearest-Mean-of-Exemplars*	Los prototipos se seleccionan con base en la cercanía a la media de la clase, y sólo funciona en un ambiente con clases estrictamente incrementales (cada clase nueva aparece en un nuevo bloque de datos)

2.2.4 Desbalanceo de clases

Técnica de submuestreo de minorías sintéticas (SMOTE) [31]**:** propuesto en 2002, es un enfoque para la construcción de clasificadores a partir de conjuntos de datos desequilibrados (conjuntos donde las categorías de clasificación tienen amplias diferencias en la cantidad de datos que los representan). SMOTE genera instancias sintéticas o artificiales para equilibrar los conjuntos de datos, basado en la regla del vecino más cercano, donde a cada una de las instancias del conjunto minoritario se le buscan las instancias vecinas y se generan nuevas instancias realizando una interpolación entre la instancia original y las instancias vecinas.

Aprendizaje incremental de concept drift para clasificación de datos desequilibrados [32]**:** propuesto en 2013, combina Learn++ y SMOTE en un nuevo algoritmo denominado Learn++.CDS, con el propósito de resolver el problema de desbalanceo de clases en aprendizaje incremental, mediante el uso de una matriz de pesos de penalización y muestras sintéticas. En esta propuesta se cuenta con múltiples clasificadores, donde aquellos que tienen mejor desempeño se utilizan con mayor frecuencia y los que tienen un desempeño muy pobre se van eliminando.

Aprendizaje dinámico de desbalanceo de clase para una LPSVM incremental (DCIL) [36]**:** propuesto en 2013, es un enfoque de aprendizaje dinámico de desbalanceo de clases (DCIL) en un ambiente de aprendizaje incremental de un tipo específico de máquina de soporte vectorial (SVM) llamado LPSVM. Hace uso de una matriz de pesos en la cual cada clase tiene un peso distinto. En cada entrenamiento incremental los pesos se actualizan para reflejar los cambios que puedan ocurrir en la distribución de las clases y mediante estrategias de ponderación sofisticadas toma la decisión final de la clasificación.

CURE-SMOTE [33]**:** propuesto en 2017, es un algoritmo usado en Random Forest para equilibrar conjuntos de datos desbalanceados mediante la combinación del algoritmo de clustering con representantes CURE y el algoritmo para generar muestras sintéticas SMOTE. La idea general del algoritmo consiste en agrupar las muestras de la clase minoritaria utilizando CURE, eliminar el ruido y los valores atípicos de las muestras originales y posteriormente generar muestras artificiales aleatoriamente entre los puntos representativos y el punto central usando SMOTE.

Balanceado basado en distribución (DBB-RDNN-ReL) [35]**:** propuesto en 2018 para el filtrado de spam. Usa un Perceptron Multicapa Regularizado para manejar grandes volúmenes de datos. Para corregir el desbalanceo se agregan muestras artificiales basándose en la distribución Gaussiana de cada característica (valor de entrada) y de cada clase.

Peso Sensible al costo y Bagging con Desbalanceo inverso (CWIB) [37]**:** propuesto en 2018 para clasificar precios de la electricidad. Consiste en un ensamble de ensambles: se usan clasificadores unidos por una suma ponderada

que a su vez están compuestos por clasificadores base que usan un voto de mayoría simple. Cuando llega un nuevo conjunto de datos se realiza un muestreo aleatorio de modo que se obtengan más muestras de la clase minoritaria que de la mayoritaria. Adicionalmente, los pesos de los clasificadores se modifican, favoreciendo a los que tengan mayor precisión y menor sensibilidad.

En la **Tabla 4** se muestra un resumen de los algoritmos de desbalanceo de clases y los principales aspectos que fueron incorporados a la propuesta del presente trabajo.

Tabla 4 *Algoritmos que abordan el desbalanceo de clases*

Ref.	**Nombre**	**Enfoque**	**Aporte**
[31]	SMOTE (2002)	Instancias sintéticas creadas mediante interpolación	Utilizar muestras para equilibrar la proporción de datos asegurando una buena distribución en el espacio
[32]	Learn++.CDS (2013)	SMOTE combinado con Learn++	Pesos ponderados y uso de muestras para balancear los datos
[36]	DCIL (2013)	Matriz de pesos de clases usada en LPSVM	Pesos ponderados por clase
[33]	CURE-SMOTE (2017)	Instancias sintéticas creadas usando CURE e interpolación	Realizar una buena distribución de muestras usando puntos representativos y centrales
[35]	DBB-RDNN-ReL (2018)	Instancias sintéticas creadas a partir de la distribución Gaussiana de los datos	Utilizar muestras para el balanceo de datos
[37]	CWIB (2018)	Pesos dinámicos y bagging con inversión de balanceo de clases	Pesos ponderados para dirigir el entrenamiento

CAPÍTULO 3

3 FRAMEWORK PARA SOPORTAR LA EXPERIMENTACIÓN

3.1 DESCRIPCIÓN DEL FRAMEWORK (DILF)

Teniendo en cuenta que en el presente trabajo se requería realizar un amplio conjunto de experimentos, donde se comparan distintos algoritmos de aprendizaje con diferentes combinaciones de redes neuronales y datasets, se realizó un análisis sobre el uso directo de TF o de librerías de alto nivel como Keras.

Se determinó que usar TF de manera directa hacia demasiado tediosa y elevaba la complejidad de la tarea, debido al gran número de funcionalidades que TF posee y a que, al ser una librería, no proporciona una estructura o arquitectura sobre la cuál construir una solución (como si lo haría un framework).

Por otro lado, con las API de alto nivel como Keras [50], la flexibilidad de TF se ve drásticamente reducida y para trabajos de investigación como este, donde se requiere manejar algunos aspectos de bajo nivel, su utilidad es poca o nula.

Además, se observó que muchas de las soluciones encontradas al momento de realizar este trabajo (en Deep Learning en general y en incremental learning en particular), son soluciones en scripting (Python) que no cuentan con una apropiada arquitectura y no siguen principios básicos de diseño y documentación, lo cual dificulta la replicación de resultados (repetibilidad de los experimentos) y realizar comparaciones debido a la gran complejidad que tiene su comprensión y extensibilidad. Por este motivo se decidió crear una framework que fuese fácil de adaptar y que pudiera ser usada para la construcción de diferentes algoritmos.

A partir del análisis realizado se diseñó e implementó *Deep Incremental Learning Framework* (DILF) [51], un marco de trabajo de acceso libre y código abierto dividido en módulos independientes, que poseen métodos conectores que permiten eliminar gran parte de la verbosidad y complejidad de TF, de modo que el enfoque se centre en la investigación y construcción de nuevos algoritmos de aprendizaje, y no en la cantidad de operaciones que posee TF.

En la **Figura 6** se muestra una vista de alto nivel de DILF. Las clases de color azul corresponden al núcleo del framework, y las de color verde son clases derivadas asociadas a implementaciones de algoritmos específicos.

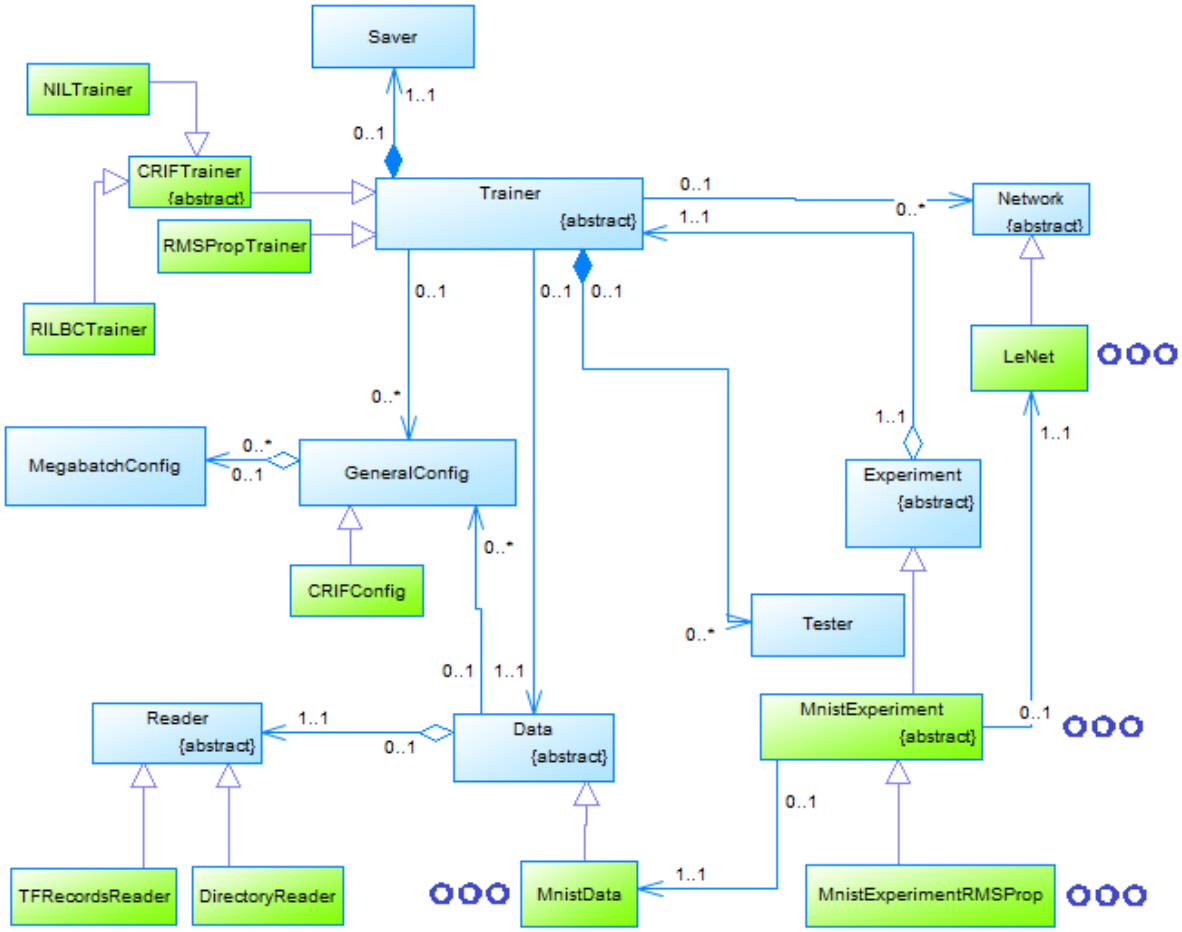

Figura 6 *Diagrama de clases de alto nivel de DILF*

DILF se ha dividido en 4 módulos:

- **Módulo ETL:** centrado exclusivamente en el cargue de datos para el entrenamiento y pruebas
- **Módulo de Redes:** permite integrar fácilmente nuevas arquitecturas de redes neuronales
- **Módulo de Entrenamiento:** se encarga del entrenamiento de una red. Está diseñado para agregar nuevos algoritmos de entrenamiento sin afectar a los demás módulos
- **Módulo de Experimentos:** permite crear y realizar pruebas sobre múltiples arquitecturas, algoritmos y datasets

En esta sección se explica brevemente cada uno de los módulos mencionados. Adicionalmente, una documentación completa de DILF se encuentra disponible en el **Anexo B**.

3.1.1 Módulo ETL

En la **Figura 7** se muestra el módulo encargado de realizar el proceso de **ETL**, Extracción, Transformación y Carga (Load) de datos, que usa como base la clase **Dataset** de TF, la cual permite paralelizar operaciones gracias a las funciones que ya provee.

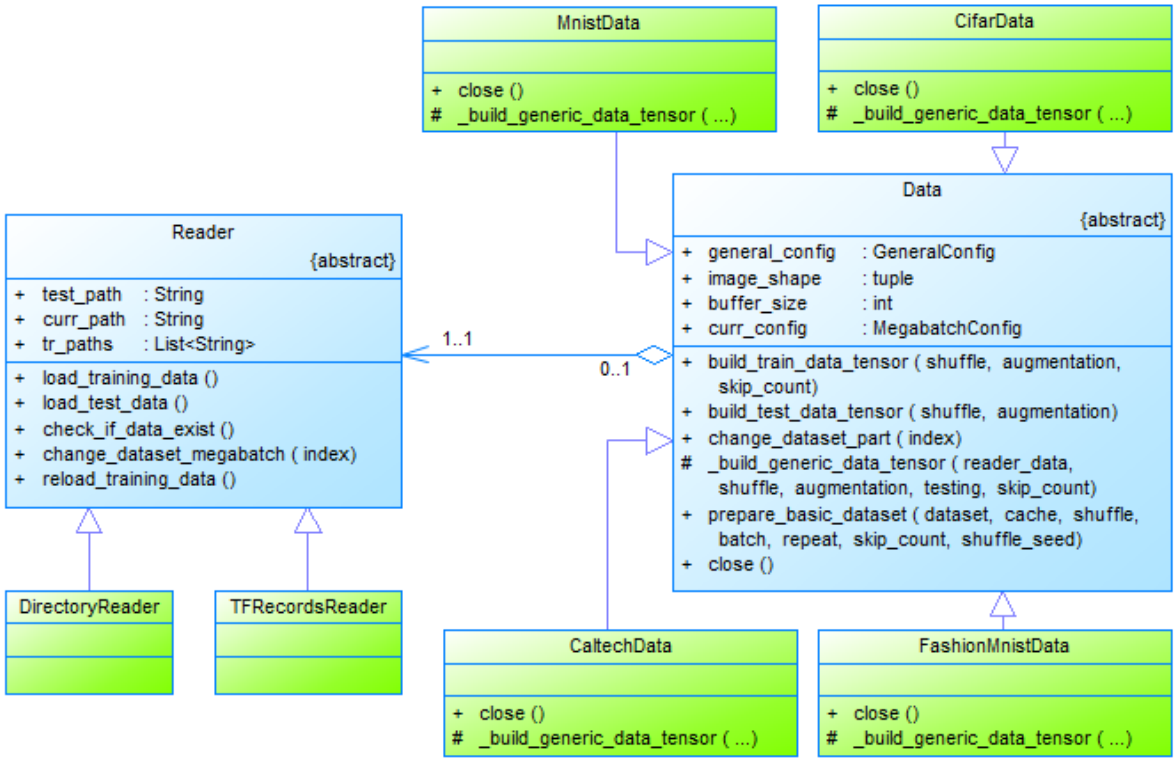

Figura 7 *Diagrama de clases detallado del módulo ETL*

Existen dos clases principales en este módulo: Reader, que se centra en la extracción de los datos en disco; y Data, que se encarga de procesar y cargar los datos en forma de lotes. Se divide de esta manera para facilitar la agregación de múltiples formas de acceso a disco, y para independizar las etapas del proceso ETL. El funcionamiento de estas clases se explica a continuación.

3.1.1.1 Reader

La clase abstracta **Reader** se encarga de listar los archivos desde los cuales serán cargados los datos para realizar un experimento en concreto. Esta clase se encarga de completar la fase de **Extracción**.

También se pueden apreciar dos implementaciones de Reader, que dan soporte a dos formas de cargue de datos en disco. **TFRecordsReader** se encarga de datasets que se encuentran almacenados en el formato TFRecord que define TF. **DirectoryReader** soporta datasets que se encuentren estructurados en árboles de

directorios, donde cada directorio representa una clase y cada imagen que contiene representa un ejemplo de ella.

Es importante resaltar la forma en la que se dividen los datasets. En la **Figura 8** se puede apreciar que un dataset está conformado por uno o más megalotes incrementales, y a su vez cada megalote está conformado por múltiples lotes. Los megalotes se usan para facilitar la ejecución del aprendizaje incremental sobre diferentes conjuntos de datos, y los lotes se usan para dar flexibilidad en la alimentación de datos para el entrenamiento de la red.

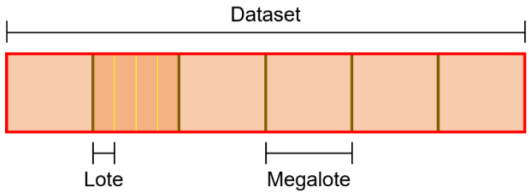

Figura 8 *División de un dataset en lotes y megalotes*

3.1.1.2 Data

Toma la lista de archivos que se construye en **Reader** como entrada para realizar las fases de transformación y cargue mediante las funciones ***build_train_data_tensor*** y ***build_test_data_tensor***, las cuales construyen los tensores para la fase de entrenamiento y para realizar la validación respectivamente. En estas funciones se define cómo deben cargarse los datos y cómo serán transformados si así se requiere, para ello se hace uso de la clase **Dataset** que provee TF la cual actúa como un **Pipeline** de datos.

3.1.2 Módulo de Redes

En la **Figura 9** se muestra el módulo de **Redes**, cuya base es **Network**, esta es una clase que representa a una RNA genérica que puede tener N capas de distintos tipos. Esta clase posee varios métodos destinados a facilitar el desarrollo de redes con diversas arquitecturas, ofreciendo funciones de utilidad que soportan la creación de diversos tipos de capas tales como ReLU, convoluciones y *pooling*.

Esta clase fue adaptada de la librería Caffe-Tensorflow [52] para facilitar la agregación de múltiples arquitecturas de RNA, sin que ello afecte las demás partes del modelo. Para crear una red específica, se crea una subclase de Network en la cual se define por medio de métodos encadenados cuáles van a ser las diferentes capas que contiene la red.

Adicionalmente, Network da soporte al uso de *Transfer Learning*, principalmente por medio de dos métodos: ***maybe_load_model*** para cargar los pesos de una red preentrenada desde un archivo de arreglos Numpy; y ***trainable_variables*** para determinar qué capas de la red serán congeladas y qué capas verán sus pesos modificados durante el entrenamiento.

Las arquitecturas que se implementaron para realizar los distintos experimentos se presentan a continuación.

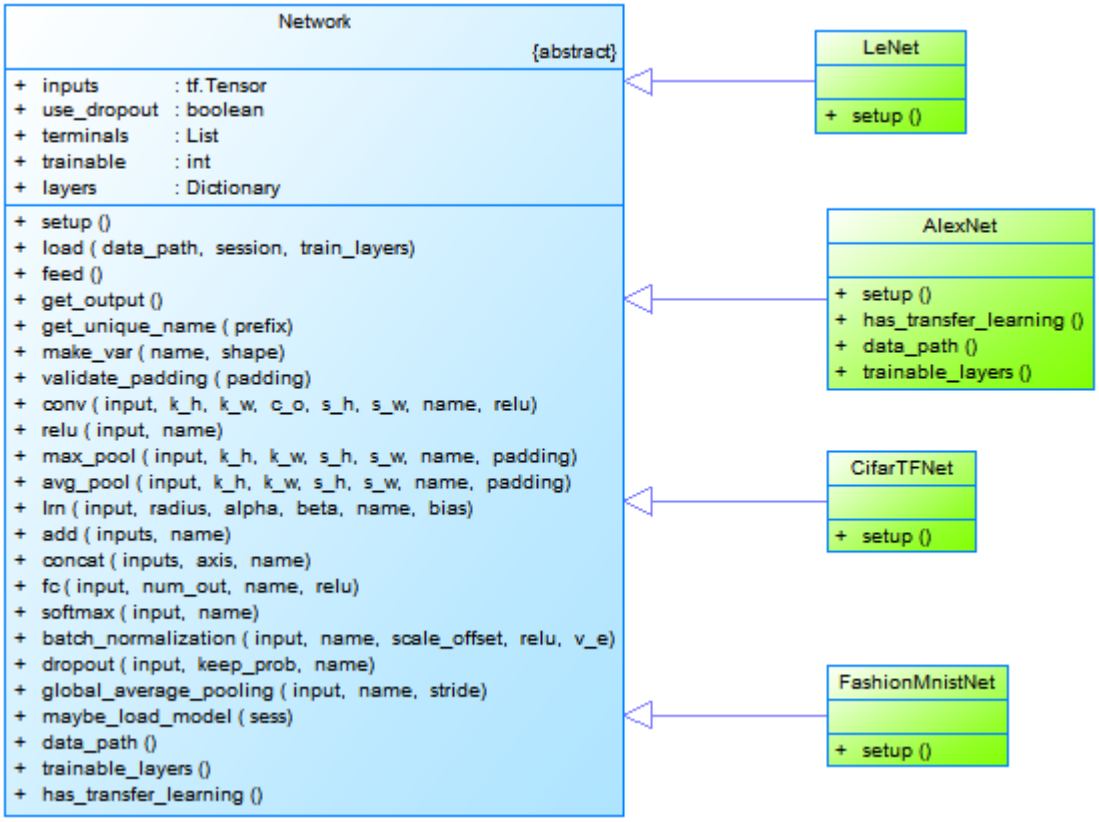

Figura 9 *Diagrama de clases detallado del módulo de Redes*

3.1.2.1 LeNet

Fue la primera arquitectura con capas convolucionales propuesta por LeCun [53] para el reconocimiento de números manuscritos MNIST [54]. Esta arquitectura fue implementada sin realizar modificaciones y se puede observar en la **Figura 10**.

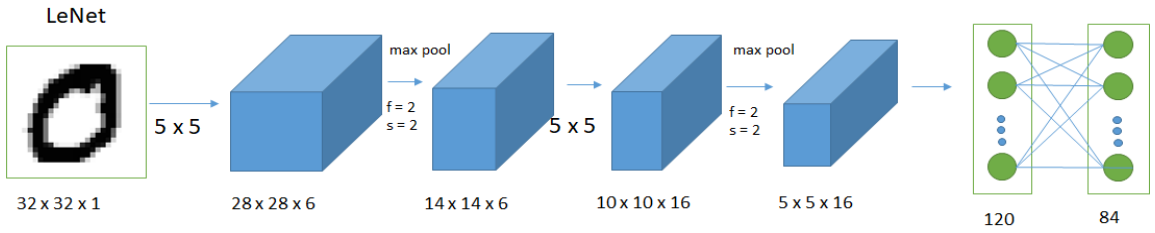

Figura 10 *Arquitectura de red LeNet para MNIST*

3.1.2.2 CifarTFNet

Es una arquitectura propuesta en un tutorial de TF [55] para la clasificación del dataset Cifar-10 [56], y aunque no tiene un nombre conocido, de ahora en adelante se le denomina **CifarTFNet**. Para su implementación, se tomó la arquitectura originalmente propuesta y que se muestra en la **Figura 11**.

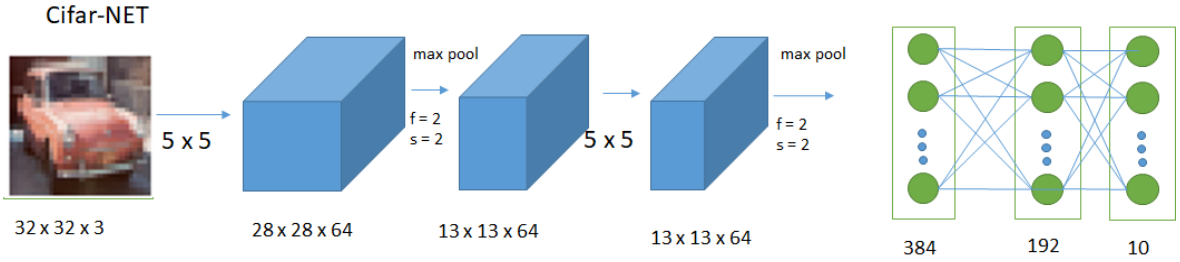

Figura 11 *Arquitectura de red "CifarTFNet" para CIFAR-10*

3.1.2.3 AlexNet

AlexNet es una CNN propuesta por **Alex Krizhevsky** para participar en el desafío de reconocimiento visual ImageNet-2012 consiguiendo un error top-5 de 15.3 % [57]. La arquitectura original se muestra en la **Figura 12**, sin embargo, para su implementación y uso con el dataset Caltech 101 [58] se realizaron algunas modificaciones: se removió la última capa FC y se redujo la cantidad de neuronas de las dos capas FC restantes a 2048 y 1024, respectivamente. También se modificó la salida para que tuviera 101 neuronas, el mismo número de clases que Caltech 101. Adicionalmente, se usó Transfer Learning, copiando los pesos de una red pre entrenada en ImageNet.

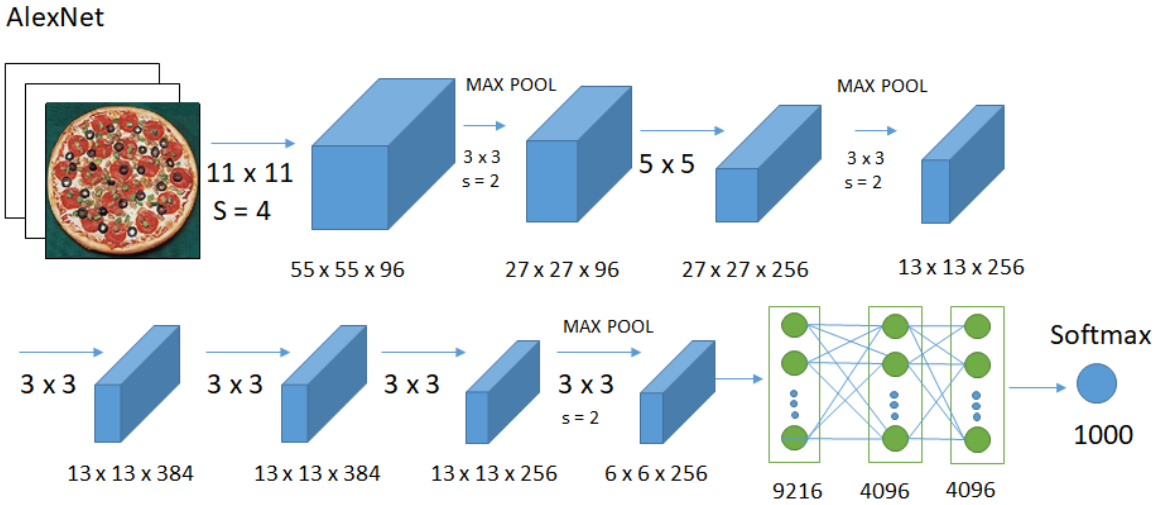

Figura 12 *Arquitectura AlexNet para Caltech 101*

3.1.2.4 FashionMnistNet

Esta arquitectura se usa para la clasificación del dataset Fashion-MNIST [59] en [60], y aunque no cuenta con un nombre conocido, de ahora en adelante se denomina **FashionMnistNet**. En la **Figura 13** se puede observar la red propuesta, la cual está compuesta por dos capas convolucionales junto a operaciones de pooling y dropout. Esta arquitectura se implementó sin realizar cambios.

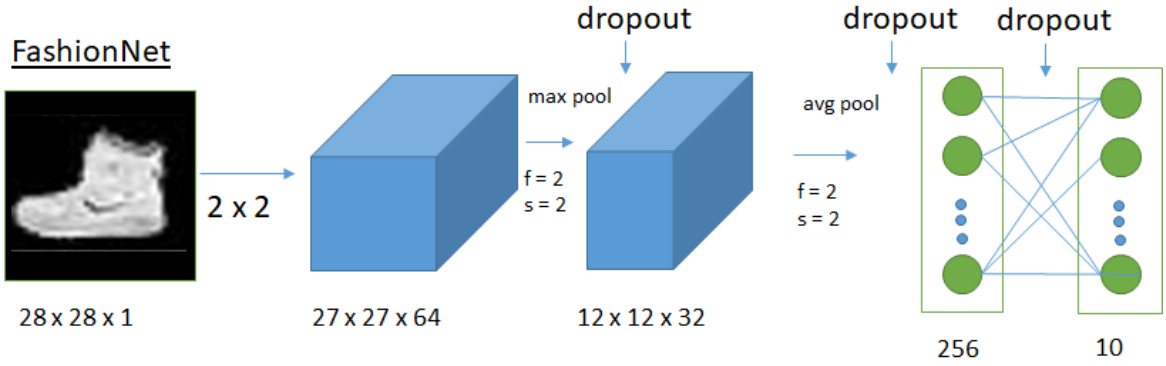

***Figura 13** Arquitectura de red "FashionMnistNet" para Fashion-MNIST*

3.1.3 Módulo de Entrenamiento

En la **Figura 14** se muestran las clases usadas para el entrenamiento y almacenamiento de pesos de las distintas redes utilizadas en las pruebas, y en la **Figura 15** se muestran las clases usadas en los algoritmos propuestos en el Capítulo 4. El funcionamiento de estas clases se explica a continuación.

3.1.3.1 Trainer

La clase **Trainer** es el componente central de este módulo. Se encarga de ejecutar el proceso de entrenamiento según la configuración provista. Para esto se hace uso de tres métodos que trabajan en diferentes niveles: **train** se encarga de preparar el entorno para iniciar el entrenamiento y entrenar la red en un dataset completo; **train_megabatch** realiza el entrenamiento sobre un megalote del dataset que está conformado por múltiples lotes; y finalmente **_train_batch** se encarga de aplicar el optimizador escogido para entrenar la red con un lote de datos. Esta forma de manejar el entrenamiento permite aplicar distintas configuraciones en cada nivel de especificidad, y da soporte directo para la realización de un entrenamiento incremental.

Adicionalmente, Trainer cuenta con las funciones **_create_loss** y **_create_optimizer** que permiten que cada subclase haga uso de distintas medidas de pérdida y optimizadores, algo que es vital para las pruebas realizadas en este trabajo.

Es importante resaltar que esta clase fue pensada para soportar a los múltiples algoritmos de optimización que fueron probados, de forma tal que el algoritmo propuesto en este trabajo fue construido como una subclase de Trainer.

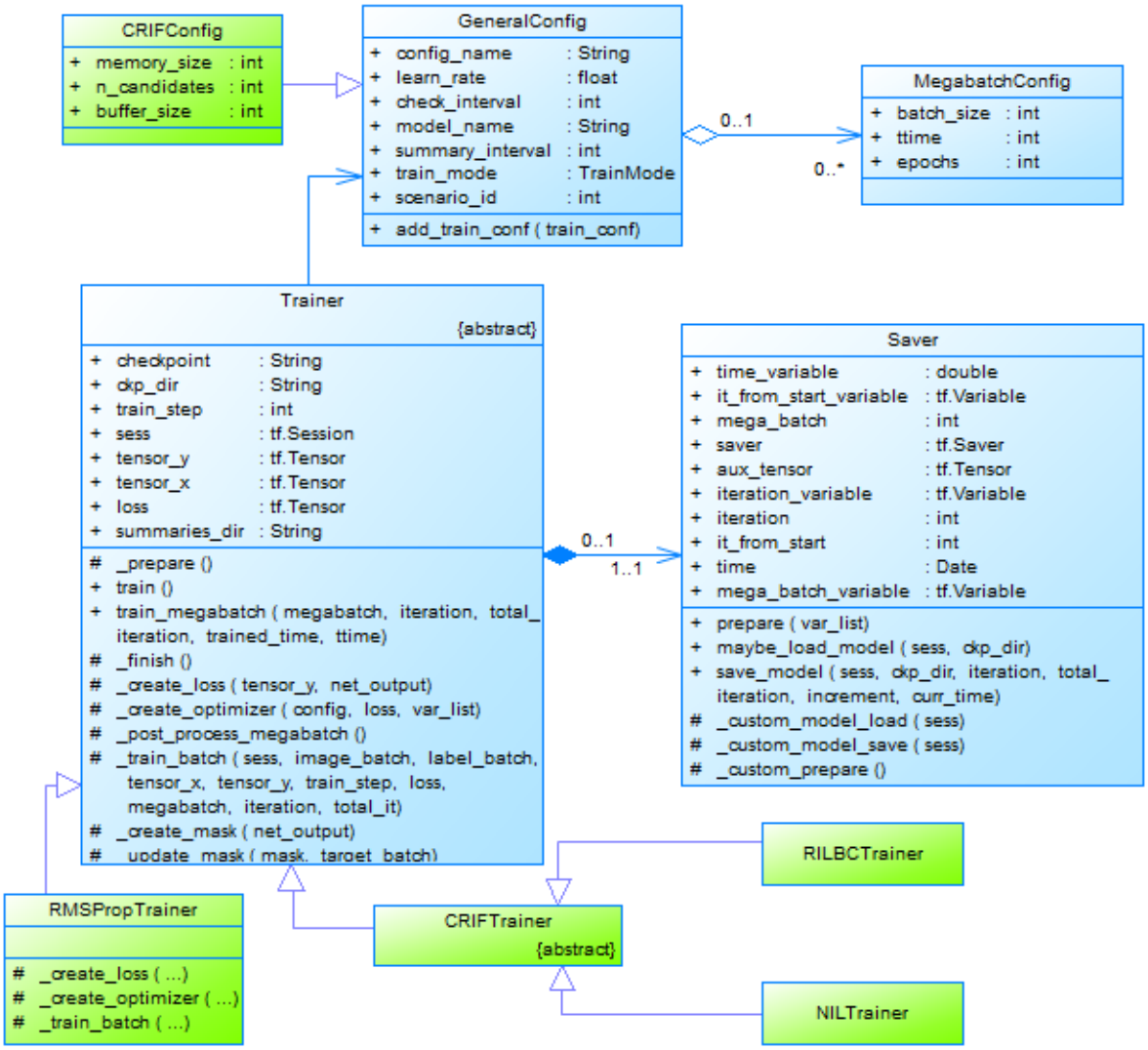

Figura 14 *Diagrama de clases detallado del módulo de Entrenamiento*

3.1.3.2 GeneralConfig

La clase **GeneralConfig** contiene la configuración de los parámetros usados durante un entrenamiento y que no varían entre los diferentes megalotes, y adicionalmente contiene la lista de configuraciones de cada megalote. Los parámetros más importantes son la tasa de aprendizaje, el número de iteraciones en el cual se evaluará el desempeño del modelo sobre los datos de prueba, y el número de iteraciones en el cual se guardarán los pesos del modelo de forma persistente.

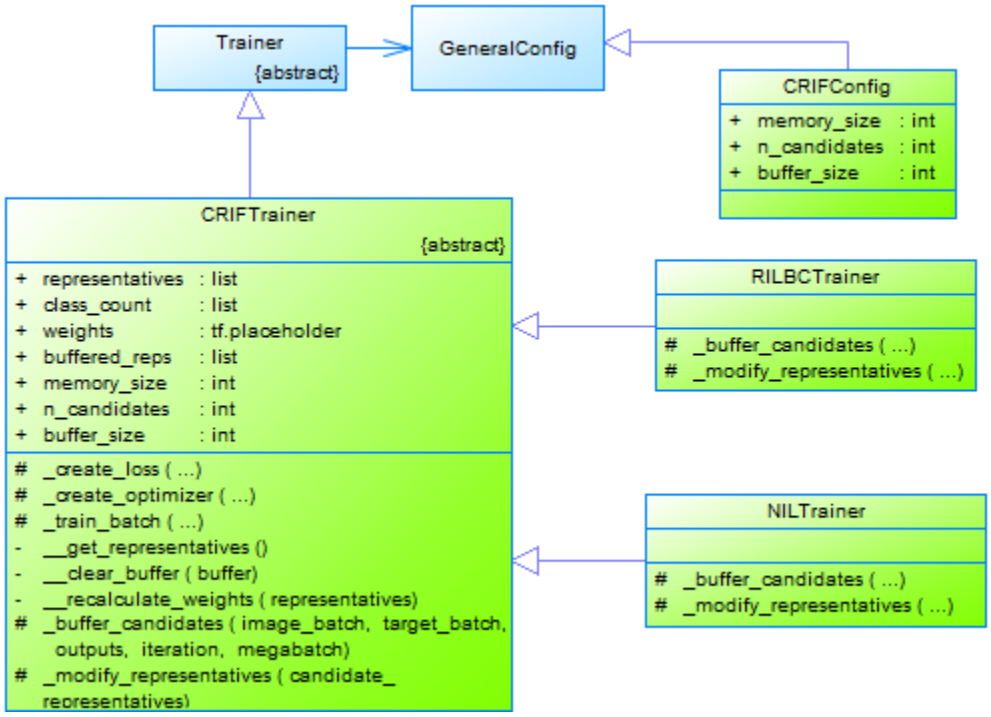

Figura 15 *Diagrama de clases detallado de los algoritmos propuestos CRIF, NIL y RILBC*

3.1.3.3 MegabatchConfig

La clase **MegabatchConfig** contiene la configuración de los parámetros usados durante un entrenamiento para un megalote específico. Estos parámetros son el número de épocas (número de veces que se recorren los datos), el tiempo máximo de entrenamiento y el tamaño de los lotes individuales.

Esta clase se creó con el fin de flexibilizar la creación de diferentes pruebas y configuraciones de entrenamiento para cada megalote.

3.1.3.4 Saver

La clase **Saver** se encarga de guardar los pesos de una RNA en entrenamiento, y de cargar los pesos resultantes de un entrenamiento previo. Para cumplir con estas tareas, se hace uso de los métodos **save** y **restore** de la clase **Saver** provista por TF. Es de resaltar que esta clase únicamente puede cargar modelos guardados en formato *ckpt*, y que los pesos se cargan sobre una red previamente creada que tiene la misma estructura que la red que se desea cargar.

La principal utilidad de esta clase radica en la posibilidad de retomar un entrenamiento que fue interrumpido previamente.

3.1.4 Módulo de Experimentos

En la **Figura 16** se muestran las clases usadas para el registro de resultados de los distintos algoritmos. El funcionamiento de estas clases se explica a continuación.

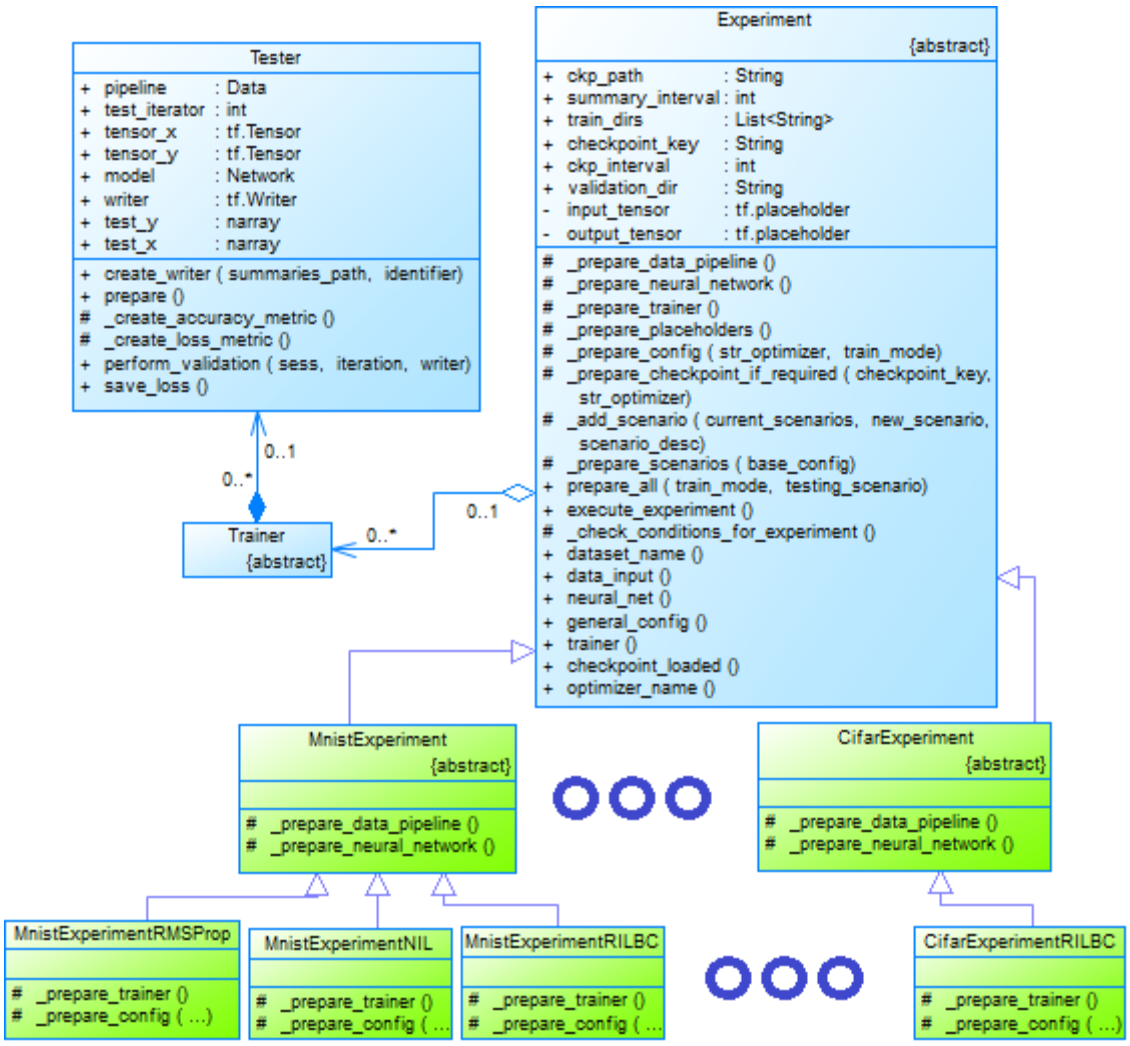

Figura 16 *Diagrama de clases detallado del módulo de Experimentos*

3.1.4.1 Experiment

Experiment agrupa cada una de las partes para realizar un experimento. En ella se definen qué arquitectura de red y qué algoritmo de entrenamiento será usado para la prueba, además de otras configuraciones adicionales como el tamaño del lote o el número de megalotes.

Esta clase fue creada con el fin de que la definición de los valores de configuración de un experimento esté centralizada, para así facilitar su lectura y modificación.

3.1.4.2 Tester

La clase **Tester** se encarga de llevar a cabo la validación del modelo, para lo cual calcula métricas (como la exactitud) que son almacenadas en un archivo de tipo log. Los resultados pueden visualizarse de manera gráfica usando **TensorBoard**.

3.2 Diseño y Ejecución de un Experimento

Para ilustrar el proceso llevado a cabo para integrar un algoritmo de entrenamiento en el framework de pruebas, a continuación, se muestra un ejemplo detallado de la implementación del algoritmo **RMSProp** [8] para realizar un entrenamiento sobre el dataset **MNIST** [54] con **LeNet** [53].

Para esto, se detallan los pasos necesarios para usar cada uno de los módulos descritos en la sección anterior en este ejemplo concreto. La implementación de otros algoritmos y la integración de otros datasets sigue un proceso similar.

3.2.1 Módulo ETL

Para hacer uso de la tubería de entrada y dar soporte al nuevo dataset, es necesario crear una clase que herede de **Data**, y que implemente los siguientes métodos:

- ***_build_generic_data_tensor():*** construye los tensores correspondientes a imágenes y etiquetas.
- ***close():*** cierra cualquier archivo abierto por la tubería.

Adicionalmente, en la implementación se tiene que tener en cuenta el tipo de almacenamiento del dataset para usar el **Reader** apropiado. En el framework base se soportan datasets tanto en formato TFRecord, como distribuidos en directorios cuya estructura sea: *dataset/clase/imagen*.

En este caso se crea la clase **MnistData** como implementación, haciendo uso de **TFRecordsReader** como fuente de datos para la tubería. El constructor de la clase se muestra en la **Figura 17**, y en él se define la configuración general del experimento, los directorios de entrenamiento y el directorio de validación, además de los datos adicionales que se requieren para la creación de los tensores de entrenamiento y validación.

```
class MnistData(Data):

    def __init__(self, general_config,
                 train_dirs: [str],
                 validation_dir: str,
                 buffer_size=60000,
                 image_height=32,
                 image_width=32):
        my_mnist = TFRecordsReader(train_dirs, validation_dir, general_config.train_mode)
        super().__init__(general_config, my_mnist, (image_height, image_width, 1), buffer_size=buffer_size)
        self.data_reader.check_if_data_exists()
```

Figura 17 *Implementación de MnistData*

El siguiente paso es la implementación del método **_build_generic_data_tensor** el cual permite construir los tensores de entrenamiento y evaluación en la súper clase **Data**. En este método se pueden aplicar diferentes operaciones a los datos antes de empezar con el entrenamiento, como la transformación de los datos, *data augmentation*, ordenamiento aleatorio, etc.

En la **Figura 18** se muestra un ejemplo de implementación, donde se realiza una transformación de datos usando la función **map**. Posteriormente se hace uso de la función **prepare_basic_dataset** que incorpora la aplicación de varias transformaciones comunes, como el ordenamiento, uso de caché, repetición por varias épocas y división en lotes. Finalmente se crea un iterador y se obtienen los tensores correspondientes a imágenes y etiquetas.

```
def _build_generic_data_tensor(self, reader_data, shuffle, augmentation, testing, skip_count=0):

    filenames = reader_data[0]
    dataset = tf.data.TFRecordDataset(filenames, num_parallel_reads=len(self.general_config.train_configurations))
    dataset = dataset.map(self.parser, num_parallel_calls=8)
    dataset = self.prepare_basic_dataset(dataset, shuffle=shuffle, cache=True, repeat=testing,
                                         skip_count=skip_count, shuffle_seed=const.SEED)

    iterator = dataset.make_initializable_iterator()
    images_batch, target_batch = iterator.get_next()
    return iterator, images_batch, target_batch
```

Figura 18 *Implementación de _build_generic_data_tensor*

3.2.2 Módulo de Redes

Para crear una red cualquiera, sólo es necesario crear una clase que herede de **Network** que implemente el método setup(), definiendo cada una de las capas, en el orden correcto.

En este caso se creará una red **LeNet** tal y como se puede observar en la **Figura 19**. Se crea cada tipo de capa (convolucional, *pooling* y FC) usando las funciones predefinidas en Network y se conectan encadenando sus funciones. Adicionalmente se provee la entrada de datos con ***feed***.

```
class LeNet(Network):
    def setup(self):
        (self.feed('data')
         .conv(5, 5, 6, 1, 1, padding='VALID', name='conv1')
         .max_pool(2, 2, 2, 2, padding='VALID', name='pool1')
         .conv(5, 5, 20, 1, 1, padding='VALID', name='conv2')
         .max_pool(2, 2, 2, 2, padding='VALID', name='pool2')
         .fc(120, name='fc1')
         .fc(84, name='fc2')
         .fc(10, relu=False, name='fc3'))
```

Figura 19 *Implementación de LeNet*

3.2.3 Módulo de Entrenamiento

Para implementar el algoritmo RMSProp es necesario crear una clase que herede de **Trainer**, y que implemente los siguientes métodos:

- **_create_loss():** donde se define la medida de pérdida que se va a usar entre las medidas soportadas por TF (e.g. **Softmax Cross Entropy**).
- **_create_optimizer():** se define el optimizador que será usado.
- **_train_batch():** recibe las muestras de un lote y aplica el optimizador.

La clase creada se puede observar en la **Figura 20**. La medida de pérdida usada es *Softmax Entropy*, y se debe proveer la salida de la red y las etiquetas reales de las muestras para que sea calculada. Adicionalmente se hace uso de *mask_tensor*, que guarda las clases que se han visto hasta el momento para permitir un entrenamiento con clases incrementales.

El optimizador provisto es *RmsProp*, y únicamente se debe pasar la tasa de aprendizaje y la medida de pérdida, así como la lista de variables a optimizar. Es este componente el que modifica los pesos de la red durante el entrenamiento.

Finalmente, para entrenar la red con un lote de datos se corren los operadores creados previamente en una sesión de TF, y se proveen los datos.

```
class RMSPropTrainer(Trainer):

    def _train_batch(self, sess, image_batch, target_batch, tensor_x: tf.Tensor, tensor_y: tf.Tensor,
                     train_step: tf.Operation, loss: tf.Tensor, megabatch: int, iteration: int, total_it: int):
        return sess.run([train_step, loss],
                        feed_dict={tensor_x: image_batch, tensor_y: target_batch,
                                   self.mask_tensor: self.mask_value})

    def _create_loss(self, tensor_y: tf.Tensor, net_output: tf.Tensor):
        return tf.losses.softmax_cross_entropy(tf.multiply(tensor_y, self.mask_tensor),
                                               tf.multiply(net_output, self.mask_tensor))

    def _create_optimizer(self, config: GeneralConfig, loss: tf.Tensor, var_list=None):
        return tf.train.RMSPropOptimizer(config.learn_rate).minimize(loss, var_list=var_list)
```

Figura 20 *Implementación de RMSPropTrainer*

3.2.4 Módulo de Experimentación

El experimento debe heredar de la clase **Experiment**, e implementar los siguientes métodos:

- _prepare_data_pipeline(): crea la tubería que suministra los datos. Para este caso se crea una tubería de tipo **MnistData**.
- _prepare_neural_network(): crea el modelo que será entrenado. Para este caso se crea una red de tipo **LeNet**.
- _prepare_trainer(): crea el objeto encargado del entrenamiento. Para este caso se crea un **RMSPropTrainer**.
- _prepare_config(): se crean las configuraciones específicas para el entrenamiento y pruebas. Se debe instanciar un único objeto de

configuración global (**GeneralConfig**) y tantos objetos de configuración local como megalotes se tengan en el dataset (**MegabatchConfig**).

También es factible organizar estos métodos en una jerarquía de clases para definir elementos comunes a múltiples experimentos. Éste fue el enfoque usado para diseñar el experimento actual: se crea la tubería y la red neuronal en una subclase llamada **MnistExperiment**, y a su vez se crea una subclase de ésta denominada **MnistExperimentRMSProp** que define al entrenador y la configuración de la prueba. Estas clases se pueden observar en la **Figura 21** y la **Figura 22** respectivamente.

```
class MnistExperiment(Experiment, ABC):

    dataset_name = const.DATA_MNIST
    data_input = None
    neural_net = None

    def _prepare_data_pipeline(self):
        self.data_input = MnistData(self.general_config, self.train_dirs, self.validation_dir)

    def _prepare_neural_network(self):
        self.neural_net = LeNet({'data': self.input_tensor})
```

***Figura 21** Implementación de MnistExperiment*

```
class MnistExperimentRMSProp(MnistExperiment):

    optimizer_name = const.TR_BASE
    general_config = None
    trainer = None

    def _prepare_trainer(self):
        tester = Tester(self.neural_net, self.data_input, self.input_tensor, self.output_tensor)
        self.trainer = RMSPropTrainer(self.general_config, self.neural_net, self.data_input,
                                      self.input_tensor, self.output_tensor,
                                      tester=tester, checkpoint=self.ckp_path)

    def _prepare_config(self, str_optimizer: str, train_mode: TrainMode):
        self.general_config = GeneralConfig(train_mode, 0.0001, self.summary_interval, self.ckp_interval,
                                            config_name=str_optimizer, model_name=self.dataset_name)
        for i in range(5):
            train_conf = MegabatchConfig(epochs=20, batch_size=128)
            self.general_config.add_train_conf(train_conf)
```

***Figura 22** Implementación de MnistExperimentRMSProp*

Cabe resaltar que los componentes creados en los otros módulos son independientes uno de otro, de modo que se puedan usar en otros experimentos sin realizar ningún cambio.

3.2.5 Ejecución del Experimento

Para ejecutar un experimento es necesario primero tener una carpeta con los datos que serán usados para el entrenamiento (i.e. el dataset). Luego, sólo es

necesario hacer uso del constructor de la clase, y los métodos *prepare_all* y *execute_experiment.* Esto puede observarse en la **Figura 23**.

```
exp = MnistExperimentRMSProp(train_dirs, validation_dir,
                             s_interval, ckp_interval, checkpoint_key)
exp.prepare_all(train_mode=TrainMode.INCREMENTAL)
exp.execute_experiment()
```

Figura 23 *Ejecución de un Experimento*

Mientras se ejecuta el experimento, los resultados del entrenamiento se van guardando en tiempo real en la carpeta *summaries* y se pueden ver al acceder a ellos con TensorBoard. La **Figura 24** muestra un ejemplo.

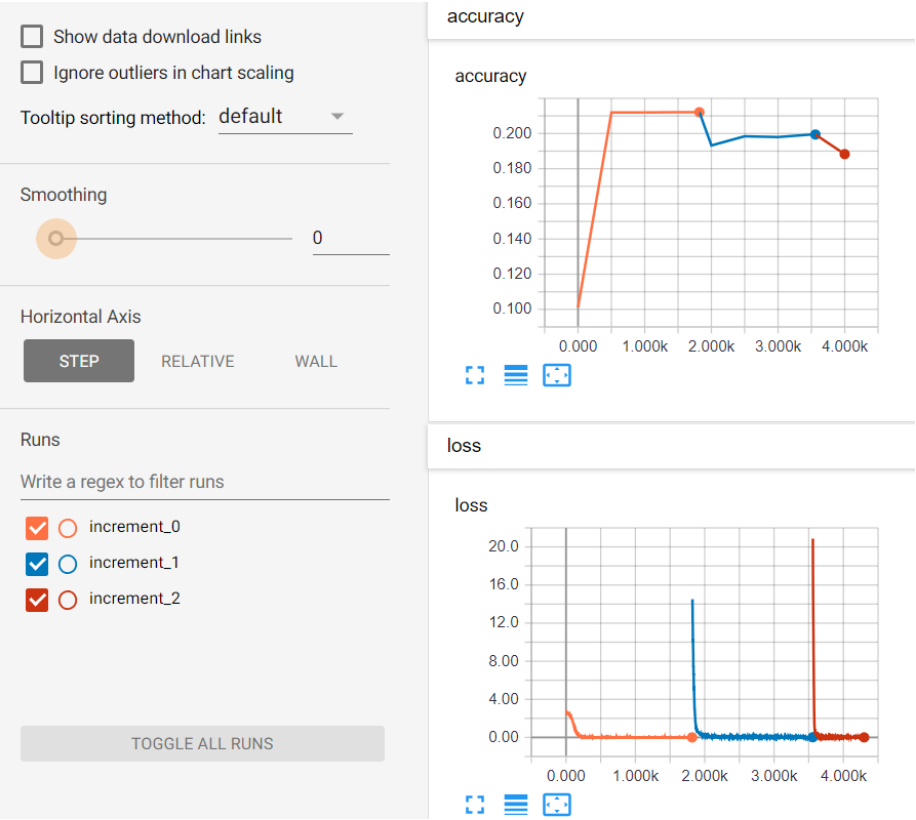

Figura 24 *Ejemplo de resultados en TensorBoard*

CAPÍTULO 4

4 PROPUESTA

4.1 METODOLOGÍA DE DESARROLLO

En este trabajo se usó el patrón de investigación iterativo (PII) propuesto por Pratt [61], para el desarrollo de los algoritmos propuestos en este capítulo. PII se compone de cuatro etapas principales, a saber: observación de campo (O), identificación del problema (I), desarrollo de la solución (D) y pruebas de campo (P). Los algoritmos se desarrollaron y refinaron a lo largo de 3 ciclos compuestos de estas etapas.

En todos los ciclos, en la etapa de observación, se realizó una búsqueda de mecanismos de selección y competencia de representantes presentes en otras soluciones de aprendizaje incremental. Adicionalmente, en los ciclos iniciales se identificó la necesidad de agregar mecanismos complementarios para el control del entrenamiento, lo que llevó a la búsqueda de técnicas de regularización para asignar un peso de penalización a las muestras durante el entrenamiento.

Posteriormente, en la etapa de identificación se escogieron las estrategias que se consideraban más prometedoras para cada componente. Por otra parte, las etapas de desarrollo y pruebas (experimentación) se realizaron en paralelo, en un proceso de retroalimentación constante. Este proceso funcionó de la siguiente manera: se implementaron versiones básicas de las técnicas seleccionadas, que eran inmediatamente probadas en experimentos rápidos en ambientes pequeños, cuyos resultados fueron usados para guiar el desarrollo de pequeñas variaciones sobre las técnicas. Estas variaciones incluyeron aspectos como: variación en valores de los parámetros, inclusión de factores adicionales en las ecuaciones, cambios en el orden de ejecución de los pasos, fusión de componentes de múltiples algoritmos, entre otros.

Por otro lado, se resalta que en el último ciclo de este proceso se inició la escritura de artículos para divulgación científica. Durante esta etapa se revisaron detalles relacionados con los pseudocódigos de los algoritmos y, se realizó un proceso retrospectivo para identificar todos los aspectos necesarios para lograr una adecuada reproducibilidad de los resultados de las pruebas finales presentadas en el Capítulo 5.

4.2 MARCO CONCEPTUAL PARA EL APRENDIZAJE INCREMENTAL BASADO EN REHEARSAL

El marco conceptual de aprendizaje incremental propuesto en esta investigación se basa en la estrategia de aprendizaje conocida como *Rehearsal*, que en el proceso de enseñanza-aprendizaje con niños consiste en la repetición literal de las palabras exactas que se desea que los alumnos recuerden, en forma oral, verbal o escrita usando la repetición simple, la repetición acumulativa, la toma de notas y el marcado o resaltado de textos [62]. En el contexto de las redes neuronales, consiste en almacenar muestras de entrenamientos previos para apoyar entrenamientos posteriores, combinando parte de los datos previos con nuevos datos de entrenamiento para entrenar la red, logrando con esto fortalecer las conexiones de lo ya aprendido por la red (estabilidad) e incorporando nuevos aprendizajes a partir de los nuevos datos (plasticidad).

Una estrategia sencilla de *Rehearsal* es el *Full Rehearsal*, donde se hace uso de todas las muestras vistas hasta el momento [63]. Este método es ineficiente en términos de tiempo de entrenamiento y almacenamiento de datos (no es incremental sino acumulativo), por lo cual en algunos escenarios puede resultar inviable. El objetivo de esta investigación es generar un algoritmo que se acerque al rendimiento obtenido por *Full Rehearsal*, medido en calidad de la clasificación, disminuyendo el tiempo de entrenamiento y la cantidad de memoria requerida para el almacenamiento de las muestras.

Para lograr lo anterior, se propone un *Competitive Representatives Incremental Framework* (CRIF) que permite el entrenamiento incremental de redes neuronales profundas en problemas de clasificación usando muestras representativas. Cuando se inicia el entrenamiento con el primer megalote del datataset, se realiza el entrenamiento de la red con un optimizador externo (algoritmo de entrenamiento) usando una medida de pérdida (*loss*) previamente definida (paso 1 de la **Figura 25**). El algoritmo modifica los pesos de la red y al terminar el lote se inicia el proceso de selección de candidatos por parte de CRIF, en el cual se toman unos datos del lote como candidatos a ser representantes, basado en alguna medida de incertidumbre o representatividad (paso 2). Luego estos candidatos se usan para definir los primeros representantes de las diferentes clases del dataset (paso 3). En este caso el paso de “Actualización de Representantes” no realiza ninguna tarea específica.

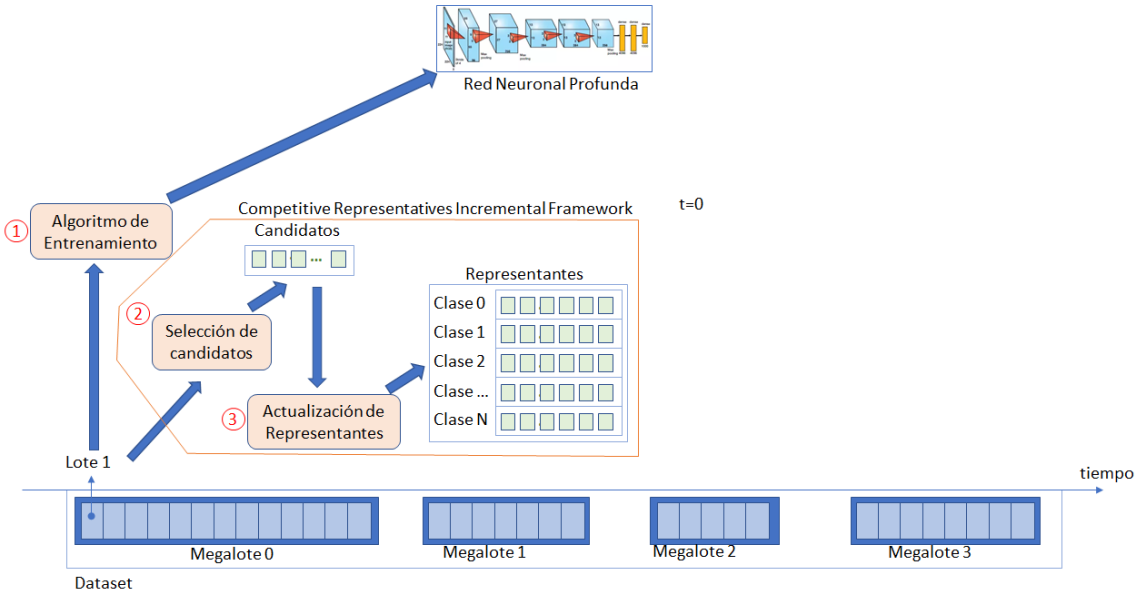

Figura 25 *CRIF y su interacción con otros componentes del proceso de entrenamiento (datos y algoritmo de entrenamiento) en su primer uso*

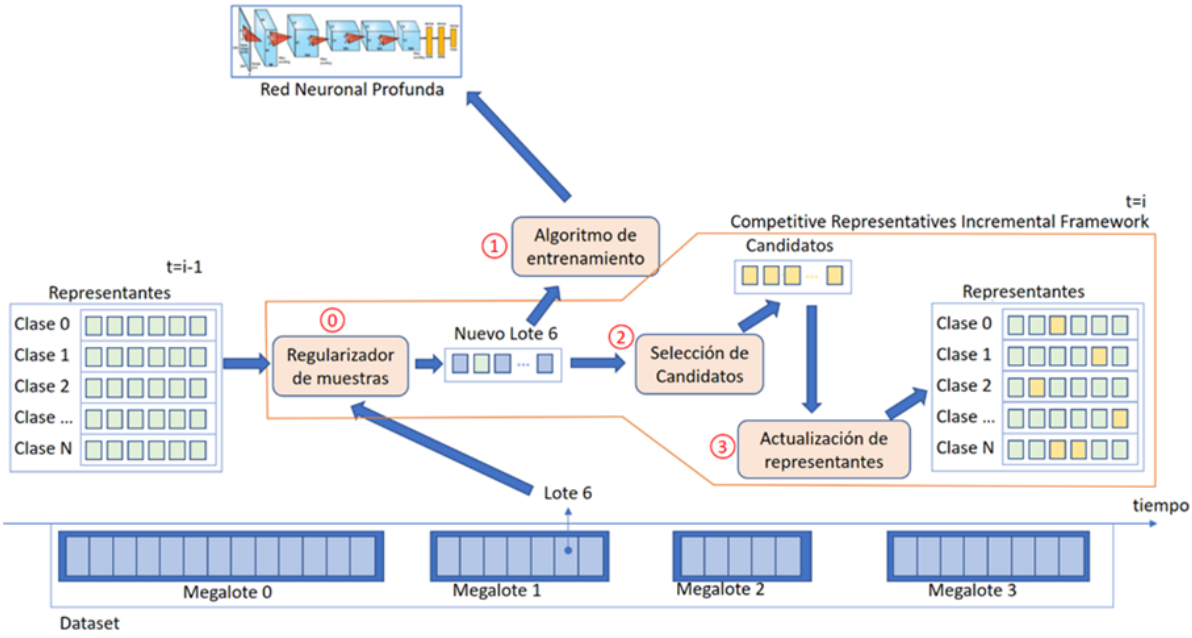

Figura 26 *CRIF y su interacción con otros componentes del proceso de entrenamiento (datos y algoritmo de entrenamiento) en pasos posteriores*

Desde el siguiente lote del dataset se realizan los pasos presentados en la **Figura 26**. En esta figura se introduce un paso cero (0) relacionado con la regularización de muestras, proceso que se encarga de seleccionar muestras de los representantes previamente guardados (en iteraciones anteriores) y generar un nuevo lote que incluye los datos del lote actual y los representantes seleccionados con un peso diferente al de los datos nuevos para ser enviados al algoritmo de

entrenamiento. Cuando el algoritmo de entrenamiento termina, se realiza el mismo paso de “Selección de Candidatos” pero en este caso se toma como entrada el nuevo lote sin los representantes escogidos por el Regularizador de Muestras (paso 2). El paso 3 se encarga de hacer la actualización de los representantes, que en este caso toma los candidatos seleccionados y establece un mecanismo para remplazar algunos de los representantes de las clases, son estos representantes los que se utilizarán luego como base del paso cero en la siguiente iteración. CRIF sólo requiere como costo adicional en almacenamiento, guardar la lista de los datos que forman los representantes, y no modifica el interior de los algoritmos de entrenamiento, lo que es una ventaja para su uso con diferentes arquitecturas DNN.

Además de la lista de representantes, los principales componentes de CRIF son: (1) Selección de Candidatos, (2) Actualización de Representantes y (3) Regularizador de Muestras. Estos componentes están diseñados como cajas negras que pueden ser extendidos y cambiados por otras propuestas, dándole flexibilidad al framework. Más adelante en este mismo capítulo se presentan dos propuestas que definen de forma diferente estos componentes.

4.2.1 Selección de Candidatos

Este componente se encarga de preseleccionar un subconjunto de muestras candidatas a partir de los datos del lote actual, tomando cada muestra y evaluándolo según una métrica establecida. Se define un número fijo de candidatos *c* para cada lote, de modo que las *c* muestras con mejor puntaje en la métrica se guardan en un vector de candidatos*,* que posteriormente compite con los representantes ya existentes para producir un conjunto de representantes actualizado. Esta competencia se realiza únicamente cada *q* iteraciones, siendo *q* un parámetro configurable. De este modo, los candidatos se guardan en un vector buffer cuyo tamaño máximo es *c x q*, hasta tanto no compitan con los representantes (Actualización de Representantes). El buffer se libera cada vez que se realiza dicha competencia, para volverse a llenar con un nuevo grupo de candidatos. Un ejemplo de este proceso para c=3 y q=3 se muestra en la **Figura 27**.

El objetivo de realizar una preselección de candidatos en cada lote de datos es lograr una actualización permanente de los representantes. Esto es útil e incluso necesario, en escenarios como el aprendizaje *online*, donde las muestras aparecen una única vez. Sin embargo, aún en casos menos extremos, es de mucha utilidad para poder incorporar de manera inmediata nuevas muestras al mecanismo de *rehearsal,* para así reforzar el entrenamiento actual.

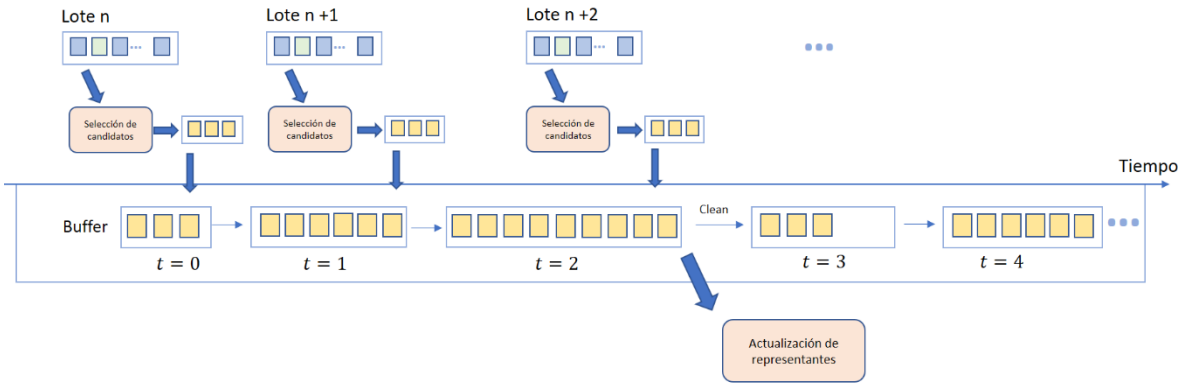

Figura 27 *Estado del Buffer para c=3 y q=3*

4.2.2 Actualización de Representantes

Este componente se encarga de guardar y actualizar el conjunto de representantes del algoritmo. Para ello, define una matriz de representantes de *N x r*, donde N es el número de clases y *r* es el número máximo de representantes por clase; es decir, en cada fila se almacenan los representantes etiquetados con una misma clase. La división de representantes por clase permite asegurar la existencia de muestras correspondientes a todo el espectro de clases vistas, i.e. siempre permanecen muestras para recordar a todas las clases. Esto es relevante si se tiene en cuenta que la proporción de las clases puede variar en el tiempo, e incluso, algunas clases pueden no volver a aparecer durante el entrenamiento de futuros megalotes (como en entornos con clases completamente incrementales).

Para actualizar la matriz de representantes el componente "Actualización de Representantes" toma el *vector* de candidatos obtenido con el componente "Selección de Candidatos" junto a la matriz de representantes y realiza una competencia para escoger el conjunto de representantes que mejor modele los datos, manteniendo los límites de memoria previamente establecidos (valor *r*). Dado que los representantes se encuentran divididos según sus etiquetas de clase, la competencia entre candidatos y representantes también se realiza por clases. Esto significa que, los candidatos de una clase determinada realizan una competencia contra los representantes existentes de la misma clase de acuerdo con la estrategia elegida. En la **Figura 26** (paso 3) se muestra gráficamente este proceso. La actualización de representantes se puede realizar en cada iteración (por cada lote procesado) o cada cierto número de iteraciones, según el tamaño *q* que se defina para el buffer de candidatos.

4.2.3 Regularizador de Muestras

El último componente de CRIF se encarga del proceso de *Rehearsal,* para lo cual escoge un subconjunto de muestras de la matriz de representantes cada vez que llega un nuevo lote de datos para el entrenamiento. Para esto, se crea un nuevo

lote de datos compuesto por los nuevos datos y un número *c* representantes, y es este lote el que se usa para entrenar la red. Según esto, el nuevo lote de datos se define conforme se muestra en la Ecuación (4).

$$lote = lote_{nuevos\_datos}\ U\ lote_{reps} \quad (4)$$

La estrategia para obtener $lote_{reps}$ es definida e implementada por una instanciación de CRIF, como **NIL** o **RILBC**, los cuáles se explican más adelante.

Por otro lado, y teniendo en cuenta que en un entrenamiento incremental es posible que aparezcan nuevas clases no presentes inicialmente, CRIF se adapta a esta situación al preestablecer previamente una arquitectura de red cuyo número de salidas corresponde al número máximo de clases esperadas. Adicionalmente, se introduce el uso de una máscara que cubre el resultado de la salida de la red. Esta máscara consiste en un vector $\overrightarrow{\boldsymbol{M}} = (m_1, m_2, \dots, m_n), m \in \{0,1\}$ donde un 0 corresponde a una clase no observada, y un 1 corresponde a una clase ya observada, y ***n*** es el número de salidas de la red. Esta máscara se introduce en la medida de *loss* conforme se muestra en la Ecuación (5).

$$loss_{máscara}(\vec{y}_{True}, \vec{y}_{Red}) = loss(\vec{y}_{True}, \vec{y}_{Red} \cdot \overrightarrow{M}) \quad (5)$$

Donde $\vec{y}_{True}$ corresponde al vector con las ground-truth labels, y $\vec{y}_{Red}$ es un vector con la salida de la red. Al aplicar la Ecuación (5) se calcula el *loss* de una porción de una red con una arquitectura cuyo número de salidas sea mayor que el número de clases visto hasta el momento, y se ignora la salida de las neuronas asignadas a clases que no han sido observadas.

Finalmente, dentro de este mismo componente se realiza una regularización de las muestras, cuyo objetivo es asignar un peso de penalización específico a cada muestra, de modo que la incorrecta clasificación de ciertas muestras tenga mayor impacto en el cálculo de la medida de *loss* que la de otras muestras. En este caso, se busca incrementar la participación en el entrenamiento de las muestras representantes de cada clase respecto a las nuevas muestras. Para aplicar la regularización se usa un vector $\overrightarrow{\boldsymbol{W}} = (w_1, w_2, \dots, w_n), w_i \in \mathbb{R}^+$, con lo cual la ecuación de *loss* queda según la Ecuación (6). Al integrar la Ecuación (5) junto a la Ecuación (6), se obtiene que el *loss* se rige por la Ecuación (7).

$$loss_{CRIF}(\vec{y}_{True}, \vec{y}_{Red}) = loss_{máscara}(\vec{y}_{True} \cdot \overrightarrow{W}, \vec{y}_{Red} \cdot \overrightarrow{W}) \quad (6)$$

$$loss_{CRIF}(\vec{y}_{True}, \vec{y}_{Red}) = loss(\vec{y}_{True} \cdot \overrightarrow{W}, \vec{y}_{Red} \cdot \overrightarrow{M} \cdot \overrightarrow{W}) \quad (7)$$

Un resumen con las variables más importantes usadas en los componentes de CRIF y que se usan en los algoritmos que se presentan en las secciones posteriores se muestran en la **Tabla 5**.

Tabla 5 *Lista de variables relevantes en los algoritmos*

Variable	Significado
R	Lista de Representantes agrupados por clase
O	Arreglo con la salida de la RNA ($\vec{y}_{Red}$)
B	Lote de nuevos datos (incluyendo datos y etiquetas)
Br	Lote de muestras representativas (incluyendo datos y etiquetas)
H	Vector de conteo con el número histórico de candidatos por clase
N	Número de clases
r	Número de representantes por clase
q	Tamaño del buffer de candidatos, medido en número de iteraciones
c	Número de candidatos que son seleccionados en cada lote, y número de representantes que son usados para entrenamiento en cada lote

4.3 NAIVE INCREMENTAL LEARNING (NIL)

Inspirándose en el hecho de que introducir aleatoriedad en algunos componentes del entrenamiento de una ANN ha probado ser efectivo [64], se propone una implementación sencilla de la propuesta *Rehearsal* denominada *Naive Incremental Learning* (**NIL**), la cual define el componente de "Selección de Candidatos" como una selección aleatoria de los candidatos a representantes. La gran ventaja de **NIL** reside en que su complejidad computacional es bastante baja, de modo que agrega poca sobrecarga al algoritmo de entrenamiento base, pero incorpora las ventajas del *Rehearsal*.

De igual manera en el componente de "Actualización de Representantes" se hace una selección aleatoria entre los representantes y nuevos candidatos de cada clase. Lo anterior implica que cada muestra de una clase determinada tiene la misma probabilidad de ser seleccionada sin importar si es una muestra reciente o antigua, aunque la posibilidad de que una muestra se mantenga durante múltiples iteraciones sigue un decrecimiento geométrico, como se muestra en la Ecuación (8).

$$P_m(i) = (r/(r + q * \frac{c}{N}))^{\lfloor i/q \rfloor} \quad (8)$$

Donde *r* es el número de prototipos por clase, *i* es la iteración actual de entrenamiento, *q* es el tamaño del buffer de candidatos en número de iteraciones y c/N es el promedio de candidatos de cada clase que se seleccionan en cada iteración. De este modo $q * c/N$ es el promedio de candidatos por clase en cada competencia, teniendo en cuenta que el proceso de reemplazo sólo se realiza cuando $q \ mod \ i = 0$. Esta ecuación es aplicable para una distribución uniforme de los datos (i.e. clases balanceadas). La **Figura 28** muestra la curva de P_m para unos valores de ejemplo.

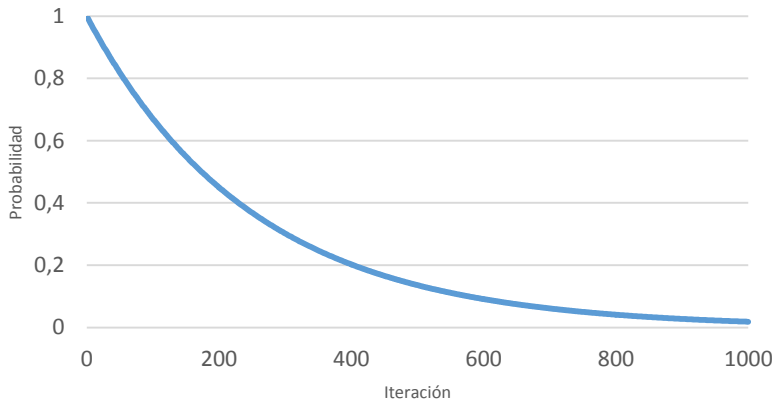

Figura 28 *Probabilidad estimada del tiempo de vida de un representante para c=20, r=500, N=10, y q=1. La gráfica muestra cuán probable es que una muestra que ha sido seleccionada como representante lo siga siendo después de x iteraciones con un método de selección aleatoria*

El componente "Regularizador de Muestras" se muestra en el **Algoritmo 1**. Se usa una selección aleatoria para escoger a los representantes que serán usados en un entrenamiento, ya que dada su sencillez y a que, por su propia naturaleza, tenderá a obtener una gran variedad de clases en las muestras seleccionadas. Esta selección no se realiza por clase, sino entre todos los representantes (línea 2). Por otro lado, los pesos de las muestras en el entrenamiento se asignan con la siguiente estrategia: a las muestras nuevas se les asigna un peso de 1.0 (línea 3), mientras que a los representantes se les asigna un peso según el **Algoritmo 2** (línea 5). En este, se crea un peso proporcional a dos factores: la relación entre el número de representantes de una clase respecto al número total de representantes, y la relación entre el número de candidatos y el número total de muestras observadas hasta el momento (línea 4). Esta relación es suavizada por medio de un Logaritmo natural, asegurando que su valor no sea menor a 1.0 (líneas 6-9). Es importante resaltar que para poder realizar estos cálculos se necesita mantener un histórico del número de candidatos (H), el cuál es actualizado en "Selección de Candidatos" y su estructura es $\vec{\boldsymbol{H}} = (h_1, h_2, \dots, h_n), h_i \in \mathbb{R}^+$, donde h_i corresponde al número total de muestras de la clase ***i*** que han ingresado al *buffer* de candidatos desde el inicio del entrenamiento.

Algoritmo 1 Regularizador de muestras para NIL	
Entradas:	B: lote de datos de entrada, R: matriz de representantes actuales, H: vector de conteo con el número histórico de candidatos por clase, c: número de representantes a ser seleccionados para el entrenamiento en cada lote
Salidas:	un vector con el lote de datos actualizado para entrenamiento, Un vector con el peso de cada muestra
1.	**Inicio**
2.	Crear Br como los c representantes seleccionados aleatoriamente de R

3.	Inicializar W con los pesos para el lote de datos actual $W \leftarrow \vec{1}$, con longitud igual a la longitud de B
4.	B ← B U Br
5.	Wr ← Asignación_Pesos_Reps(Br, R, H, c, |B|)
6.	W ← W U Wr
7.	**Retornar** B, W
8.	**Fin**

Algoritmo 2 Asignación de pesos de representantes para NIL	
Entradas:	B: lote de datos de entrada, R: matriz de representantes actuales, H: vector de conteo con el número histórico de candidatos por clase, c: número de representantes a ser seleccionados para el entrenamiento en cada lote l: tamaño del lote de los no-representantes
Salidas:	un vector con los pesos de cada muestra representante
1.	**Inicio**
2.	Crear T como el número total de candidatos $T \leftarrow \sum_{i=1}^{k} h_i \; ; \; donde \; \vec{\boldsymbol{H}} = (h_1, h_2, \dots, h_n), h_i \in \mathbb{R}^+$
3.	Crear P como la relación entre el número de candidatos de cada clase y el número total de candidatos $P \leftarrow \left(\frac{h_1}{T}, \frac{h_2}{T}, \dots, \frac{h_k}{T}\right), h_i \in H$
4.	Crear wt como la relación entre el número de nuevas muestras y representantes en el lote, combinada con la relación entre el número total de candidatos y la cantidad actual de representantes $wt \leftarrow \left(\frac{l}{c}\right) * \left(\frac{T}{\|R\|}\right)$
5.	Crear W como un vector vacío
6.	**Para** x,y en Br **Hacer**
7.	Calcular el peso de la muestra (***x,y***) donde ***x*** corresponde a los datos de entrada de la muestra, siendo ***y*** es la etiqueta de clase asociada $ws \leftarrow \max(\ln(P[y] * wt), 1.0)$
8.	$W \leftarrow W \; U \; ws$
9.	**Fin Para**
10.	**Retornar** W
11.	**Fin**

4.4 REPRESENTATIVES INCREMENTAL LEARNING WITH BVSB AND CROWDING DISTANCE (RILBC)

Representatives Incremental Learning with BvSB and Crowding Distance (**RILBC**) se presenta como una segunda alternativa de implementación de CRIF. El componente de "Selección de Candidatos" de **RILBC** se basa en la métrica BvSB, la cual hace uso de los resultados de la red para cada uno de los datos del lote. Esta métrica se encarga de medir la incertidumbre que posee la red sobre los datos y permite saber si la muestra se encuentra cerca de una frontera de separación entre las clases (alta incertidumbre) o por el contrario se encuentra alejado de ellas (baja incertidumbre).

BvSB es una métrica sencilla que ya ha sido efectivamente usada en trabajos previos [27][28][65]. Como se define en [27], BvSB es una medida que únicamente considera la diferencia de probabilidad de los valores de las dos clases con mayor probabilidad estimada. La fórmula de BvSB se presenta en la Ecuación (9), donde $p(y_{Best}|x_i)$ es la probabilidad de clase más alta, y $p(y_{Second-Best}|x_i)$ es la segunda probabilidad de clase más alta.

$$x_{BvSB} = p\left(y_{Best}|x_i\right) - p\left(y_{Second-Best}|x_i\right), \forall\, x_i \in X \quad (9)$$

Los valores con incertidumbre más alta se denominan en este trabajo "fronteras", debido a que son ejemplos que se encuentran muy cercanos a una frontera de separación entre dos o más clases, y los ejemplos con menor incertidumbre serán los que se encuentran más alejados y se denominan "centros". Los candidatos escogidos para esta implementación son los ejemplos cuya medida BvSB se encuentre más cercana a la mediana de los valores, que se denominan "medios".

El componente de selección de candidatos se muestra en el **Algoritmo 3**. Primero se calcula el BvSB de todos los elementos del lote usando la salida obtenida desde la red según la Ecuación (9) (líneas 2-5), seguidamente se ordenan los elementos según el valor de BvSB de manera ascendente (línea 6) y se calculan los índices donde se encuentran los candidatos tomando en cuenta el número máximo de candidatos por lote (líneas 7-9), y por último se retorna el vector con los candidatos seleccionados (línea 10).

Algoritmo 3 Selección de Candidatos para RILBC	
Entradas:	B: lote de datos de entrada, O: vector con las salidas de la red después del entrenamiento, c: número de candidatos a ser seleccionados en cada lote
Salidas:	un vector con los representantes seleccionados del lote
1.	**Inicio**
2.	Crear V como un vector vacío
3.	Dividir el lote de datos, en entradas y etiquetas (X, Y) ← B
4.	Calcular la métrica BvSB para todas las muestras según la Ecuación (9), tomando *O* como el vector correspondiente a las etiquetas de salida de la RNA $V \leftarrow \left(p(o_{Best}\|x_i) - p(o_{Second-Best}\|x_i)\right), \forall\, x_i \in X$
5.	Crear D como un conjunto formado por X, Y y V (D← X || Y || V)
6.	Ordenar D según los valores de D_V de forma ascendente
7.	Crear t como el índice inferior para extraer los candidatos de D $t \leftarrow \max(\frac{\|B\|}{2} - \frac{c}{2}, 0)$
8.	Crear u como el índice superior para extraer los candidatos de D $u \leftarrow \min(\frac{\|B\|}{2} + \frac{c}{2}, \|B\|)$
9.	Crear Br como los candidatos de D ubicados entre los índices inferior y superior $Br \leftarrow (D_t, D_i, \dots, D_{u-1}), donde\ t \leq i < u$
10.	**Retornar** Br
11.	**Fin**

Luego de seleccionar los candidatos se realiza el proceso de "Actualización de Representantes", en este paso se usa la distancia de *Crowding*, utilizada en

algoritmos de optimización multiobjetivo para distribuir en el diagrama de Pareto diferentes soluciones, que además ha sido usada para distribuir apropiadamente soluciones en el espacio de características cuando se realiza un muestreo [66]. La distancia de *Crowding* se usa en **RILBC** buscando seleccionar las muestras representantes que mejor se distribuyan en el espacio de características respecto al total de muestras.

La Actualización de Representantes en **RILBC** funciona de la siguiente forma: se calcula la distancia de *Crowding* de cada uno de los representantes y candidatos actuales, a partir de la salida de la red asociada a cada representante, siguiendo el **Algoritmo 4**, el cual es una modificación del encontrado en [67]. Para cada una de las clases (línea 2), se asigna una distancia de *Crowding* 0 a sus representantes (líneas 3-5), seguidamente se realiza un ordenamiento por cada una de las salidas de la red (línea 7), se toma el vector ordenado y seguidamente se calcula la distancia euclidiana entre la salida de la red de cada uno de los representantes con la de sus dos vecinos inmediatos, es decir su vecino anterior y siguiente en el arreglo ordenado (líneas 8-11), teniendo en cuenta que la distancia euclidiana entre dos puntos $P = (p_1, p_2, \dots, p_n)$ y $Q = (q_1, q_2, \dots, q_n)$ se define siguiendo la Ecuación (10).

$$d_{eucl}(P, Q) = \sqrt{\sum_{i=1}^{n}(p_i - q_i)^2} \quad (10)$$

El cálculo de la distancia se hace únicamente entre las muestras de una misma clase. Posteriormente, se ordenan las muestras de cada clase según la distancia de *Crowding*, y para cada una, se seleccionan las *r* muestras con una mayor distancia total. Son estas muestras las que se preservan como representantes.

Algoritmo 4 Distancia de Crowding	
Entradas:	R: matriz de representantes actuales, N: número de salidas/clases que soporta la RNA
Salidas:	R con la distancia de Crowding asignada a cada representante
1.	**Inicio**
2.	**Para** cada clase $C_k \in R$ **Hacer**
3.	**Para** cada representante $R_j \in C_k$ **Hacer**
4.	$R_j.dist\_crowding \leftarrow 0$
5.	**Fin Para**
6.	**Para** ***i*** desde 0 hasta N **Hacer**
7.	$R \leftarrow ordenado\ por\ la\ salida\ de\ la\ red\ \boldsymbol{i}\ del\ representante$
8.	**Para** ***j*** desde 1 hasta $\|R\| - 1$ **Hacer**
9.	$R_j.dist\_crowding \leftarrow R_j.dist\_crowding + d_{eucl}(R_j.salida\_red[i], R_{j-1}.salida\_red[i])$
10.	$R_j.dist\_crowding \leftarrow R_j.dist\_crowding + d_{eucl}(R_{j+1}.salida\_red[i], R_j.salida\_red[i])$
11.	**Fin Para**
12.	**Fin Para**
13.	**Fin Para**
14.	**Retornar** R

15.	**Fin**

Por último, para el componente “Regularizador de Muestras” se aplica la misma estrategia propuesta en **NIL**.

Con el propósito de facilitar el entendimiento del algoritmo RILBC, se presenta un ejemplo sencillo de la ejecución de sus fases de selección de candidatos y del cálculo de la distancia de *Crowding*, en el cual se usa una RNA con 3 salidas para clasificar imágenes de números del 1 al 3. En este ejemplo, el tamaño del lote es 10, el número candidatos *c* es 4, y el número de representantes por clase *r* es 5.

La **Figura 29** muestra el proceso de selección de candidatos. Primero se establece el valor más alto obtenido en la salida de la red (***Best***) y el segundo más alto (***Second Best***) para cada una de las muestras del lote. Luego se realiza la resta de estos valores (***BvSB***) y se hace un ordenamiento con respecto a ese resultado, luego los candidatos son los que se encuentran dentro del rango que se calcula a partir de la mediana y el número de candidatos máximo (4).

Clase	Salida 1	Salida 2	Salida 3	Best	Second Best	BvSB	Ind.	
3	0.2	0.8	0.9	0.9	0.8	0.1	0	
3	0.3	0.6	0.5	0.6	0.5	0.1	1	
1	0.4	0.3	0.2	0.4	0.3	0.1	2	
3	0.4	0.5	0.8	0.8	0.5	0.3	3	→ Límite inferior = 3 = (5-4/2)
1	0.6	0.2	0.1	0.6	0.2	0.4	4	
2	0.3	0.8	0.4	0.8	0.4	0.4	5	→ Mediana = 5
3	0.1	0.9	0.5	0.9	0.5	0.4	6	
1	0.7	0.1	0.2	0.7	0.2	0.5	7	→ Límite superior = 7 = (5+4/2)
1	0.9	0.3	0.4	0.9	0.4	0.5	8	
2	0.9	0.1	0.1	0.9	0.1	0.8	9	

Figura 29 *Ejemplo de Selección de candidatos RILBC*

La siguiente parte es la actualización de representantes, en la **Figura 30** se muestra la lista de representantes temporal para la clase 3, en la cual se incorporan los representantes actuales de dicha clase y los candidatos escogidos en la clase anterior cuya etiqueta corresponda a la clase 3.

	ETIQUETA CLASE	SALIDA 1	SALIDA 2	SALIDA 3
REPRESENTANTES ACTUALES	3	0.4	0.2	0.8
	3	0.5	0.5	0.7
	3	0.3	0.1	0.9
	3	0.2	0.3	0.9
	3	0.4	0.3	0.8
CANDIDATOS	3	0.4	0.5	0.8
	3	0.1	0.9	0.5

Figura 30 *Resultado de selección de candidatos RILBC*

Posteriormente se hace el cálculo de la **Distancia de *Crowding (CD)***. La **Figura 31** muestra la primera ejecución del algoritmo donde se calcula la CD para la

primera salida, por lo tanto, se encuentra ordenada con respecto a ella, el algoritmo llena con ceros los representantes extremos (primer y último elemento) y calcula las CD parciales para los demás, este proceso se repite ordenando los valores por cada salida de la red y acumulando los resultados parciales de CD. Finalmente se ordenan las muestras según el total de su CD y se escogen las 5 muestras con el mayor valor (para este ejemplo), tal y como se muestra en la **Figura 32**.

$d_{eucl}(0.3,0.2) + d_{eucl}(0.3,0.4) = 0.2$

Etiqueta clase	Salida 1	Salida 2	Salida 3	CD Salida 1	CD Acumulada
3	0.1	0.9	0.5	0	0
3	0.2	0.3	0.9	0.2	0.2
3	0.3	0.1	0.9	0.2	0.2
3	0.4	0.3	0.8	0	0
3	0.4	0.5	0.8	0.1	0.1
3	0.4	0.2	0.8	0.1	0.1
3	0.5	0.5	0.7	0	0

Figura 31 *Ejemplo de la Iteración 1 del cálculo de la distancia de Crowding*

Etiqueta clase	Salida 1	Salida 2	Salida 3	CD Salida 1	CD Salida 2	CD Salida 3	CD Acumulada
3	0.1	0.9	0.5	0	0	0	0
3	0.4	0.3	0.8	0	0.2	0	0.2
3	0.4	0.5	0.8	0.1	0	0.1	0.2
3	0.3	0.1	0.9	0.2	0	0.1	0.3
3	0.2	0.3	0.9	0.2	0.1	0	0.3
3	0.4	0.2	0.8	0.1	0.2	0.1	0.4
3	0.5	0.5	0.7	0	0.4	0.3	0.7

NUEVOS REPRESENTANTES

Figura 32 *Ejemplo de Actualización de representantes para RILBC*

CAPÍTULO 5

5 EXPERIMENTOS Y RESULTADOS

5.1 DATASETS

Se seleccionaron cuatro datasets de diversa complejidad comúnmente usados en investigaciones de aprendizaje incremental, los cuales proporcionan distintos escenarios de prueba para la ejecución de los experimentos. Estos datasets son: ***MNIST*** [54], ***Fashion-MNIST*** [60], ***CIFAR-10*** [56] y ***Caltech-101*** [68]. La configuración detallada para cada dataset se presenta en la **Figura 33** y en la **Figura 34**, se escogen cinco megalotes porque esta cantidad permitió tener megalotes con el mismo número de clases en el caso incremental y un suficiente número de incrementos en cada experimento para mostrar un aprendizaje incremental.

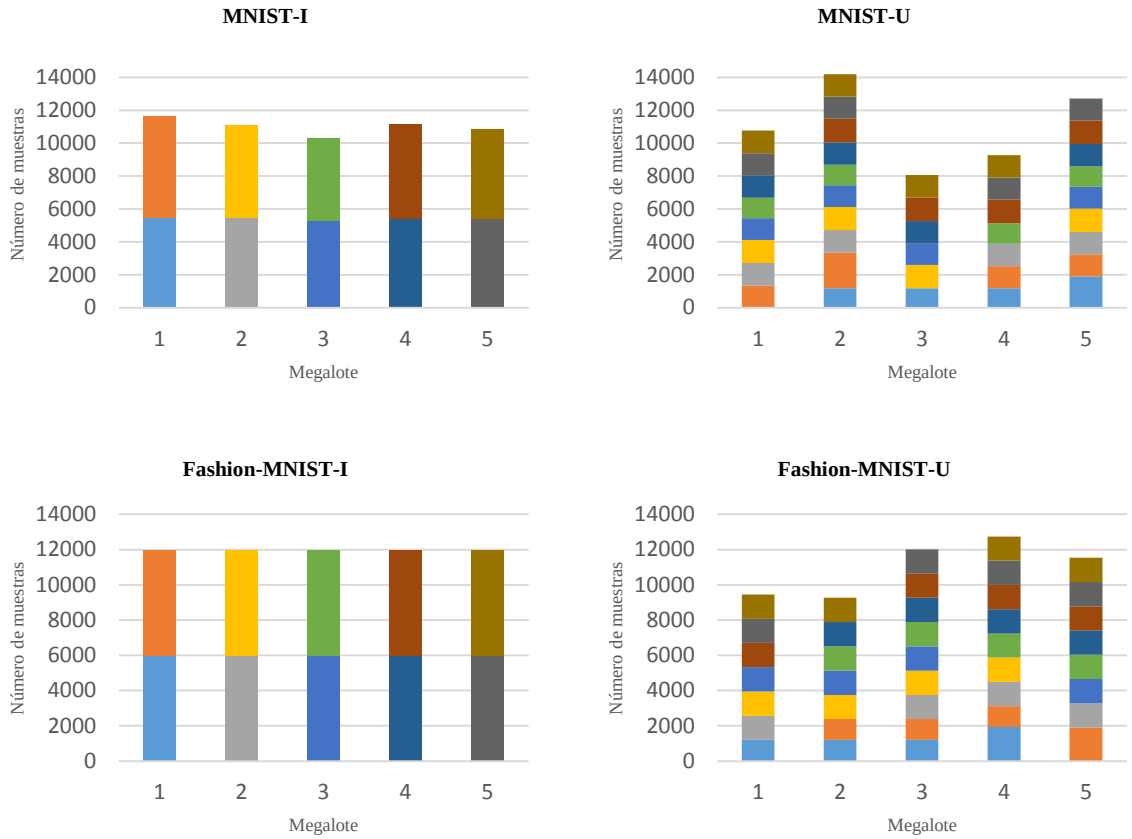

Figura 33 *Distribución de datos por clase y megalote para MNIST y Fashion-MNIST. Cada color representa una clase distinta*

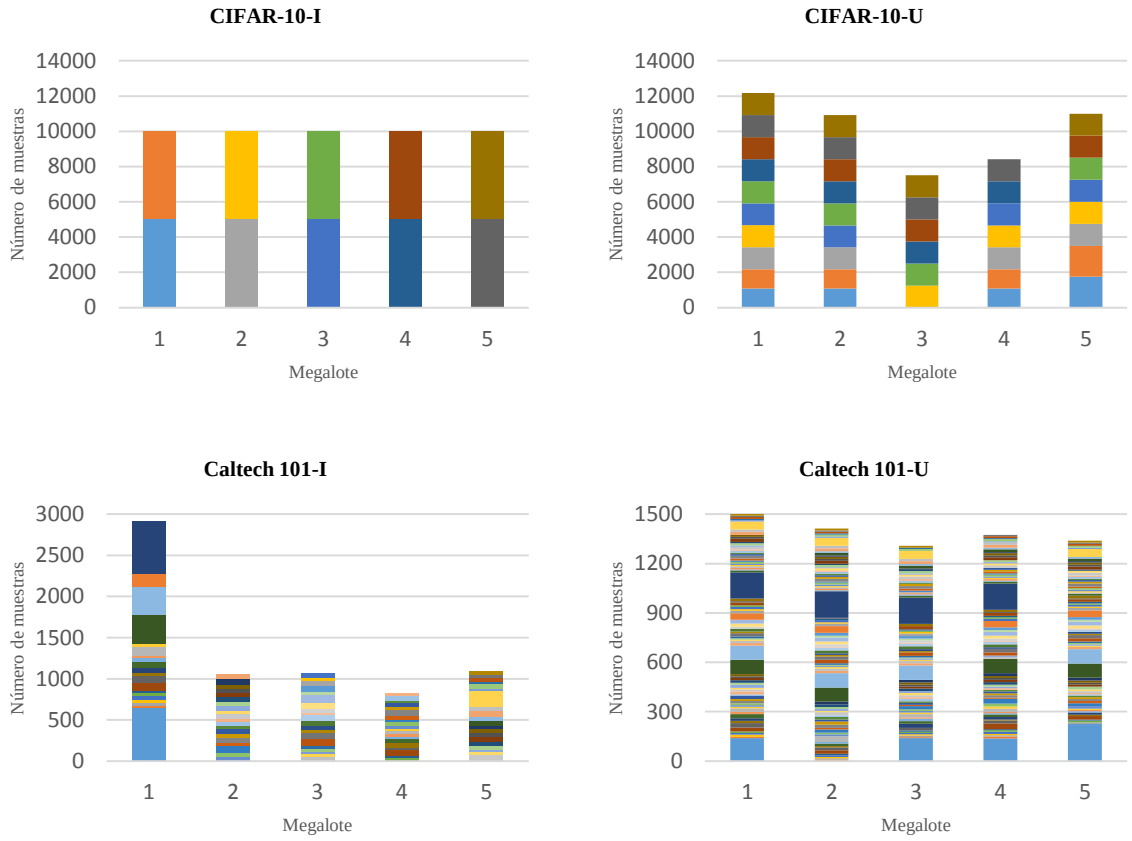

Figura 34 *Distribución de datos por clase y megalote para CIFAR-10 y Caltech 101. Cada color representa una clase distinta*

Los datos de cada uno de los datasets se organizaron de dos maneras: i) clases incrementales, i.e. cada clase aparece en un único megalote, y corresponden a los datasets del lado izquierdo de la figura con el sufijo "-I" y ii) clases desbalanceadas, i.e. las clases aparecen en múltiples megalotes en diferentes proporciones, y corresponden a los datasets del lado derecho de la figura con el sufijo "-U".

Para cada uno de los datasets, los datos totales se dividieron en 5 megalotes, y en el escenario de clases incrementales se repartió de forma equitativa el número de clases exclusivas en cada megalote. En el caso de Caltech 101 se removió una clase para una repartición equitativa en número de clases. En los datasets de clases desbalanceadas las clases fueron repartidas de tal forma que se distribuyen en 4 de los 5 megalotes, pero en diferente proporción en cada uno.

Por otro lado, para los conjuntos de prueba de MNIST, Fashion-MNIST y CIFAR-10 se tomaron los conjuntos que vienen previamente separados de los datasets

originales. Para Caltech 101 se hizo una división en la cual el 80% de los datos de cada clase se usó para entrenamiento y el 20% para pruebas.

5.2 ARQUITECTURAS DE DNNS

Se seleccionó una arquitectura de red especifica para cada dataset, conforme lo recomendado en el estado del arte. Esto ademas permitió evaluar la adaptación del enfoque a esas diferentes arquitecturas. Las arquitecturas seleccionadas fueron ***LeNet*** para MINIST [53], ***FashionMnistNet*** para Fashion-MNIST [59], ***CifarTFNet*** para CIFAR-10 [55] y ***AlexNet*** para Caltech-101 [69]. Los detalles de cada arquitectura se muestran en la **Tabla 6**.

A la única arquitectura que se le realizaron modificaciones fue a ***AlexNet***; ya que se removió la última capa FC y se redujo la cantidad de neuronas de las 2 capas FC restantes a 2048 y 1024, respectivamente, para asegurar que los equipos de pruebas fueran capaces de ejecutar los experimentos con esta red. También se fijaron 100 neuronas en la capa de salida. Adicionalmente, se usó Transfer Learning tomando como base una red pre entrenada con datos de ***ImageNet***, y además se congelaron los pesos hasta la última capa de Pooling.

Tabla 6 *Configuraciones de experimentación para cada dataset*

Dataset	RNA	Épocas	Tam. Lote	Tasa Aprendizaje	Características
MNIST	LeNet	25	128	0.0001	Conv (5x5) – ReLu – Max. Pool (2x2) – Conv (5x5) – ReLu – Max. Pool (2x2) – FC (120) – FC (84) – FC (10)
Fashion-MNIST	Fashion-MnistNet	100	256	0.0001	Conv (2x2) – ReLu – Max. Pool (2x2) – Dropout (30%) – Conv (2x2) – ReLu – Max. Pool (2x2) – Dropout (30%) – FC (256) – Dropout (50%) – FC (10)
CIFAR-10	Cifar-TFNet	100	128	0.0001	Conv (5x5) – ReLu – Max. Pool (3x3) – LRN – Conv (5x5) – ReLu – LRN – Max. Pool (3x3) – FC (384) – Dropout (60%) – FC (192) – FC (10)
Caltech 101	Modified AlexNet	60	128	0.0001	Conv (11x11) – ReLu – LRN – Max. Pool (3x3) – Conv (5x5) – ReLu – LRN – Max. Pool (3x3) – Conv (3x3) – ReLu – Conv (3x3) – ReLu – Conv (3x3) – ReLu – Max. Pool (3x3) – FC (2048) – Dropout (50%) – FC (1024) – Dropout (50%) – FC (100)

5.3 ALGORITMOS COMPARADOS

Los algoritmos propuestos, **NIL** y **RILBC** se compararon con **iCaRL**, **RMSProp Ac** (define el valor **Máximo de exactitud** posible y corresponde a la versión de RMSProp [8] con ***Full Rehearsal***, i.e. cada vez que se va a entrenar la red se toman los datos del megalote actual junto con todos los datos de los megalotes previamente presentados a la red) y como línea base **RMSProp Inc** (que

corresponde a la versión de RMSProp incremental, i.e. donde se toma la red con los pesos definidos en el megalote anterior y se entrena sólo con los datos del nuevo megalote, excepto para el primer megalote donde los pesos se definen inicialmente en forma aleatoria)*.*

Para **NIL**, **RILBC** e **iCaRL** se usa RMSProp como optimizador base para entrenar la RNA y se evalúan dos escenarios: uno donde se almacena una cantidad máxima de representantes equivalente al 1% del tamaño total del dataset para así observar el comportamiento cuando se cuenta con pocos representantes, y otro donde no supere el 10% para **NIL** e **iCaRL**, y 5% para **RILBC** para observar el comportamiento con una mayor cantidad de representantes. Por otro lado, **iCaRL** únicamente fue evaluado con los datasets de clases incrementales, dado que el algoritmo fue propuesto y diseñado para este caso de pruebas, y no fue posible adaptarlo a los experimentos con datasets de clases desbalanceadas sin cambiar la propuesta original de sus autores. En la **Tabla 7** se presenta un resumen de los escenarios de experimentación de cada algoritmo.

Adicionalmente, para **NIL** y **RILBC** se estableció un tamaño de buffer *q* de 1 iteración para todos los experimentos, y un número de candidatos *r* de 20 para todos los datasets, excepto para Fashion-MNIST, cuyo *r* es de 40, teniendo en cuenta que el tamaño del lote es mayor para este dataset.

Tabla 7 *Escenarios de experimentación para cada algoritmo*

Algoritmo	**Escenarios**		**Otros parámetros**
NIL	1% reps.	10% reps.	$q = 1$ y $r = 20$ ($r = 40$ para Fashion-MNIST)
RILBC	1% reps.	5% reps.	$q = 1$ y $r = 20$ ($r = 40$ para Fashion-MNIST)
iCaRL	1% reps.	10% reps.	N/A

5.4 PROTOCOLO DE EXPERIMENTACIÓN

Para realizar la evaluación y las comparaciones entre los diferentes escenarios de evaluación se realizaron mediciones de pérdida (*loss*), exactitud y tiempo, con el objetivo de evaluar tanto la calidad de la clasificación (desempeño de la clasificación) como la rapidez de cada algoritmo. Estas mediciones se realizaron tomando el resultado promedio de 30 ejecuciones. Para evaluar la exactitud se usa un dataset de pruebas único para todos los megalotes, el cual contiene todas las clases del dataset en la misma proporción que el dataset de entrenamiento completo.

Para propósitos de comparación de tiempos de ejecución, los experimentos sobre los datasets MNIST y CIFAR-10 se realizaron sobre un computador Dell Inspiron 15 7000, con CPU Intel Core i7-7700HQ, 8GB de RAM y una GPU NVIDIA 1050Ti. Los experimentos sobre los datasets Fashion-MNIST y Caltech 101 se ejecutaron

sobre un computador Asus Predator Helios 300, con CPU Intel Core i7-7700HQ, 16GB de RAM y una GPU NVIDIA 1060. Estudios preliminares se realizaron en un equipo con tarjeta NVIDIA Titan Xp que se comparte con otros investigadores, los cuales sirvieron para afinar los modelos propuestos.

Los algoritmos **RMSProp**, **NIL** y **RILBC** fueron implementados sobre **DILF** [51], que se encuentra construido sobre TensorFlow [70]. Por otro lado, aunque se encuentra disponible una implementación de **iCaRL** [34] provista por sus autores, no fue posible usarla debido a que, al realizar las modificaciones necesarias para usarlo con diversos datasets, no se logró repetir los resultados publicados por los autores. Debido a eso, para **iCaRL** se tomó como base la implementación usada por los autores de [71], la cual se encuentra construida sobre Pytorch [72], y se le hicieron las modificaciones respectivas para adaptarla al esquema de pruebas usado en este trabajo. Por otro lado, para **iCaRL** únicamente se registraron los resultados de exactitud al final de cada megalote, debido a que su mecanismo de clasificación sólo se actualiza al final de cada megalote.

5.5 RESULTADOS EXPERIMENTALES EN MNIST

En la **Figura 35** se muestran los resultados para los distintos algoritmos en **MNIST-I** y en **MNIST-U**. Se encuentra que en **MNIST-I NIL 1%** y **RILBC 1%** obtienen ambos un resultado del 90% de exactitud, mientras que **NIL 10%** y **RILBC 5%** obtienen un 95.7% y 94.2% de exactitud, resultados que son muy similares. Por su parte, ambas versiones de **iCaRL** obtienen alrededor de 70% de exactitud, mientras que **RMSProp Inc** no logra mejorar su exactitud a lo largo del entrenamiento y se queda en 20%. Adicionalmente, al observar las gráficas de *loss* se encuentra que en los cambios de megalotes **RMSProp Inc** tiene un aumento importante en el *loss*, lo que indica un mayor grado de inestabilidad en comparación con los otros algoritmos. Por otro lado, también se observa que en **MNIST-I** los algoritmos con mejores resultados en exactitud (**NIL 10%** y **RILBC 5%**) tienen los menores aumentos en el *loss* en los cambios de megalote.

Para **MNIST-U**, todas las versiones de **NIL** y **RILBC** obtienen resultados muy similares, valores que están alrededor de 98%, y aunque **RMSProp Inc** alcanza estos valores en momentos determinados, lo hace de forma inestable y llega a caer hasta el 60%. Siendo así, se encuentra que el mejor resultado lo obtiene **NIL 10%**, que con 95.7% y 98.6% se acerca bastante a los resultados de **RMSProp Ac (Max. Exactitud)**, que alcanza 98.2% y 98.8% respectivamente. Por otra parte, al analizar las gráficas de *loss* en **MNIST-U** se observa que tanto **NIL** como **RILBC** presentan un comportamiento más estable que en **MNIST-I**, ya que sólo presentan un aumento significativo en el *loss* en el primer cambio de megalote.

La **Tabla 8** muestra los tiempos de entrenamiento de cada algoritmo en cada megalote (1, 2, 3, 4 y 5), el tiempo total acumulado después de pasar los 5 megalotes, el porcentaje de tiempo adicional que usa el algoritmo en relación con

RMSProp Inc, (%I) y el porcentaje de tiempo adicional que usa el algoritmo en relación con RMSProp Ac (%A). En relación con **MNIST-I** se puede observar que **NIL 1%** es el algoritmo más rápido, tan sólo usa un 43% más que **RMSProp Inc** y demora un 32% menos del tiempo total usado por **RMSProp Ac**. Por su lado **NIL 10%** y **RILBC 1%** incurren en aproximadamente el mismo sobrecosto que **RMSProp Ac** sobre lo obtenido por **RMSProp Inc**, de un 104% a 110% adicional. Estos tres algoritmos (**NIL 1%, NIL 10%** y **RILBC 1%**) cumplen con el objetivo del aprendizaje incremental, y mantienen niveles altos de exactitud sin superar el tiempo del entrenamiento con todos los datos disponibles. Las propuestas **RILBC 5%** e **iCaRL (1% y 10%)** sobrepasan por mucho (255% a 356% adicional) el tiempo empleado por el entrenamiento con todos los datos (**RMSProp Ac**) y no deben ser consideradas.

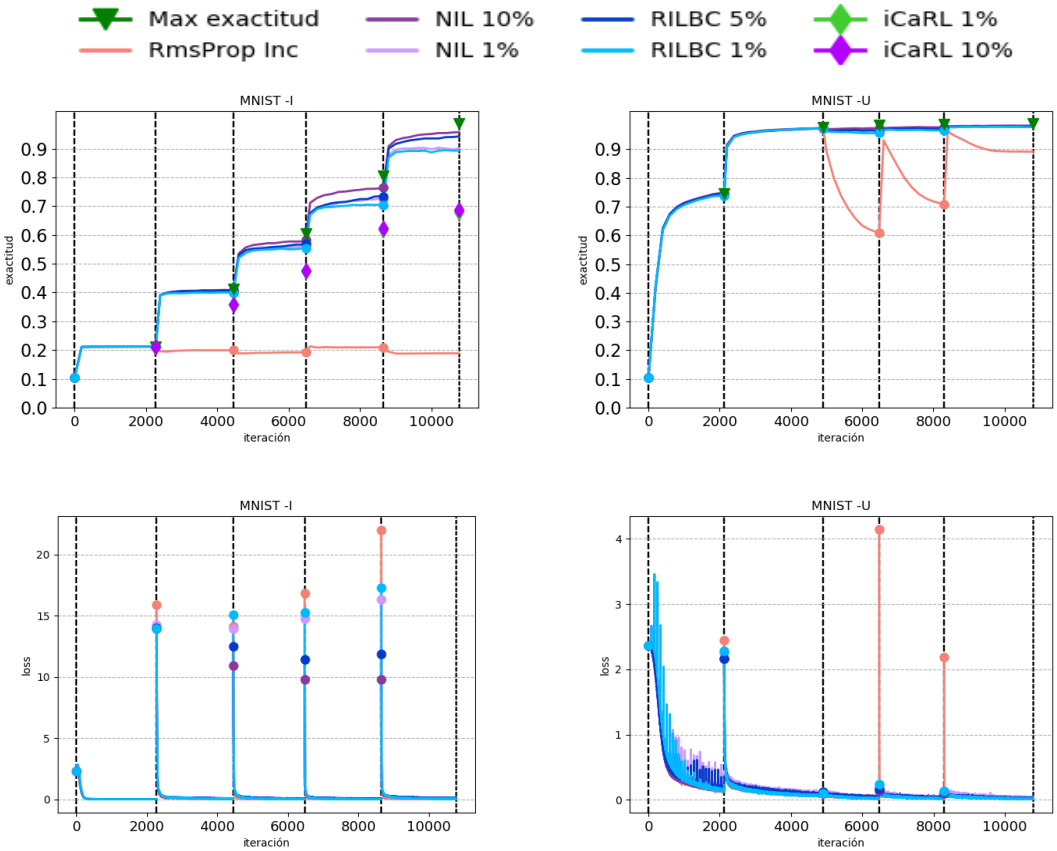

Figura 35 *Resultados de exactitud y pérdida en MNIST. Las líneas verticales representan el cambio de megalote. Los gráficos a la izquierda muestran los resultados cuando cada megalote incluye nuevas clases (incremental) y los gráficos a la derecha muestran los resultados cuando se usan megalotes desbalanceados*

En relación con **MNIST-U**, los resultados de **NIL 1%** siguen siendo los mejores en tiempo, ya que sólo tarda un 45% más que el uso de **RMSProp Inc** y demora un 30% menos que **RMSProp Ac** (todos los datos de entrenamiento). Por su parte,

NIL 10% y **RILBC 1%** incurren en un sobrecosto un poco mayor al de **RMSProp Ac** (18% a 22% adicional) sobre lo obtenido por **RMSProp Inc**, de 146% a 155% adicional, lo que pone en duda su uso con este dataset, que en general es bastante sencillo. Finalmente, **RILBC 5%** sobrepasa por mucho (545% adicional) el tiempo empleado por el entrenamiento con todos los datos (**RMSProp Ac**) y no debe ser considerado.

Tabla 8 *Tiempos promedio de entrenamiento (segundos) para cada algoritmo en MNIST-I y MNIST-U. Se proveen los tiempos para cada megalote*

Algoritmo	MNIST-I								MNIST-U							
	1	2	3	4	5	Total	%I	%A	1	2	3	4	5	Total	%I	%A
RMSProp Inc	17	16	16	17	16	82	-	-	16	21	12	14	19	82	-	-
RMSProp Ac	17	27	39	40	49	172	110%	-	16	29	39	38	49	171	109%	-
NIL 1%	23	23	23	24	24	117	**43%**	**-32%**	23	30	18	21	27	119	**45%**	**-30%**
NIL 10%	27	30	33	38	43	171	109%	**-1%**	36	54	33	36	50	209	155%	22%
RILBC 1%	27	30	32	38	40	167	104%	**-3%**	36	53	30	35	48	202	146%	18%
RILBC 5%	43	60	71	93	107	374	356%	117%	88	140	81	93	127	529	545%	209%
iCaRL 1%	50	57	58	59	68	291	255%	69%								
iCaRL 10%	74	66	66	67	85	359	338%	109%								

5.6 RESULTADOS EXPERIMENTALES EN FASHION-MNIST

En la **Figura 36** se muestran los resultados del dataset **Fashion-MNIST-I** y **Fashion-MNIST-U**. Se observa que en **Fashion-MNIST-I**, **NIL 10%** obtiene el mejor valor de exactitud, con 83.3% el cuál es un resultado cercano al de **RMSProp Ac** (**Max. Exactitud**) que alcanza un 88%, y **NIL 1%** obtiene valores muy cercanos a **NIL 10%**. En el caso de **RILBC** se encuentran resultados similares en sus dos versiones, donde **RILBC 1%** alcanza un 78.9% y **RILBC 5%** llega a 80.2%, Una situación similar se presenta en **iCaRL**, donde **iCaRL 1%** obtiene un 58.1% y la versión **iCaRL 10%** sólo llega a 59.6%. Por último, **RMSProp Inc** se estanca en aproximadamente un 20% a lo largo de todo el entrenamiento, y presenta un sobre entrenamiento similar al visto en MNIST-I.

En el caso de **Fashion-MNIST-U NIL 10%** obtiene 86.3%, acercándose más a **RMSProp Ac** (**Max. Exactitud**) y reduciendo su diferencia a menos de 2%, y al igual que con el dataset incremental, **NIL 1%** obtiene resultados muy similares a **NIL 10%**. **RILBC** tanto en 1% como en 5% alcanza un valor alrededor del 84%, resultado que pone a **RILBC 1%** a la delantera debido a sus menores requerimientos de memoria y tiempo de ejecución. **RMSProp Inc** logra una exactitud de 72% al final del entrenamiento, aunque logra alcanzar un pico de 80%, lo que indica una gran inestabilidad durante el entrenamiento con los datasets desbalanceados.

Por otro lado, en cuanto a los resultados del *loss* se encuentra un comportamiento de gran inestabilidad en el dataset de **Fashion-MNIST-U** en los megalotes 2 y 5 con el algoritmo **RILBC** (1% y 5%). También se puede observar el comportamiento

inestable del algoritmo **RMSProp Inc** que alcanza picos más grandes a medida que llegan los nuevos megalotes. **NIL** por su parte tiene un comportamiento significativamente más estable, siendo que los aumentos en el *loss* al cambiar de megalote tienden a disminuir a lo largo del entrenamiento.

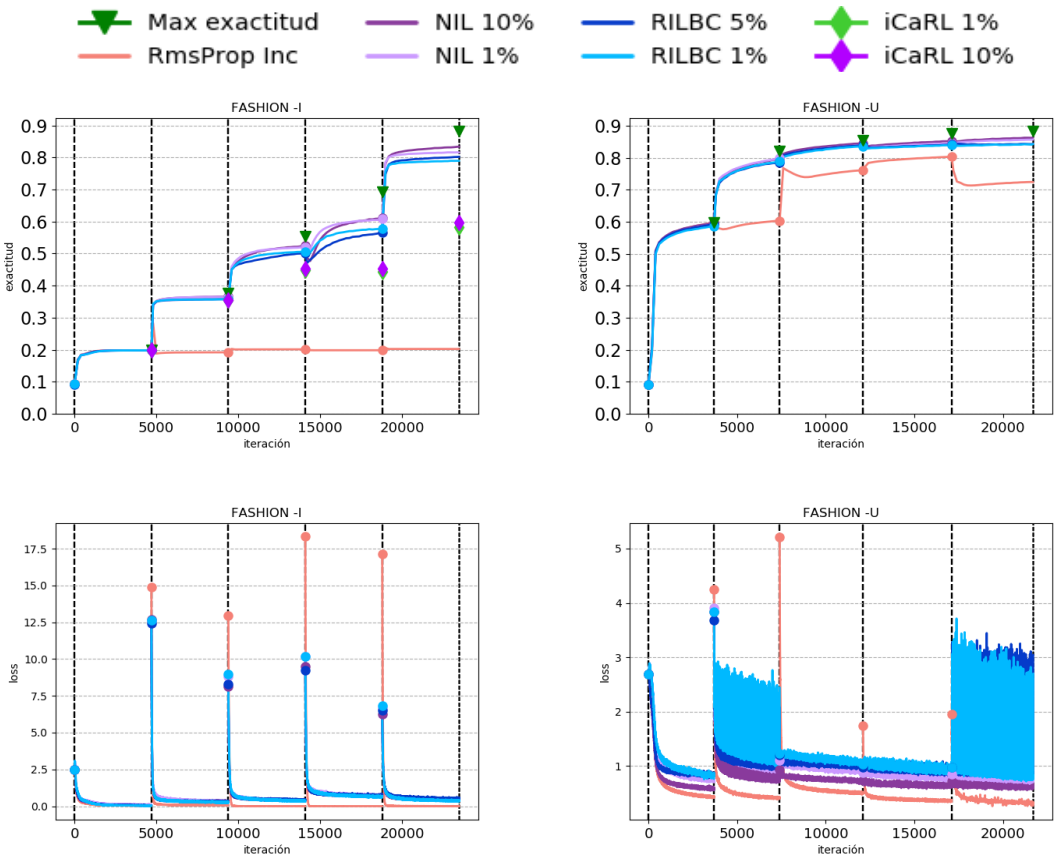

Figura 36 *Resultados de exactitud y pérdida en Fashion-MNIST*

Al igual que con el dataset anterior, la **Tabla 9** muestra los resultados asociados con los tiempos de entrenamiento de cada algoritmo. En relación con **Fashion-MNIST-I** se puede observar que **NIL 1%** es el algoritmo más rápido, tan sólo usa un 24% más tiempo que **RMSProp Inc** y demora un 14% menos del tiempo total usado por **RMSProp Ac**. Por su lado **NIL 10%** y **RILBC 1%** incurren en aproximadamente el mismo sobrecosto que **RMSProp Ac** sobre lo obtenido por **RMSProp Inc**, un 47-52% de tiempo adicional. Los algoritmos **iCaRL (1% y 10%)** también obtienen tiempos competitivos, usando un 4-11% menos del tiempo que **RMSProp Ac**. Por último, la propuesta **RILBC 5%** sobrepasan por mucho (174% adicional) el tiempo empleado por el entrenamiento con todos los datos (**RMSProp Ac**) y no debe ser considerado.

En relación con **Fashion-MNIST-U**, los resultados de **NIL 1%** siguen siendo los mejores en tiempo, ya que sólo tarda un 26% más que el uso de **RMSProp Inc** y

demora 7% menos que **RMSProp Ac** (todos los datos de entrenamiento). Por su parte, **NIL 10%** y **RILBC 1%** incurren en un sobrecosto un poco mayor al de **RMSProp Ac** (19% adicional) y sobre lo obtenido por **RMSProp Inc**, un 61% a 62% adicional, lo que también pone en duda su uso con este dataset. Finalmente, **RILBC 5%** sobrepasa por mucho (214% adicional) el tiempo empleado por el entrenamiento con todos los datos (**RMSProp Ac**) y no debe ser considerado.

Tabla 9 *Tiempos promedio de entrenamiento (segundos) para cada algoritmo en Fashion-MNIST-I y Fashion-MNIST-U. Se proveen los tiempos para cada megalote*

Algoritmo	Fashion-MNIST-I								Fashion-MNIST-U							
	1	**2**	**3**	**4**	**5**	**Total**	**%I**	**%A**	**1**	**2**	**3**	**4**	**5**	**Total**	**%I**	**%A**
RMSProp Inc	101	102	102	102	103	510	-	-	79	79	102	107	97	464	-	-
RMSProp Ac	101	153	154	204	129	741	45%	-	79	119	131	184	117	630	36%	-
NIL 1%	124	126	127	128	129	634	**24%**	**-14%**	99	99	128	136	124	586	**26%**	**-7%**
NIL 10%	132	141	150	159	167	749	47%	1%	119	128	164	176	160	747	61%	19%
RILBC 1%	135	146	155	164	175	775	52%	5%	121	128	165	177	161	752	62%	19%
RILBC 5%	173	234	293	330	368	1398	174%	89%	208	255	326	348	319	1456	214%	131%
iCaRL 1%	125	131	132	132	140	661	30%	**-11%**								
iCaRL 10%	143	139	139	139	153	713	40%	**-4%**								

5.7 RESULTADOS EXPERIMENTALES EN CIFAR-10

En la **Figura 37** se muestran los resultados del dataset **CIFAR-10-I** y **CIFAR-10-U**, y se aprecia que en **CIFAR-10-I NIL 10%** obtiene el mejor resultado con una exactitud de 57.5% siendo 14% menor al alcanzado por **RMSProp Ac** de 71.6% (**Max Accuracy**). **RILBC 5%** solo llega a un 50.3% y **RILBC 1%** sólo llega a una exactitud de 34.4%, resultado muy cercano al reportado en **NIL 1%**. **iCaRL** tiene un pobre resultado, ya que ninguna de sus dos versiones supera una exactitud de 17% y es incluso superado por **RMSProp Inc** que obtiene un 17.9% de exactitud.

En el caso de **CIFAR-10-U** los resultados son mejores, ya que **NIL 10%** obtiene un 68.9%, siendo éste un resultado cercano al reportado en **RMSProp Ac**, el cual alcanza un 73.1% de exactitud. **RILBC 1%** y **5%** logran una exactitud de 62% y 66% respectivamente, siendo estos resultados bastante cercanos. **RMSProp Inc** presenta un alto nivel de sobre entrenamiento que conlleva a un pobre rendimiento cuando hay cambio de megalote, cómo se observa en el megalote 2 dónde su accuracy cae de 63% a 47%.

Por otro lado, en cuanto a los resultados del *loss* se encuentra que el entrenamiento con **NIL** y **RILBC** es más estable que con **RMSProp Inc**, siendo que sus aumentos en el valor de *loss* al cambiar de megalote son menores. Adicionalmente se observa una mayor estabilidad general en los valores de *loss* en **CIFAR-10-U** respecto a **CIFAR-10-I**, lo cual se explica porque el cambio en la

distribución de los datos entre distintos megalotes es más acentuada en **CIFAR-10-I**, ya que cada megalote posee clases distintas que no se encuentran en los demás.

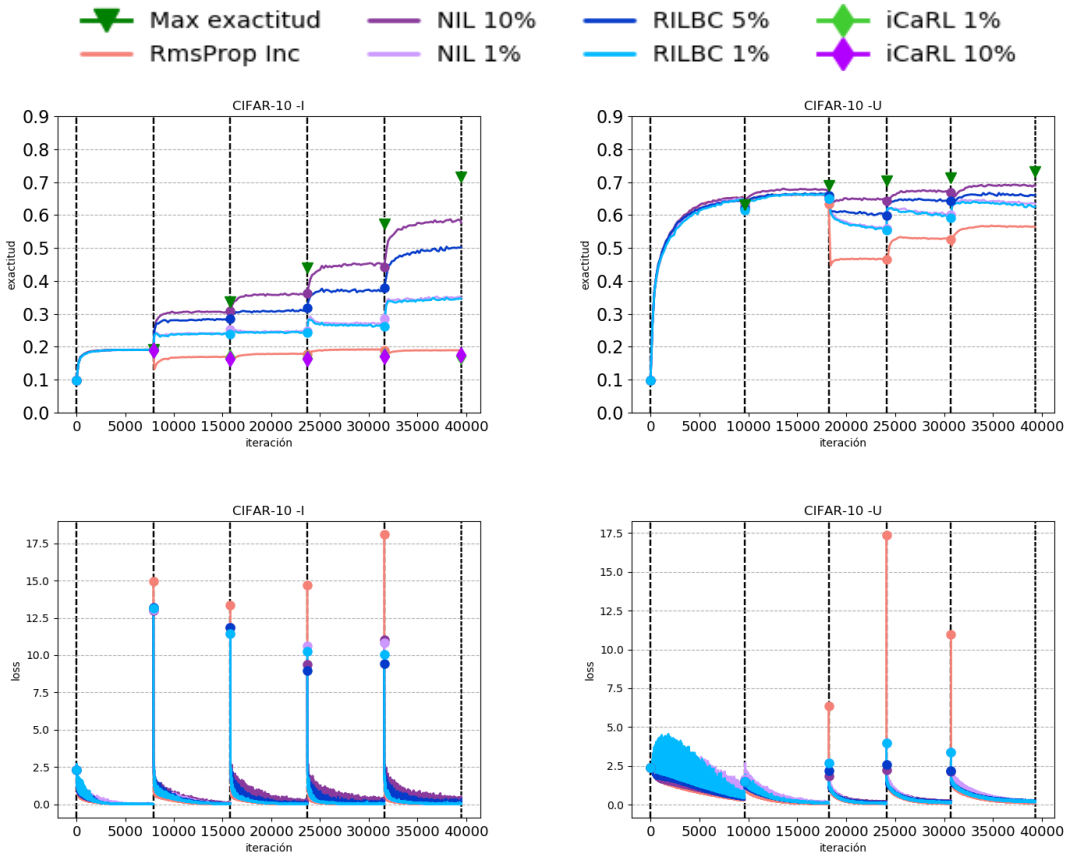

Figura 37 *Resultados de exactitud y pérdida en CIFAR-10*

Al igual que con los datasets anteriores, la **Tabla 10** muestra los resultados asociados con los tiempos de entrenamiento de cada algoritmo. En relación con **CIFAR-10-I** se puede observar que **NIL 1%** es el algoritmo más rápido, tan sólo usa un 26% más tiempo que **RMSProp Inc** y demora un 14% menos del tiempo total usado por **RMSProp Ac**. Por su lado **NIL 10%** y **RILBC 1%** incurren en aproximadamente el mismo sobrecosto, tan solo un 36% más tiempo que **RMSProp Inc** y demora un 11% menos que **RMSProp Ac**. Los algoritmos **iCaRL (1% y 10%)** también obtienen tiempos competitivos, usando un 1% a 6% menos del tiempo que **RMSProp Ac**. Por último, la propuesta **RILBC 5%** sobrepasa por poco (18% adicional) el tiempo empleado por el entrenamiento con todos los datos (**RMSProp Ac**) y esto pone en duda su uso en el escenario de aprendizaje incremental.

En relación con **CIFAR-10-U**, los resultados de **NIL 1%** siguen siendo los mejores en tiempo, ya que sólo tarda un 25% más que el uso de **RMSProp Inc** y demora 22% menos que **RMSProp Ac** (todos los datos de entrenamiento). Por su parte,

NIL 10% y **RILBC 1%** incurren en un sobrecosto menor, un 11% a 12% menos que el empleado por **RMSProp Ac**, usando un 41%-43% más que el usado por **RMSProp Inc**. Finalmente, **RILBC 5%** sobrepasa por mucho (117% adicional) el tiempo empleado por el entrenamiento con todos los datos (**RMSProp Ac**) y no debe ser considerado.

Tabla 10 *Tiempos promedio de entrenamiento (segundos) para cada algoritmo en CIFAR-10-I y CIFAR-10-U. Se proveen los tiempos para cada megalote*

Algoritmo	**CIFAR-10-I**								**CIFAR-10-U**							
	1	**2**	**3**	**4**	**5**	**Total**	**%I**	**%A**	**1**	**2**	**3**	**4**	**5**	**Total**	**%I**	**%A**
RMSProp Inc	318	322	322	322	323	1607	-	-	388	352	242	272	354	1608	-	-
RMSProp Ac	320	450	532	516	643	2461	53%	-	386	518	539	497	638	2578	60%	-
NIL 1%	399	402	405	405	407	2018	**26%**	**-18%**	487	439	304	340	442	2012	**25%**	**-22%**
NIL 10%	407	422	437	449	463	2178	36%	**-11%**	556	501	346	388	505	2296	43%	**-11%**
RILBC 1%	411	425	436	449	459	2180	36%	**-11%**	549	495	341	383	498	2266	41%	**-12%**
RILBC 5%	457	521	579	640	699	2896	80%	18%	845	764	525	589	767	3490	117%	35%
iCaRL 1%	450	460	462	467	478	2318	44%	**-6%**								
iCaRL 10%	489	475	477	483	514	2439	52%	**-1%**								

5.8 RESULTADOS EXPERIMENTALES EN CALTECH 101

Los resultados para **Caltech 101-I** y **Caltech 101-U** se muestran en la **Figura 38**. Aquí se observa que **RMSProp Inc** tiene un pobre rendimiento en **Caltech 101-I**, dado que no supera una exactitud del 50% y, de hecho, se presenta un sobre entrenamiento que ocasiona que pierda exactitud a medida que se agregan nuevos megalotes. Esto se constata al ver su comportamiento en las gráficas de *loss*, con grandes picos en cada cambio de megabatch. **iCaRL** obtiene mejores resultados llegando a superar el 70% en su versión **iCaRL 10%** y en **iCaRL 1%** superando el 62%, logrando aumentar la exactitud de la red a lo largo de los distintos megalotes. Sin embargo, **NIL** y **RILBC** obtienen mejores resultados, tanto en las variedades de 1%, las cuales alcanzan un 64% de exactitud, como en sus variedades de 10% y 5%, las cuales obtienen resultados más altos. **NIL 10%** obtiene el mejor resultado con 78.6%, el cual es un resultado competitivo respecto a **RMSProp Ac**, que alcanza un 85.6%. Aunque en Caltech 101-U no se encuentran diferencias tan grandes en los resultados como en **Caltech 101-I**, también se observa que **RMSProp Inc** es el algoritmo que alcanza la menor exactitud. El Transfer Learning aplicado a esta red hace que muy a pesar de ser un dataset relativamente complejo, se logren excelentes resultados en el proceso de entrenamiento en ambas configuraciones del dataset.

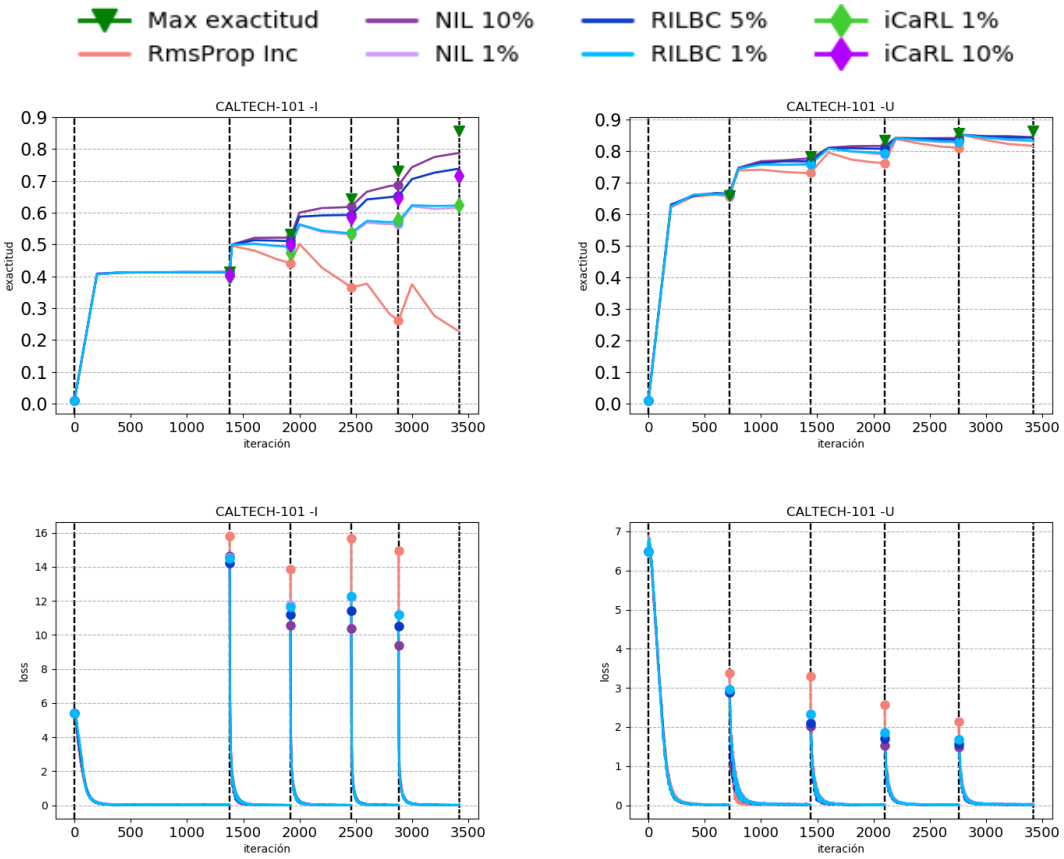

Figura 38 *Resultados de exactitud y pérdida en Caltech 101*

En la **Tabla 11** se puede evidenciar que **iCaRL** tiene tiempos incluso menores que los obtenidos por **RMSProp Inc**, siendo hasta un 42% más rápido en su versión **iCaRL 10%** y 46% en **iCaRL 1%**. Esto únicamente ocurre en este dataset y puede explicarse en parte debido a que la implementación de **iCaRL** se hizo sobre un framework diferente. Así mismo, este hecho también influencia los resultados de tiempo de ejecución de **iCaRL** en los otros tres datasets. Por otro lado, se puede apreciar que en **Caltech 101** todos los algoritmos tardan un menor tiempo que RMSProp Ac, siendo que en **Caltech 101-I**, **NIL 1%**, **NIL 10%**, **RILBC 1%** y **RILBC 5%** sólo tardan entre 37% y 44% más de tiempo que **RMSProp Inc**, mientras que usan entre 32% y 35% menos tiempo que **RMSProp Ac**. En este caso, todos los algoritmos cumplen con el requisito de realizar el entrenamiento en un tiempo menor que el empleado por **RMSProp Ac**, es decir, con todos los datos acumulados.

En **Caltech 101-U** la situación es similar, siendo que **NIL 1%**, **NIL 10%**, y **RILBC 1%** sólo tardan entre 40% y 44% más tiempo que **RMSProp Inc**, y usan entre 19% y 20% menos tiempo que **RMSProp Ac**. Se puede observar en los dos datasets de **Caltech 101**, que la diferencia de tiempo de ejecución entre los algoritmos **NIL** y **RILBC** es menor que en todos los demás, y esto se debe principalmente a que este dataset es el de menor tamaño, por lo que tiene el número más reducido de

iteraciones y de representantes, lo que conlleva a que el número de actualizaciones y extracciones de representantes sea menor. **RILBC 10%** es el que más se demora en relación con **RMSProp Inc**, tardando un 63% de tiempo adicional, pero, también se ejecuta en menos tiempo que **RMSProp Ac** (un 7% de menos tiempo), por lo cual todos los algoritmos son aceptables en un entorno de aprendizaje incremental.

Tabla 11 *Tiempos promedio de entrenamiento (segundos) para cada algoritmo en Caltech 101-I y Caltech 101-MNIST-U. Se proveen los tiempos para cada megalote*

Algoritmo	**Caltech 101-I**								**Caltech 101-U**							
	1	**2**	**3**	**4**	**5**	**Total**	**%I**	**%A**	**1**	**2**	**3**	**4**	**5**	**Total**	**%I**	**%A**
RMSProp Inc	453	171	170	132	177	1103	-	-	227	222	200	212	213	1074	-	-
RMSProp Ac	445	510	388	447	532	2322	111%	-	229	373	325	429	527	1883	75%	-
NIL 1%	627	231	238	182	241	1519	38%	**-35%**	324	314	285	298	297	1518	41%	**-19%**
NIL 10%	624	231	235	183	242	1515	37%	**-35%**	319	312	283	296	295	1505	**40%**	**-20%**
RILBC 1%	633	234	239	185	245	1536	39%	**-34%**	329	319	289	305	303	1545	44%	**-18%**
RILBC 5%	631	240	249	199	268	1587	44%	**-32%**	366	365	331	347	345	1754	63%	**-7%**
iCaRL 1%	238	89	91	73	103	594	**-46%**	**-74%**								
iCaRL 10%	239	93	98	82	130	644	**-42%**	**-72%**								

5.9 ANÁLISIS ESTADÍSTICO

Se realizó un análisis estadístico de los distintos algoritmos en los dos escenarios de experimentación propuestos: todos los datasets incrementales (datasets-I) y todos los dataset desbalanceados (datasets-U), tomando los resultados finales de exactitud en el último megalote de cada dataset. Se aplicó la prueba estadística no paramétrica de Friedman con un valor de significancia apropiado ($p < 0.05$). Posteriormente se realizó el test de Wilcoxon, en el cual los resultados de cada algoritmo se comparan con todos los demás para determinar si existe dominancia de un algoritmo sobre otro. En este test no se encontraron diferencias estadísticamente significativas, y por ello se aplicó el Post hoc de Holm, el cual sirve para el mismo propósito que el test de Wilcoxon y algunos autores lo reconocen como una mejor estrategia de evaluación. Un resumen de estos resultados se presenta en la **Figura 39**.

En el caso de los datasets-I, al observar los resultados de la prueba de Friedman, se encuentra que el ranking de los algoritmos, ordenados de mejor a peor desempeño, es: 1) **RMSProp Ac**, 2) **NIL 10%**, 3) **RILBC 5%**, 4) **NIL 1%**, 5) **RILBC 1%**, 6) **iCaRL 10%**, 7) **iCaRL 1%** y en el último lugar 8) **RMSProp Inc**. En el post hoc de Holm no se encuentra una diferencia estadísticamente significativa entre los resultados de **RMSProp Ac (Máx. Exactitud)** y **NIL** y **RILBC** en ninguna de sus versiones, lo que indica que estos algoritmos son competitivos entre sí. Sin embargo, **RMSProp Ac** sí obtiene resultados que son significativamente mejores

que los de **RMSProp Inc** y los de **iCaRL** con un 95% de significancia. Además, **NIL 10%** domina los resultados de **RMSProp Inc** con el mismo nivel de significancia.

Comparación estadística en los datasets-I

AL	1	2	3	4	5	6	7	8	Fried.
1	\	○		○					7.5
2	●	\					●	●	1
3			\						4.5
4	●			\					2
5					\				5
6						\			3.5
7		○					\		6.5
8		○						\	6

Comparación estadística en los datasets-U

AL	1	2	3	4	5	6	Fried.
1	\	○		○			6
2	●	\	●		●		1
3			\				4
4	●			\			**2**
5		○			\		4.75
6						\	3.25

1 RMSProp Inc, 2 RMSProp Ac, 3 NIL 1%, 4 NIL 10%, 5 RILBC 1%, 6 RILBC 5%, 7 iCaRL 1%, 8 iCaRL 10%

Figura 39 *Resumen de los resultados del test de Friedman y del post hoc de Holm. Los rankings de Friedman se muestran en la columna llamada "Fried." en cada tabla. Para el post hoc de Holm, el símbolo ● significa que el método de la fila mejora al método de la columna, y el símbolo ○ significa que el método de la columna mejora al método de la fila. La diagonal superior implica un nivel de significancia α = 0.9 y la diagonal inferior implica un nivel de significancia α = 0.95.*

En el caso de los datasets-U, la prueba de Friedman permite obtener el siguiente ranking: 1) **RMSProp Ac**, 2) **NIL 10%**, 3) **RILBC 5%**, 4) **NIL 1%**, 5) **RILBC 1%** y en el último lugar 6) **RMSProp Inc**. Se puede apreciar que los primeros 5 puestos de este ranking coinciden con los de los datasets-I. El post hoc de Holm permite ver que **RMSProp Inc** es dominado por **RMSProp Ac** y **NIL 10%** con un 95% de significancia, **RMSProp Ac** domina con un 95% de significancia a **RILBC 1%** y a **NIL 1%** con un 90% de significancia. Holm no permite ver una diferencia estadísticamente significativa entre los resultados de **RMSProp Ac (Máx. Exactitud)** y **NIL 10%** y **RILBC 10%.**

Con estos resultados, se puede afirmar que **NIL 10%** es el algoritmo incremental que mejores resultados presenta en los dos escenarios de prueba, y además sus resultados son comparables con los obtenidos por **RMSProp Ac** (**Máx. Exactitud**). Es preciso anotar que los resultados de Friedman y de los post hoc de Holm obtenidos en cada megalote (2, 3 y 4) son similares a los presentados en esta sección, que corresponden al megalote 5 o último.

5.10 DISCUSIÓN

Al consolidar los resultados experimentales de todos los datasets junto con el análisis estadístico realizado, se puede decir que el algoritmo incremental que obtiene los mejores resultados es **NIL**, seguido de **RILBC**, y de **iCaRL**, encontrándose **RMSProp Inc** (linea base) en último lugar. En el caso de **NIL** como

en el de **RILBC** se encuentra que sus versiones con mayor cantidad de representantes se desempeñan mejor en términos de exactitud, sin embargo, su tiempo de ejecución también aumenta de manera significativa. Por otra parte, **RILBC** tiende a tener un mayor tiempo de entrenamiento que **NIL** debido a que, aunque usa el mismo tamaño de buffer (parámetro *q*), sus operaciones de selección de candidatos y competencia entre representantes requieren más tiempo para ejecutarse. Del mismo modo, el aumento en el número de representantes afecta en mayor medida al tiempo de ejecución de **RILBC** que al tiempo de ejecución de **NIL**. También se destaca que las pruebas sobre **Caltech 101** indican que los algoritmos **NIL** y **RILBC** presentan resultados satisfactorios en escenarios de Transfer Learning. Por otro lado, **RMSProp Inc** aunque es el algoritmo más rápido, sus resultados fueron los peores debido al sobre entrenamiento que presenta, sobre todo en los datasets-I.

Aunque **iCaRL** obtuvo excelentes resultados con el uso de una red **ResNet** y con un protocolo de pruebas específico [34], en las pruebas de este trabajo no se encontró una diferencia significativa de desempeño entre sus versiones 1% y 10% excepto en el dataset **Caltech 101**. En general sus resultados están muy por debajo de los obtenidos por los dos algoritmos (**NIL** y **RILBC**) instanciados en el framework, **CRIF**, ya que su exactitud es inferior entre un 20% y 40% según el caso.

Los resultados indican que el marco conceptual propuesto en este trabajo funciona de manera satisfactoria para evitar el olvido catastrófico provocado por el desbalanceo de los datos o la incorporación de datos pertenecientes a clases nunca vistas, obteniendo resultados cercanos a un entrenamiento completo con toda la información disponible, con la ventaja de solo almacenar una porción muy pequeña de los datos (prototipos) y reduciendo el tiempo de entrenamiento de manera significativa respecto a un reentrenamiento completo como el presentado en **RMSProp Ac**.

CAPÍTULO 6

6 CONCLUSIONES Y TRABAJO FUTURO

Las conclusiones de este trabajo son:

1. En relación con el primer objetivo específico, en esta investigación además de lo originalmente planteado, se diseñó, implementó y probó un marco de trabajo para aprendizaje incremental en Python sobre TensorFlow de acceso libre y código abierto denominado *Deep Incremental Learning Framework* (DILF). Este framework facilita el desarrollo, evaluación y comparación de algoritmos de entrenamiento incremental en Deep Learning gracias a sus módulos débilmente acoplados y altamente cohesivos, a saber: ETL, Redes Neuronales, Entrenamiento y Experimentación.

2. En relación con el segundo objetivo específico, en esta investigación además de lo planteado, se definió un Marco conceptual para el aprendizaje incremental basado en Rehearsal denominado Competitive Representatives Incremental Framework (CRIF) con cuatro componentes claves, a saber: una lista de representantes (prototipos por clase), un componente de selección de candidatos, un componente de actualización de representantes y un componente de regularización de muestras. Estos componentes trabajan sinérgicamente en conjunto con el algoritmo de entrenamiento (RMSProp u otro). También se realizó la personalización de los tres componentes de CRIF en dos algoritmos, uno denominado *Naive Incremental Learning* (NIL) y otro denominado *Representatives Incremental Learning with BvSB and Crowding Distance* (RILBC) que usa la estrategia BvSB y la distancia de crowding.

3. La propuesta NIL es una instanciación de CRIF que escoge candidatos para ser usados en el proceso de Rehearsal mediante una selección completamente aleatoria de muestras, y de manera similar, su componente de actualización de representantes realiza una selección aleatoria de los representantes de cada clase que se encuentran almacenados en una lista de representantes. Así mismo, en el componente de selección aleatoria de muestras representantes, las cuales se usan para entrenar la Red Neuronal Artificial (RNA) junto a nuevas muestras, y, además, asigna un peso a cada muestra para penalizar en mayor medida la incorrecta clasificación de las muestras representantes por parte de la RNA durante el entrenamiento.

4. La propuesta RILBC es una instanciación de CRIF que escoge candidatos para ser usados en el proceso de Rehearsal mediante una selección de las muestras que se encuentran más cercanas a la mediana según el cálculo de la métrica Best vs Second Best (BvSB). Además, su componente de actualización de representantes selecciona las muestras que tienen el mayor valor en la métrica conocida como Crowding Distance, que se calcula para cada clase en la lista de representantes. Por otra parte, RILBC adopta el mismo componente de regularización de muestras que el usado en NIL.

5. En relación con el tercer objetivo específico, se puede concluir que **NIL** obtiene los mejores resultados en los experimentos con los cuatro datasets utilizados. NIL y RILBC, las dos instanciaciones propuestas de CRIF, alcanzan una mayor exactitud que iCaRL y RMSProp Inc, encontrándose que este último no logra buenos resultados en un ambiente con clases desbalanceadas y mucho menos con clases completamente incrementales. Las versiones con mayor número de representantes de NIL (10%) y RILBC (5%) tienen un mejor desempeño en términos de exactitud, pero su tiempo de ejecución aumenta significativamente, hecho que afecta en mayor medida a RILBC. Además, los algoritmos NIL y RILBC presentan resultados satisfactorios en escenarios de Transfer Learning.

6. El marco conceptual propuesto en este trabajo, **CRIF**, en especial las dos instanciaciones de este (NIL y RILBC) funcionan de manera satisfactoria en escenarios de clases incrementales y desbalanceadas, obteniendo resultados cercanos a un entrenamiento con toda la información, con la ventaja de solo almacenar una porción pequeña de los datos (prototipos) y reduciendo el tiempo de entrenamiento significativamente respecto a un reentrenamiento completo como el presentado en **RMSProp Ac.** Estos resultados están soportados por el test estadístico no paramétrico de Friedman y el post hoc de Holm con un 90-95% de significancia.

7. Debido a los problemas encontrados durante el proceso de selección y experimentación con el algoritmo del estado del arte (iCaRL), referentes a su dificultad para ser entendido, adaptado y extendido, se encuentra imperativo que los investigadores del área adopten adecuadas prácticas de diseño y programación, para que las nuevas propuestas de aprendizaje incremental en Deep Learning sean fáciles de extender, mejorar y de replicar. Esto ayudaría a que en el área se cuente con una mayor cantidad y calidad de nuevas propuestas. Lo anterior, motivó el desarrollo de DILF y se espera que este sea un catalizador de nuevas y disruptivas propuestas.

8. En esta investigación se usó el Patrón de Investigación Iterativo (PII) como base para el desarrollo del trabajo conceptual (DILF y CRIF) y de los

algoritmos propuestos (NIL y RILBC). Con esta metodología se ejecutaron múltiples ciclos de observación, identificación, desarrollo y pruebas. Las etapas de desarrollo y pruebas se ejecutaron en paralelo, lo cual permitió un ciclo de retroalimentación rápida. Adicionalmente, durante este proceso se crearon artículos científicos para ser enviados a revistas internacionales reconocidas, lo cual fue sumamente útil para guiar el proceso de divulgación científica de este trabajo, ya que permitió identificar aspectos clave relacionados con la reproducibilidad de los resultados cuya divulgación no había sido considerada en un inicio.

Como trabajo futuro se considera importante realizar lo siguiente:

1. Realizar un análisis de sensibilidad exhaustivo de los parámetros de **NIL** y **RILBC** (tamaño del buffer de candidatos, número de candidatos y número de representantes por clase), realizando pruebas para medir los resultados de tiempo y exactitud sobre múltiples datasets, incluyendo datasets de gran tamaño tanto en número de muestras como en número de clases, como es el caso de ImageNet.

2. Realizar una extensión de **NIL** y **RILBC** que los convierta en *Algoritmos de entrenamiento adaptativos*, es decir, con parámetros que se calculen automáticamente durante el entrenamiento de acuerdo con los resultados que se van obteniendo, evitando que tengan que ser fijados previamente por un humano.

3. Modificar **CRIF** para que pueda ser usado con diferentes arquitecturas de Redes Neuronales, como las Redes Recurrentes o las Redes de Creencia Profunda, de modo que los mecanismos de selección y competencia de representantes hagan uso de la salida de estas redes. Adicionalmente, se considera relevante probar el uso de optimizadores diferentes a RMSProp, tales como Adam o Momentum para comprobar la adaptabilidad de **CRIF** a múltiples optimizadores.

4. Modificar en **CRIF** el mecanismo de selección de representantes por clases para que pueda ser aplicado a escenarios donde no todas las muestras se encuentren etiquetadas (i.e. no se sabe con certeza a qué clase pertenece cada muestra), como en el caso del aprendizaje no supervisado (ninguna muestra etiquetada) y semi-supervisado (algunas muestras etiquetadas).

5. Buscar y seleccionar otras métricas de cálculo de representatividad e incertidumbre, diferentes a las usadas en **NIL** y **RILBC**, de modo que se puedan crear nuevas instanciaciones de **CRIF**. Algunas métricas que pueden ser exploradas son: entropía, probabilidad de rechazo y la realización de clústeres de representantes.

6. Realizar una segunda versión de DILF que incorpore características adicionales que no se encuentran en su versión inicial, tales como: i) soporte nativo para *Knowledge Distillation* mediante una extensión de los módulos de Redes y Entrenamiento; ii) extensión de la clase Tester para evaluar algoritmos diferentes a RNAs; iii) soporte de datasets adicionales por medio del módulo ETL y iv) agregación de algoritmos de entrenamiento adicionales en el módulo de Entrenamiento.

CAPÍTULO 7

7 BIBLIOGRAFÍA

[1] N. J. Nilsson, “Introduction to Machine Learning.”

[2] S. J. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 3rd ed. Prentice Hall, 2009.

[3] Alex Smola and and S. V. N. Vishwanathan, “Introduction to Machine Learning,” *Press synd icate o f un ivers ity o f cambr idge*, 2008.

[4] R. Ranawana and V. Palade, “Optimized Precision - A New Measure for Classifier Performance Evaluation,” in *2006 IEEE International Conference on Evolutionary Computation*, 2006, pp. 2254–2261.

[5] M. Schlesinger and V. Hlavác, “Supervised and unsupervised learning.,” *Artificial Intelligence*, no. April. 2011.

[6] Y. LeCun, “A theoretical framework for Back-Propagation,” *Proceedings of the 1988 Connectionist Models Summer School*. pp. 21–28, 1988.

[7] J. Schmidhuber, “Deep Learning in Neural Networks: An Overview,” *Neural Networks*, vol. 61, pp. 85–117, 2015.

[8] G. E. Hinton, N. Srivastava, and K. Swersky, “Lecture 6.5- Divide the gradient by a running average of its recent magnitude,” *COURSERA: Neural Networks for Machine Learning*. p. 31, 2012.

[9] M. C. Mukkamala and M. Hein, “Variants of RMSProp and Adagrad with Logarithmic Regret Bounds,” *ICML*, 2017.

[10] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. Alemi, “Inception-v4, Inception-ResNet and the Impact of Residual Connections on Learning,” pp. 4278–4284, 2016.

[11] A. Karpathy and L. Fei-Fei, “Deep Visual-Semantic Alignments for Generating Image Descriptions,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 39, no. 4, pp. 664–676, 2017.

[12] T. Schaul, I. Antonoglou, and D. Silver, “Unit Tests for Stochastic Optimization,” *ICLR*, no. December 2013, 2013.

[13] A. Gepperth and B. Hammer, “Incremental learning algorithms and applications,” in *European Symposium on Artificial Neural Networks (ESANN)*, 2016.

[14] M. Režnáková, L. Tencer, and M. Cheriet, “Incremental Similarity for real-time on-line incremental learning systems,” *Pattern Recognit. Lett.*, vol. 74, pp. 61–67, Apr. 2016.

[15] K. Shakib Sarwar, Syed; Ankit, Aayush; Roy, “Incremental Learning in Deep Convolutional Neural Networks Using Partial Network Sharing,” 2017.

[16] S. Grossberg, “Nonlinear neural networks: Principles, mechanisms, and

architectures," *Neural Networks*, vol. 1, no. 1. Pergamon, pp. 17–61, 01-Jan-1988.

[17] B. Ren, H. Wang, J. Li, and H. Gao, "Life-long learning based on dynamic combination model," *Appl. Soft Comput.*, vol. 56, pp. 398–404, Jul. 2017.

[18] S. Han, Z. Meng, A. S. Khan, and Y. Tong, "Incremental boosting convolutional neural network for facial action unit recognition," in *Advances in Neural Information Processing Systems*, 2016, pp. 109–117.

[19] H. Guan, X. Xue, and A. Zhiyong, "Online video tracking using collaborative convolutional networks," in *Proceedings - IEEE International Conference on Multimedia and Expo*, 2016, vol. 2016-Augus.

[20] R. Polikar, L. Udpa, S. S Udpa, S. Member, and V. Honavar, *Learn++: An Incremental Learning Algorithm for Supervised Neural Networks*. 2002.

[21] A. Baraka, G. Panoutsos, and S. Cater, "Perpetual Learning Framework based on Type-2 Fuzzy Logic System for a Complex Manufacturing Process," *IFAC-PapersOnLine*, vol. 49, no. 20, pp. 143–148, 2016.

[22] A. Hebboul, F. Hachouf, and A. Boulemnadjel, "A new incremental neural network for simultaneous clustering and classification," *Neurocomputing*, vol. 169, pp. 89–99, Dec. 2015.

[23] H. Xu, Y. Xing, F. Shen, and J. Zhao, "An online incremental learning algorithm for time series," in *Proceedings of the International Joint Conference on Neural Networks*, 2015, vol. 2015-Septe.

[24] B. Settles, "Active Learning Literature Survey," *Mach. Learn.*, vol. 15, no. 2, pp. 201–221, 2010.

[25] P. K. Rhee, E. Erdenee, S. D. Kyun, M. U. Ahmed, and S. Jin, "Active and Semi-Supervised Learning for Object Detection with Imperfect Data," *Cogn. Syst. Res.*, vol. 45, pp. 109–123, 2017.

[26] M. Pratama, J. Lu, S. Anavatti, E. Lughofer, and C. P. Lim, "An incremental meta-cognitive-based scaffolding fuzzy neural network," *Neurocomputing*, vol. 171, pp. 89–105, Jan. 2016.

[27] S. Huang, H. Xu, and X. Xia, "Active deep belief networks for ship recognition based on BvSB," *Optik (Stuttg).*, vol. 127, no. 24, pp. 11688–11697, Dec. 2016.

[28] M. Ullrich, H. Ali, M. Durner, Z.-C. Marton, and R. Triebel, "Selecting CNN features for online learning of 3D objects," in *IEEE International Conference on Intelligent Robots and Systems*, 2017, vol. 2017-Septe, pp. 5086–5091.

[29] P. Liu, H. Zhang, and K. B. Eom, "Active Deep Learning for Classification of Hyperspectral Images," *IEEE J. Sel. Top. Appl. Earth Obs. Remote Sens.*, vol. 10, no. 2, pp. 712–724, Feb. 2017.

[30] C. Käding, E. Rodner, A. Freytag, and J. Denzler, "Watch, Ask, Learn, and Improve: A lifelong learning cycle for visual recognition," in *ESANN 2016 - 24th European Symposium on Artificial Neural Networks*, 2016, pp. 381–386.

[31] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, "SMOTE: Synthetic minority over-sampling technique," *J. Artif. Intell. Res.*, vol. 16, pp. 321–357, 2002.

[32] G. Ditzler and R. Polikar, "Incremental Learning of Concept Drift from Streaming Imbalanced Data," *IEEE Trans. Knowl. Data Eng.*, vol. 25, no. 10, pp. 2283–2301, 2013.

[33] L. Ma and S. FAN, “CURE-SMOTE algorithm and hybrid algorithm for feature selection and parameter optimization based on random forests,” *BMC Bioinformatics*, vol. 18, 2017.
[34] S.-A. Rebuffi, A. Kolesnikov, and C. H. Lampert, “iCaRL: Incremental Classifier and Representation Learning,” *CoRR*, vol. abs/1611.0, 2016.
[35] A. Barushka and P. Hajek, “Spam filtering using integrated distribution-based balancing approach and regularized deep neural networks,” *Appl. Intell.*, vol. 48, no. 10, pp. 3538–3556, 2018.
[36] S. Pang, L. Zhu, G. Chen, A. Sarrafzadeh, T. Ban, and D. Inoue, “Dynamic class imbalance learning for incremental LPSVM,” *Neural Networks*, vol. 44, pp. 87–100, 2013.
[37] W. W. Y. Ng, J. Zhang, C. S. Lai, W. Pedrycz, L. L. Lai, and X. Wang, “Cost-Sensitive Weighting and Imbalance-Reversed Bagging for Streaming Imbalanced and Concept Drifting in Electricity Pricing Classification,” *IEEE Trans. Ind. Informatics*, 2018.
[38] Q. J. Zhang and K. C. Gupta, “Neural Network Structures,” *Neural Networks RF Microw. Des.*, pp. 61–103, 2000.
[39] C. Bishop and C. M. Bishop, “Neural networks for pattern recognition,” 1995.
[40] Y. LeCun, P. Haffner, L. Bottou, and Y. Bengio, “Learning, Object recognition with gradient-based,” *Shape, contour Group. Comput. Vis.*, pp. 319–345, 1999.
[41] M. Matsugu, K. Mori, Y. Mitari, and Y. Kaneda, “Subject independent facial expression recognition with robust face detection using a convolutional neural network.,” *Neural Netw.*, vol. 16, no. 5–6, pp. 555–9, 2003.
[42] fei fei, “Convolutional Neural Networks for Visual Recognition,” 2015. [Online]. Available: http://cs231n.github.io/convolutional-networks/.
[43] Y.-H. Chang, Z.-X. Peng, and L.-D. Jeng, “Digital image processing,” vol. 9, no. 7, pp. 1811–1817, 2015.
[44] Mate Labs, “How these researchers tried something unconventional to come out with a smaller yet better Image Recognition,” 2017. [Online]. Available: https://medium.com/@matelabs_ai/how-these-researchers-tried-something-unconventional-to-came-out-with-a-smaller-yet-better-image-544327f30e72?ct=t(Active_Users_Reachout3_8_2017)#.mbc5k0mo8.
[45] R. Ajemian, A. D’Ausilio, H. Moorman, and E. Bizzi, “A theory for how sensorimotor skills are learned and retained in noisy and nonstationary neural circuits,” *Proc. Natl. Acad. Sci.*, vol. 110, no. 52, pp. E5078–E5087, 2013.
[46] L. Yang, S. Hanneke, and J. Carbonell, “A theory of transfer learning with applications to active learning,” *Mach. Learn.*, vol. 90, no. 2, pp. 161–189, 2013.
[47] S. Ruder, “Transfer Learning - Machine Learning’s Next Frontier,” 2017. [Online]. Available: http://ruder.io/transfer-learning/. [Accessed: 21-Jan-2019].
[48] Y. Zhang, D. Zhao, J. Sun, G. Zou, and W. Li, “Adaptive Convolutional Neural Network and Its Application in Face Recognition,” *Neural Process. Lett.*, vol. 43, no. 2, pp. 389–399, 2016.
[49] A. Besedin, P. Blanchart, M. Crucianu, and M. Ferecatu, “Evolutive deep

models for online learning on data streams with no storage," in *CEUR Workshop Proceedings*, 2017, vol. 1958, pp. 1–12.
[50] F. Chollet and others, "Keras." 2015.
[51] C. Narvaez, D. Muñoz, and C. Cobos, "DILF: Deep Incremental Learning Framework over TensorFlow," *SoftwareX*, vol. In evaluat, 2019.
[52] Ethereon, "Caffe-Tensorflow." GitHub, 2016.
[53] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proc. IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998.
[54] Y. LeCun and C. Cortes, "{MNIST} handwritten digit database," 2010.
[55] TensorFlow, "Advanced Convolutional Neural Networks," *TensorFlow Documentation*, 2018. [Online]. Available: https://www.tensorflow.org/tutorials/images/deep_cnn.
[56] A. Krizhevsky, V. Nair, and G. Hinton, "CIFAR-10 and CIFAR-100 datasets," *https://www.cs.toronto.edu/~kriz/cifar.html*, 2009. .
[57] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems 25*, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2012, pp. 1097–1105.
[58] R. F. and P. P. L. Fei-Fei, "Caltech 101."
[59] M. Maynard-Reid, "Fashion-MNIST with tf.Keras," 2018. [Online]. Available: https://medium.com/tensorflow/hello-deep-learning-fashion-mnist-with-keras-50fcff8cd74a.
[60] H. Xiao, K. Rasul, and R. Vollgraf, "Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms," *CoRR*, vol. abs/1708.0, 2017.
[61] K. Pratt, "Design Patterns for Research Methods: Iterative Field Research," *AAAI Spring Symp. Exp. Des. Real …*, no. 1994, 2009.
[62] L. Joseph, "Incremental Rehearsal: A Flashcard Drill Technique for Increasing Retention of Reading Words," *Read. Teach. - READ TEACH*, vol. 59, pp. 803–807, 2006.
[63] T. Hayes, N. Cahill, and C. Kanan, "Memory Efficient Experience Replay for Streaming Learning." 2018.
[64] S. Scardapane and D. Wang, "Randomness in Neural Networks: An Overview," *Wiley Interdiscip. Rev. Data Min. Knowl. Discov.*, vol. 7, 2017.
[65] a.J. Joshi, F. Porikli, N. Papanikolopoulos, A. J. Joshi, F. Porikli, and N. Papanikolopoulos, "Multi-class active learning for image classification," *2009 IEEE Conf. Comput. Vis. Pattern Recognit.*, pp. 2372–2379, 2009.
[66] S. S. Garud, I. A. Karimi, and M. Kraft, "Smart Adaptive Sampling for Surrogate Modelling," in *26th European Symposium on Computer Aided Process Engineering*, vol. 38, Z. Kravanja and M. Bogataj, Eds. Elsevier, 2016, pp. 631–636.
[67] S. Luke, *Essentials of Metaheuristics, second edition*. Lulu, 2013.
[68] R. Fergus and P. Perona, "Learning Generative Visual Models from Few Training Examples: An Incremental Bayesian Approach Tested on 101 Object Categories," in *2004 Conference on Computer Vision and Pattern Recognition Workshop*, 2004, p. 178.

[69] A. Krizhevsky, “Learning Multiple Layers of Features from Tiny Images,” *Ttechnical Rreport, Dep. Comput. Sci. Univ. Toronto*, pp. 1–60, 2009.
[70] M. Abadi *et al.*, “TensorFlow : A System for Large-Scale Machine Learning,” *Proc 12th USENIX Conf. Oper. Syst. Des. Implement.*, pp. 272–283, 2016.
[71] D. Lopez-Paz and M. Ranzato, “Gradient Episodic Memory for Continuum Learning,” *CoRR*, vol. abs/1706.0, 2017.
[72] A. Paszke *et al.*, “Automatic Differentiation in PyTorch,” *NIPS 2017 Work. Autodiff Decis. Progr. Chairs*, vol. 22, no. 1, pp. 2–8, 2017.