Implementación de un Algoritmo de Aprendizaje para la Arquitectura *Deep-Networks*

Fredy Antonio Salazar-Vasquez

Trabajo de Grado presentado como requisito para optar el título de

Ingeniero Electrónico

Facultad de Ingeniería
Programa de Pregrado en Ingeniería Electrónica
Santiago de Cali
2017

Carta de Aprobación

Nota de Aprobación

_____

_____

_____

_____

N.N., Ph.D.

_____

N.N., Ph.D.

Aceptado por

_____

Eduardo F. Caicedo - Supervisor Trabajo de Grado

_____

Wilfredo Alfonso - Supervisor Trabajo de Grado

Santiago de Cali – Julio, 2017

Resumen

Este trabajo presenta una revisión histórica de la evolución de las redes neuronales artificiales, desde el uso de modelos mono capa hasta el origen del *Deep Learning*. Se explica el motivo y la necesidad de adoptar estas tecnologías y se describe diferentes arquitecturas de *Deep Networks*. Se presenta el desarrollo de una aplicación que permite el diseño de *Deep Networks* y su entrenamiento utilizando algoritmos de *Deep Learning*. Se estudia la metodología propuesta para el desarrollo *Deep Networks* y se utiliza la tarjeta NVidia Jetson TK1 para evaluar el desempeño de la aplicación desarrollada. Pese a las limitaciones que posee la tarjeta, se verifica el funcionamiento de las arquitecturas propuestas y los algoritmos de aprendizaje sobre 2 bases de datos de mediana complejidad.

Palabras claves: Caffe, *Deep Learning*, *Deep Networks*, *Machine Learning*, NVidia Jetson TK1, Redes Neuronales Artificiales.

Abstract

This work presents a historical review of artificial neural networks evolution, from the use of single-layer models to the origin of Deep Learning. It explains the reason and need to adopt these technologies and describes different architectures of Deep Networks. The development of an application that allows the design of Deep Networks and its training using Deep Learning algorithms is presented. The methodology proposed for the Deep Networks development is studied and the NVidia Jetson TK1 card is used to evaluate the performance of the developed application. Despite the limitations of the card, the proposed architectures and learning algorithms are tested on 2 databases of medium complexity.

Keywords: Artificial Neural Network, Caffe, Deep Learning, Deep Networks, Machine Learning, NVidia Jetson TK1.

Dedicatoria

A mi hermano Antonio Jose, por estar conmigo, ser mi mano derecha y mi principal motivación en seguir adelante.

A mis padres Fredy y Giomara, por su apoyo incondicional y por ser el pilar fundamental en todo lo que soy.

A Katheryn Ortiz, por su cariño, motivación y acompañamiento a lo largo de este proceso.

A mis maestros Eduardo y Wilfredo, por apoyo y motivación para la culminación de mis estudios profesionales y para la elaboración de esta tesis.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

Aunque hoy en día, el uso de una Red Neuronal Artificial (ANN, por sus siglas en inglés) simple podría no significar mucho desde el ámbito académico, se han presentado casos de éxito que hacen que perdure en el tiempo, como los sistemas de reconocimiento de voz y de caracteres manuscritos; máxime, si se tiene en cuenta la diversidad de aplicaciones que requieren del uso de estas tecnologías software para validar, aproximar o clasificar información. Desde su aparición en 1958, las ANN se han convertido en una herramienta computacional capaz de resolver problemas principalmente no-lineales con gran destreza (Caicedo y Lopez, 2010); sin embargo, las arquitecturas en una ANN parecían estancarse en las tres ramas principalmente: perceptrón multi-capa (MLP), las arquitecturas basadas en aprendizaje no-supervisado como las redes asociativas, los mapas auto-organizados de Kohonen (SOM) y el gas neuronal, y los sistemas híbridos como las redes de base radial. En el año 2004, reaparece una arquitectura de ANN conocida como *Deep Network* que hasta entonces, debido a sus características y sus algoritmos de aprendizaje, no había sido implementada. El origen de esta arquitectura surge durante la década de los 80's, cuando aún se desarrollaban algoritmos de aprendizaje *back-propagation*, pero dada la poca capacidad de computo en cuanto a recursos de memoria y tiempos de procesamiento, con este único algoritmo de aprendizaje

basado en gradiente, presentaron resultados desfavorables. Esto conllevó a la pérdida de interés de la comunidad científica en el desarrollo de técnicas de inteligencia artificial en el área de las *Deep Networks* por casi tres décadas (Deng, 2011). Además, en los 90's otras técnicas de aprendizaje automático (*Machine Learning*, ML) habían alcanzado mayores avances que superaban las capacidades de cómputo, implementación y desempeño de las ANN, lo cual significó un mayor desinterés en esta herramienta.

Hasta entonces, muchos de los esfuerzos se habían aunado en mejorar el desempeño de las redes y el aprendizaje no-supervisado, el cual tomó un nuevo rumbo que revolucionó y despertó el interés de la comunidad científica al presentar resultados satisfactorios para aplicaciones en minería de datos. Mas adelante el uso de estrategias de entrenamiento basadas en este aprendizaje no-supervisado junto con el aprendizaje supervisado se convirtio en una de las bases fundamentales de funcionamiento de los algoritmos en arquitecturas *Deep Networks*, mejor conocido como, *Deep Learning* (Wang y Raj, 2015).

En 2004, Geoffry Hinton apoyado por el Instituto Canadiense de Investigación Avanzada (CIFAR, por sus siglas en inglés) y el respaldo de Yann Le Cun y Yoshua Bengio, fundó el grupo "*Neural Computation and Adaptive Perception* - NCAP", el cual incluyó investigadores de diversas ramas: ingenieros, neuro-científicos, biólogos, físicos y psicólogos. Este grupo estaba encargado de crear y desarrollar sistemas computacionales capaces de imitar la inteligencia orgánica. Para entonces, ya se contaba con la capacidad de cómputo para desarrollar aquellas ideas y retomar las arquitecturas *Deep Networks*. Así, nuevos desarrollos de algoritmos *Deep Learning* para las arquitecturas *Deep Networks* se han probado de manera satisfactoria con bases de datos de gran tamaño (LeCun, Bengio, y Hinton, 2015).

Dado el crecimiento e impacto que han tenido estos nuevos algoritmos de aprendizaje, gigantes de la WEB y del mundo de la informática como Google, Facebook, Apple y Microsoft, han optado por promover nuevos proyectos innovadores en esta línea de desarrollo. Por ejemplo, Andrew Ng, profesor de Stanford y miembro del NCAP, fundó el proyecto *Google*

Brain de Google orientado al reconocimiento de comandos de voz para teléfonos Android y al etiquetado de imágenes de la red social Google Plus. Microsoft incorporó técnicas de *Deep Learning* en su propio sistema de reconocimiento de voz. Facebook con ayuda de Yann LeCun, trabaja con esta tecnología para el etiquetado de anuncios, objetos y rostros en fotos y vídeos (Hernandez, 2014).

Actualmente existen diferentes *frameworks* que han sido diseñados específicamente para el desarrollo de *Deep Networks* y la implementación de algoritmos de *Deep Learning*. Estos *frameworks* están desarrollados en diferentes lenguajes y entornos de programación, como lo son C++, Python, Java, entre otros, e interactúan con otras plataformas que permiten mejorar notoriamente su desempeño, como lo es el caso de las librerías de CUDA, las cuales permiten la utilización de GPUs para realizar el entrenamiento de las redes, además de contar con una librería diseñada específicamente para el desarrollo de *Deep Networks*, esta librería es llamada CuDNN (del inglés *Cuda Deep Neural Networks*).

Hace 30 años, el problema que enfrentaban las sociedades de la información se reflejaba en la ausencia de datos para ser analizados y no se contaba con capacidad de cómputo para procesarla. Con el avance de la tecnología y de los mismos sistemas de información, además de los avances en las ciencias de la computación, Internet y las redes sociales, se cuenta con un gran volumen de datos que requieren ser procesados. Esta información es relevante para compañías que crecen en los entornos competitivos o aquellos interesados en extraer el conocimiento conforme al comportamiento de las señales en los procesos industriales (Gonzalez y Sierra, 2000). Todo esto apunta hacia la minería de datos, donde se busca procesar información y extraer las características fundamentales que faciliten la toma de decisiones, tal que se pueda mejorar o activar un proceso.

Una alternativa para el procesamiento y extracción de conocimiento de la información la ofrece la inteligencia artificial a través de las ANN y sus nuevas arquitecturas; sin embargo, el procesamiento de grandes volúmenes de datos solo se ha hecho posible con la aparición

de nuevas tecnologías y sistemas de computación más robustos y accesibles a las comunidades académicas (Haykin, 1999). Los algoritmos *Deep Learning* son una herramienta muy importante en la actualidad en cuanto al desarrollo de la inteligencia artificial, teniendo una gran variedad de campos de acción, como lo son el reconocimiento de voz o sonidos, reconocimiento de rostros y objetos, *clustering* y análisis de bases de datos complejas en cuanto al número de características y la cantidad de información.

En el grupo de investigación de Percepción y Sistemas Inteligentes de la Universidad del Valle, se trabaja la línea de investigación de inteligencia computacional, donde se han desarrollado proyectos que involucran aprendizaje automático, ANN, problemas de optimización, problemas de procesamiento de señales e imágenes, entre otros. Aunque las ANN han sido aplicadas con éxito en problemas de identificación, clasificación y predicción usando arquitecturas MLP (*Multilayer Perceptron*) sencillas (es decir, una capa oculta con menos de 30 neuronas), sus capacidades de cómputo se ven diezmadas al incrementar el número de neuronas y las capas ocultas.Parte de este problema se debe a que los algoritmos de aprendizaje en su mayoría se basan en métodos de gradiente, los cuales a través de procesos iterativos realizan una búsqueda exhaustiva de la solución mas favorable para un proceso de optimización, con la desventaja de que puede quedar estancado en un mínimo local y no llegar a la mejor solución. Las arquitecturas *Deep Networks* hacen uso de un número considerable de neuronas en las capas ocultas, y usan un esquema de aprendizaje que están basados en el mismo principio que los métodos de gradiente descendente, con la diferencia de que estos realizan la búsqueda de soluciones de manera estocástica procesando entradas aleatorias para no quedar estancados en mínimos locales y no emplear procesamiento innecesario. Los algoritmos *Deep Learning* son una propuesta para el entrenamiento de estas redes. Hasta el momento las arquitecturas *Deep Networks* y el uso de los algoritmos de *Deep Learning* aún no han sido abordados por el grupo de investigación y no cuentan con un *framework* donde se pueda verificar y hacer uso de las *Deep Networks* en problemas de minería de datos, identificación y predicción de alta complejidad.

Por lo tanto, se presenta la siguiente pregunta de investigación:

¿Es posible implementar un algoritmo de aprendizaje *Deep Learning* en una arquitectura MLP compleja, es decir, con un alto número de capas ocultas y neuronas, tal que se apropie el conocimiento de las arquitecturas *Deep Network* en el grupo de investigación PSI y al mismo tiempo se resuelva eficientemente problemas de alta complejidad?

1.1. Objetivos

Con este trabajo de grado, el grupo de investigación Percepción y Sistemas Inteligentes, PSI, busca estudiar e interiorizar estas herramientas de procesamiento especializado, tal que puedan ser aplicables a problemas en minería de datos y proyectos que requieran una fuerte abstracción del conocimiento para resolver problemas de clasificación, interpretación o interpolación de los datos de alta complejidad. Además, este proyecto de investigación permitirá conocer la pertinencia y utilidad, no solo en los campos de la ingeniería, también en áreas del conocimiento como la economía, la estadística, las ciencias sociales, la agricultura y en aplicaciones médicas. Para ello se plantean los siguientes objetivos:

1.1.1. Objetivo General

Implementar un algoritmo de aprendizaje de *Deep Learning* aplicado al entrenamiento de una red neuronal tipo *Deep Network*.

1.1.2. Objetivos Específicos

- Caracterizar los algoritmos de aprendizaje *Deep Learning* en las arquitecturas *Deep Network*.
- Proponer una aplicación software que permita simular el proceso de aprendizaje usando un algoritmo de *Deep Learning*.
- Evaluar el desempeño de la *Deep Network* a través de su aplicación en al menos dos problemas prácticos reconocimiento y clasificación de patrones.

1.2. Bosquejo del Libro

En el primer capítulo de este libro se tratará la problemática que motiva la realización de este trabajo, su justificación y los objetivos planteados para dar solución a la misma. En el segundo capítulo se estudiará a fondo los diferentes conceptos y teorías que permitirán al lector tener una mejor comprensión de la problemática, iniciando desde los modelos más viejos de redes neuronales hasta los modelos de las *Deep Networks* y también se estudiará las diferentes tecnologías que existen para el desarrollo de estas. En el tercer capítulo se presenta la metodología utilizada para el desarrollo de la aplicación, y la explicación de los flujogramas de las arquitecturas y algoritmos utilizados. El cuarto y último capítulo se presentará los casos de aplicación, los resultados obtenidos, la comparación de los resultados y las conclusiones del presente trabajo.

Capítulo 2

Marco Teórico

2.1. Redes Neuronales Artificiales – ANN

Las redes neuronales artificiales (ANN, por sus siglas en inglés) surgen como una propuesta a la solución de problemas de clasificación a partir de imitar el funcionamiento de las neuronas del cerebro humano. Las ANN se basan en un sistema de interconexión de neuronas que transmiten una salida a partir de los estímulos percibidos en su entorno de entrada; este proceso es denominado sinapsis y ocurre en las neuronas biológicas (Anderson, 2007).

Los primeros modelos de redes neuronales datan de 1943 por los neurólogos Warren McCulloch y Walter Pitts, cuando estos intentaron explicar el funcionamiento del cerebro humano por medio de una red de células conectadas entre sí. Años más tarde, en 1949, Donald Hebb desarrolló sus ideas sobre el aprendizaje neuronal, su propuesta tenía que ver con la conductividad de la sinapsis, es decir, con las conexiones entre neuronas. En 1957, Frank Rosenblat desarrolló el perceptrón, la cual es la red neuronal más antigua; Este modelo era capaz de generalizar y hoy en día se utiliza para aplicaciones como identificadores de pa-

trones, sin embargo, este modelo presentó limitaciones ante la presencia de problemas con no linealidades, como por ejemplo resolver el problema de la función lógica XOR. En 1960, Bernard Widrow y Marcian Hoff desarrollaron el ADALINE (*ADAptative LInear Elements*), Este modelo se ha aplicado durante varias décadas y fue la primera red neuronal con uso industrial real. En el año 1986, hubo un renacimiento de las redes neuronales cuando David Rumelhart y G. Hinton redescubrieron el algoritmo de aprendizaje *backpropagation* (BP), y a partir de este momento la redes neuronales obtuvieron un nuevo impulso en cuanto al desarrollo de modelos y nuevas aplicaciones (Ruiz y Basualdo, 2004).

Hinton reemplazó la capa de procesamiento de perceptrón por múltiples capas ocultas, creando la segunda generación de redes neuronales. Esta red podía aprender funciones más complicadas en comparación con el primer perceptrón, mediante el algoritmo de aprendizaje BP; pero a pesar de haber ampliado su capacidad de aprendizaje esta red aún tenía 4 desventajas:

1. Carecían de la capacidad para extraer información de sistemas no supervisados;
2. La señal de corrección se debilita cuando se pasa a través de las múltiples capas;
3. El aprendizaje es demasiado lento a través de las múltiples capas ocultas;
4. Se puede atascar en óptimos locales desfavorables.

Muchas personas intentaron mejorar la red de Hinton, pero no pudieron superar estas desventajas por lo que migraron a otras herramientas de procesamiento como el *SUPPORT VECTOR MACHINE* (SVM) (Mo, 2012).

La arquitectura más común en una ANN es un esquema de estimulación hacia adelante (*feed-forward*), donde los estímulos de entrada son percibidos por sensores, que emiten su información hacia la siguiente capa de neuronas a través de las conexiones sinápticas artificiales que emulan los pesos o ponderaciones de las dendritas en los sistemas biológicos

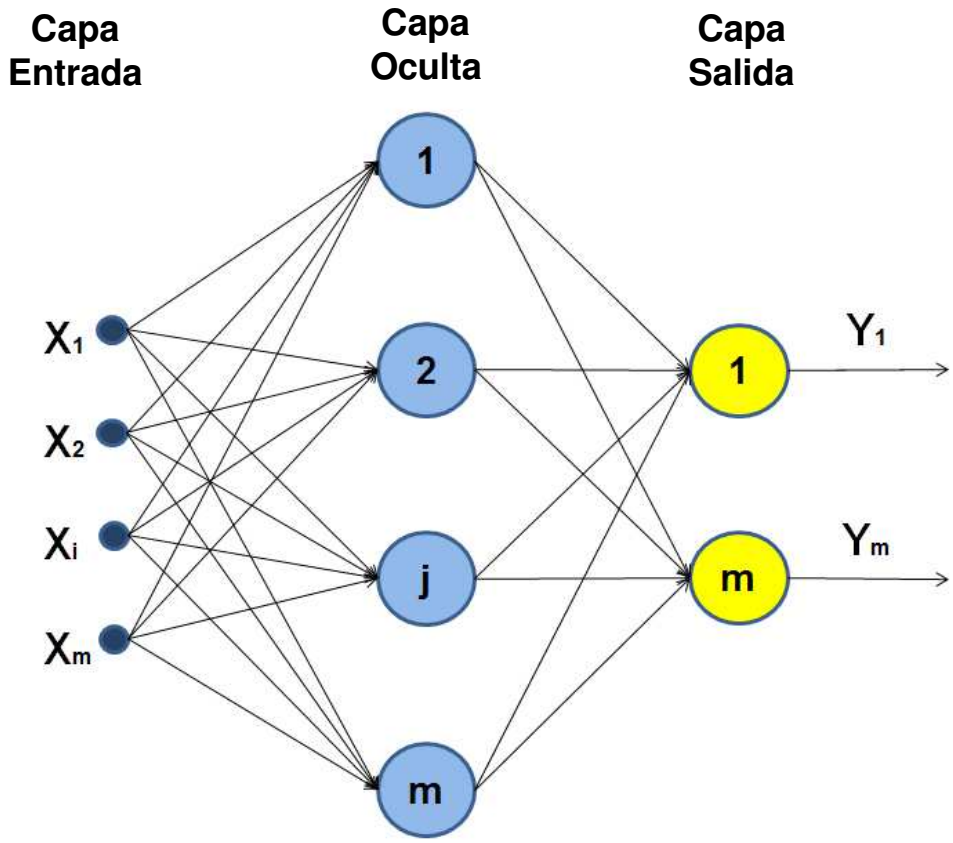


Figura 2-1: Red Feed-Forward.

reales. El proceso interno en cada neurona produce un nuevo estímulo que irá dirigido hacia la nueva capa de neuronas o simplemente será la respuesta final de todo el sistema interconectado. El número de neuronas en cada capa es definido por el usuario y eventualmente esto convierte la forma como son interpretados los estímulos de entrada en la siguiente capa de la ANN. Por lo tanto, aunque se conozcan los estímulos de entrada y las salidas esperadas, el procesamiento en las capas internas puede ser diversificado. Esta diversificación convierte la arquitectura en un sistema de caja negra que estará valorada por los pesos sinápticos adquiridos durante la fase de entrenamiento. Las capas internas son llamadas capas ocultas y el número de estas capas incrementará la complejidad y la diversificación en la interpretación de la información. La Figura 2-1 presenta la estructura de una red *Feed-Forward*.

Sin embargo, cuando se realizaban experimentos de redes con un gran número de capas ocultas se obtenían resultados poco favorables. El problema de optimización asociado a los modelos de *Deep Networks* fue solucionado empíricamente con un algoritmo razonablemente eficiente llamado *Greedy Layer Wise* (Hinton, Osindero, y Teh, 2006; Salakhutdinov y Hinton, 2009), el cual es considerado el primer algoritmo de *Deep Learning* (Nielsen, 2015).

2.2. El Aproximador Universal y el *Deep Learning*

Uno de los objetivos más importantes de la inteligencia artificial es simular la inteligencia humana, por lo que es necesario desarrollar sistemas con arquitecturas y modelos de aprendizaje que representen fielmente el funcionamiento y la arquitectura que posee el cerebro.

Los mecanismos de procesamiento de información humanos sugieren la necesidad de arquitecturas profundas (DA, por sus siglas en inglés) para la extracción y representación interna de la información proveniente de cada entrada sensorial, como por ejemplo el sentido de la audición y la visión. Los sistemas de procesamiento de señales con DA están compuestas de muchas capas de procesamiento no lineal con su salida conectada inmediatamente a la entrada de la capa siguiente. Un método adecuado de aprendizaje contribuye a la naturaleza generativa del modelo y hace uso efectivo de grandes volúmenes de datos tal que extrae las estructuras y las regularidades de la información (Deng, 2011).

Recientemente neuro-científicos han brindado visiones acerca de la representación de la información en el cerebro, lo que ha promovido ideas innovadoras para diseñar nuevos sistemas. Uno de los descubrimientos es la neo corteza, la cual ha sido asociada con muchas habilidades cognitivas. Este descubrimiento impulsó el desarrollo del *Deep Machine Learning*, el cual se enfoca en modelos computacionales para representar la información tal que presente características similares a las de la neo corteza (Arel, Rose, y Karnowski, 2010).

El desarrollo de nuevas arquitecturas ha sido un camino truncado. Inicialmente, las redes *feed–forward* parecían suficientes para resolver problemas de complejidad no lineal; así, Haohan Wang y Bhiksha Raj afirmaban que una red neuronal de 2 capas podría representar cualquier hipótesis, mejor conocido como “Aproximación Universal”; sin embargo, una red neuronal representa únicamente una aproximación de cualquier función, pero no es necesariamente la función exacta y además dicha función debe ser continua. El problema surge cuando se debe procesar grandes volúmenes de datos, ya que naturalmente uno de los mecanismos en una red *feed-forward* es aumentar el número de neuronas y/o capas ocultas que flexibilicen la interpretación y generalización de dichos datos; sin embargo, los algoritmos de aprendizaje resultaron ser ineficientes para este tipo de datos. La necesidad de *Deep Learning* es el resultado de mecanismos de entrenamiento en redes *feed-forward* tal que puedan procesar grandes volúmenes de datos, sin que pierda la capacidad de generalización en la clasificación de estos (Wang y Raj, 2015).

2.3. *Deep Architectures*

2.3.1. Máquinas de Boltzman Restringidas - RBM’s

Las máquinas de Boltzman (BM’s) son redes conectadas bidireccionalmente de unidades estocásticas de procesamiento, que pueden ser tomadas como redes neuronales artificiales. Aunque el entrenamiento de estas redes es complejo en cuanto a dificultad y tiempo de procesamiento, se puede solucionar al imponer restricciones en la topología de la red, dando origen a lo que se conoce como máquinas de Boltzman restringidas (*Restricted Boltzman Machines*), en la Figura 2-2 se muestra la arquitectura de las BM’s y las RBM’s. Después de un correcto entrenamiento, una RBM proporciona una aproximación muy cercana de la distribución subyacente de los datos, la cual puede usarse para comparar probabilidades de

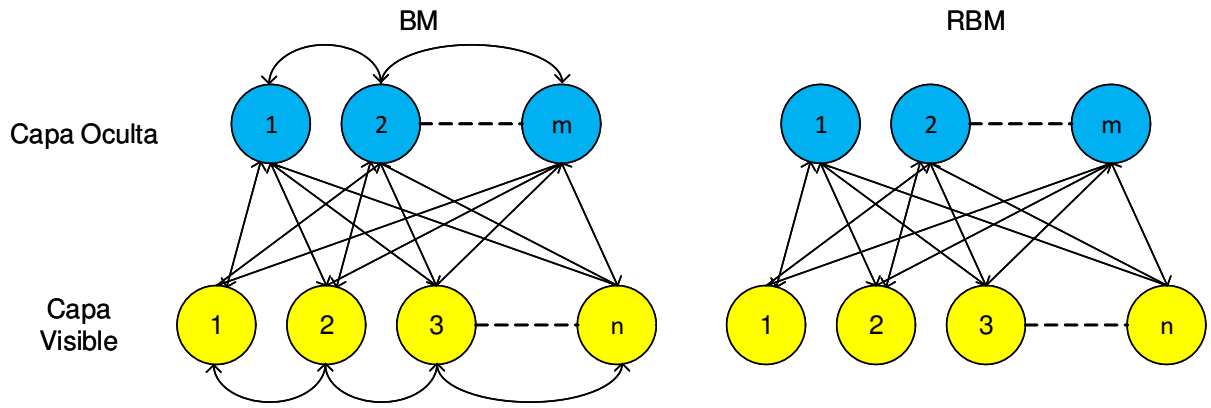


Figura 2-2: a) Maquina de Boltzman. b) Maquina de Boltzman restringida.

datos no observados, o muestrear la distribución marginal de interés.

La idea es que una RBM extraiga las características más relevantes de un conjunto de datos, siendo éstas la entrada de otra RBM y así sucesivamente hasta alcanzar un alto nivel de representación. Una RBM, o un conjunto de RBM's conectadas entre sí, podría interpretarse como una red neuronal *feed-forward* a la cual se le puede agregar una capa de salida con unidades que representen las salidas, haciendo que este modelo corresponda a una red neuronal común usualmente utilizada para clasificación o regresión (Fischer y Igel, 2012). Gracias al aumento en la capacidad de cómputo de los últimos años y al desarrollo de nuevos algoritmos y estrategias de aprendizaje, las RBM's han obtenido una gran utilidad en aplicaciones más interesantes, siendo propuestas como bloques de construcción para arquitecturas multi-capa llamadas *Deep Belief Networks* (DBN).

2.3.2. *Deep Belief Networks* - DBN's

Las *Deep Belief Networks* (DBN's) son modelos de redes neuronales generativos que contienen una gran cantidad de capas con variables y factores ocultos, en las cuales cada capa encuentra las características que mejor se correlacionan en la capa anterior. Las DBN's

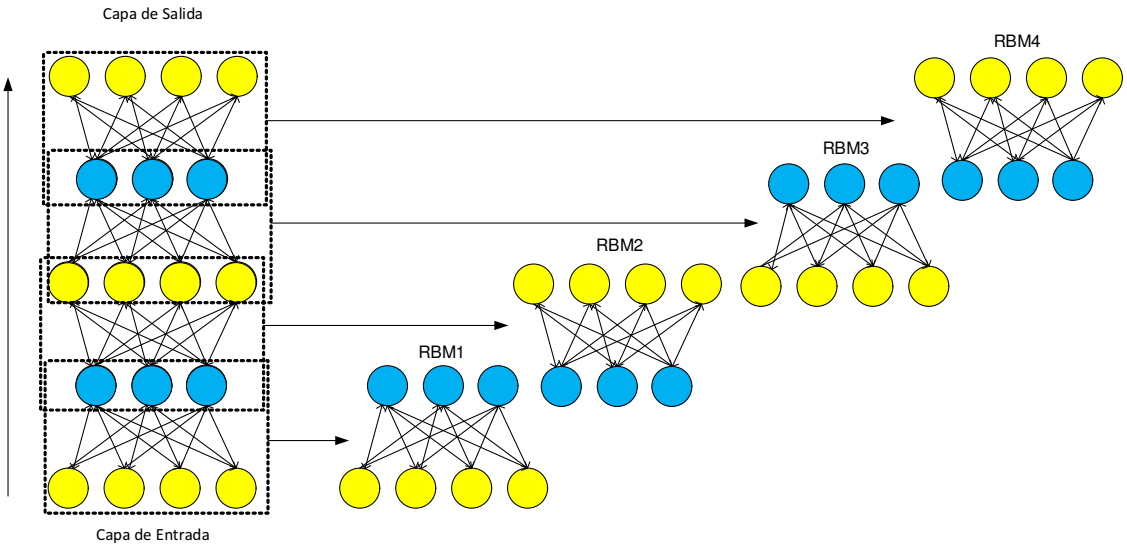


Figura 2-3: Construcción de una DBN.

se construyen a través de RBM's, como se muestra en la Figura 2-3, los cuales tienen una gran capacidad para generalizar modelos exponenciales (Salakhutdinov y Murray, 2008).

Los algoritmos de aprendizaje que se entrenan para representar funciones con muchos niveles de composición se dicen que tienen *Deep Architectures*. Estas arquitecturas son mucho más fuertes en cuanto a la representación de distribuciones más complejas, que aquellas con pocas capas, pero a su vez requieren algoritmos de aprendizaje más complejos (Le Roux y Bengio, 2008). Su entrenamiento es difícil debido a la cantidad de capas ocultas, ya que se debe inferir la distribución condicional de las actividades ocultas cuando hay presencia de un vector de datos. Algunos métodos de aprendizaje variacionales usan aproximaciones simples, pero estas tienden a ser pobres en la capa más profunda, además requieren aprender todos los parámetros por lo que a medida que el número de parámetros aumenta, la red se vuelve ineficiente en el tiempo (Hinton y Salakhutdinov, 2006).

Hinton y Salakhutdinov (2006), propusieron un algoritmo de aprendizaje más rápido en este tipo de redes, el cual consiste en entrenar una capa a la vez, con las 2 capas superiores formando una memoria asociativa no dirigida (undirected associative memory) y el resto de

capas ocultas forman un grafo acíclico dirigido "*directed acyclic graph*" el cual convierte lo que está representado en las capas superiores en variables observables. El procedimiento consiste en entrenar una pila de RBM's teniendo solo una capa como detector de características, y las características aprendidas son usadas para entrenar la siguiente RBM en la fila. Un factor importante de este algoritmo es que las características por capa no disminuyen, y cada capa oculta encuentra una mayor correlación entre las características en la capa anterior (Salakhutdinov, 2015).

2.3.3. *Autoencoders*

Los *Autoencoders* son circuitos de aprendizaje los cuales tienen como objetivo convertir una entrada en una salida con la menor distorsión posible. Los Autoencoders representan uno de los mayores paradigmas del aprendizaje no supervisado, en el cual pequeños cambios sinápticos pueden ocurrir de una manera auto-organizada para obtener un aprendizaje global (Baldi, 2012).

Al igual que algunas de las *Deep Architectures*, los *Autoencoders* no funcionan bien cuando son entrenados con gradiente descendente iniciando los pesos de manera aleatoria, pero cuando se realiza un pre-entrenamiento por cada capa para obtener una mejor representación de los datos partiendo desde la capa inmediatamente anterior, se obtienen mejores soluciones en términos de generalización usando gradiente descendente (Vincent y cols., 2010). Los *Autoencoders* permiten codificar un vector de entrada con muchas características en un vector de salida de menor dimensión y posteriormente permite decodificar este vector y recuperar el vector de entrada original, como se muestra en la Figura 2-4. La etapa intermedia, donde se presenta la mayor reducción de dimensionalidad, facilita la visualización y clasificación de los datos, y con la ayuda de una red MLP entrenada se puede obtener un mejor ajuste en la clasificación. (Hinton y Salakhutdinov, 2006)

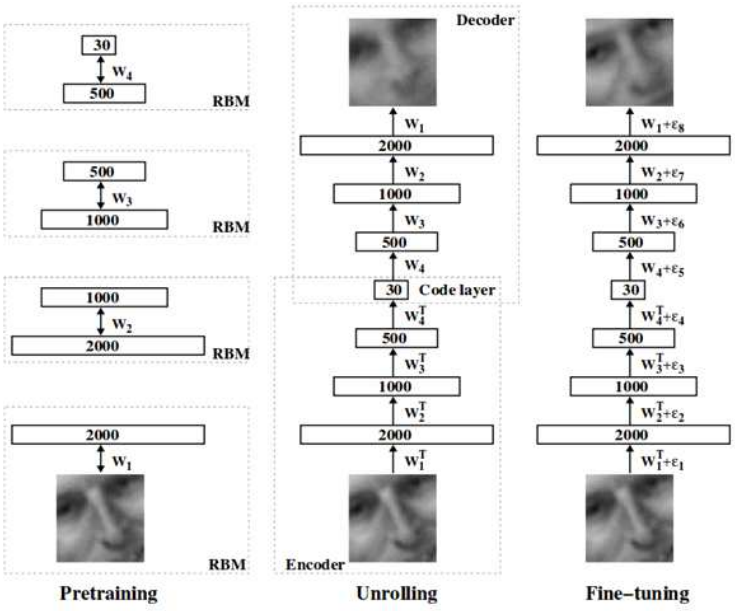


Figura 2-4: Construcción de una Autoencoder (Hinton y Salakhutdinov, 2006).

2.3.4. Redes Neuronales Recurrentes - RNN y Unidades de Memoria a Largo Plazo - LSTM

Las Redes Neuronales Recurrentes o RNN (del ingles *Recurrent Neural Network*) son un tipo de RNA donde las conexiones entre sus unidades de procesamiento o neuronas forman un ciclo dirigido, lo cual permite que estas arquitecturas tengan un comportamiento temporal dinámico, haciendo uso de su memoria interna para procesar entradas arbitrarias. Estas arquitecturas son muy utilizadas en aplicaciones de reconocimiento de texto y voz.

Las unidades de memoria a largo plazo o LSTM (del ingles *Long Short-Term Memory*) surgen como una solución al problema del desvanecimiento del gradiente durante el entrenamiento de las RNN. Estas unidades almacenan información fuera del flujo de datos de las RNN como si fueran, por ejemplo, la memoria de un computador. Estas unidades de almacenamiento deciden que almacenar y cuando permitir su lectura o borrado.

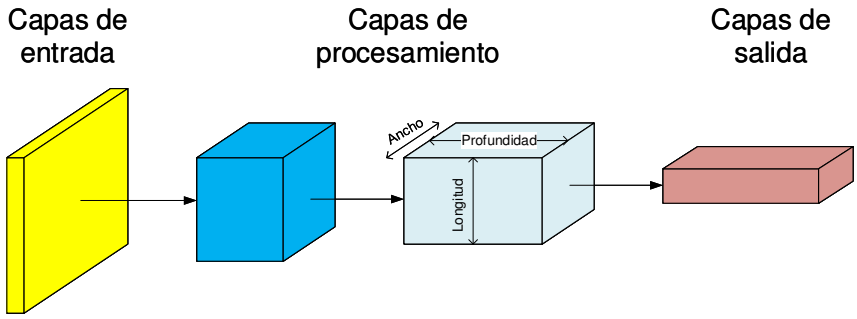


Figura 2-5: Estructura de las CNNs.

2.3.5. Redes Neuronales Convolucionales - CNN

Las redes neuronales Convolucionales (CNN, *Convolutional Neural Network*) son redes neuronales del tipo *feed-forward*, en donde la conexión entre neuronas está basada en la estructura de organización de las neuronas biológicas ubicadas en la corteza visual del cerebro y en cómo éstas procesan las imágenes percibidas por los ojos (LeCun y Bengio, 1998). Las CNN son de gran utilidad en problemas que involucran reconocimiento de imágenes y procesamiento de lenguaje natural.

Las CNN al igual que las ANN convencionales, tienen capas de entrada, capas de procesamiento, pesos sinápticos entre las neuronas, umbrales, funciones de activación y capas de salida. La gran diferencia radica en que las ANN reciben un único vector de entrada y es transformado a través de un conjunto de capas superficiales u ocultas donde las neuronas están completamente interconectadas hasta obtener un resultado en la capa de salida; mientras las CNN, al ser inspiradas en el procesamiento de imágenes, tienen como entrada una región de la imagen o de una región y capas de procesamiento son arreglos de neuronas de 3 dimensiones: ancho, longitud y profundidad, como se observa en la Figura 2-5; es decir, solo existirá conexión entre una neurona y una región de la capa anterior.

En las CNN, cada capa procesa una entrada volumétrica en una salida volumétrica a través de diferentes funciones y transformaciones que pueden tener o no parámetros.

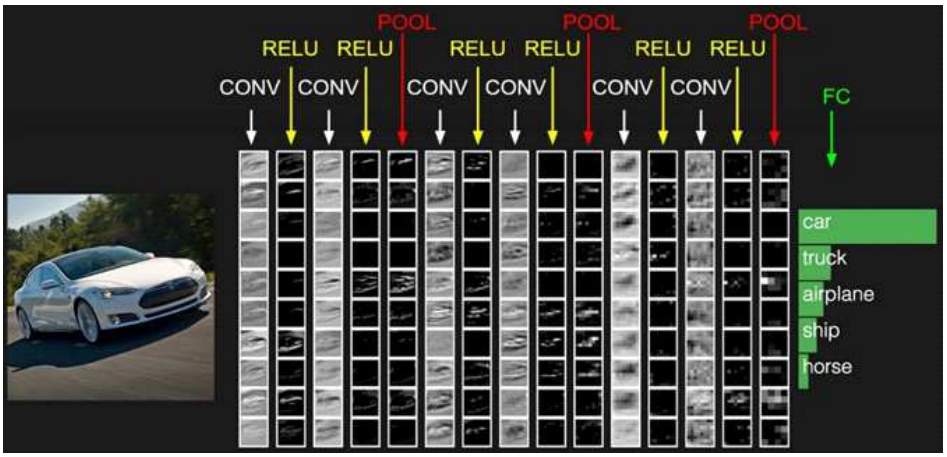


Figura 2-6: Estructura de las CNNs.

2.3.5.1. Arquitectura de las CNN

Existen cuatro tipos de capas que constituyen la arquitectura de las CNN, estas son: Capa de entrada, capa convolucional, capa de activación RELU, Capa de Pooling y una capa completamente interconectada tipo MLP. Como se mencionó anteriormente, en la capa de entrada a cada neurona se le asigna una pequeña fracción de la imagen, esta capa está conectada a la capa convolucional, en la cual a través de diferentes filtros se extraen las características más relevantes y posteriormente son pasadas a través de la capa de activación. En este punto se tienen grandes cantidades de información, por lo cual se usa la capa de *Pooling* para reducir la dimensionalidad. Finalmente, la capa completamente interconectada se encarga de realizar la clasificación. En la aplicación para resolver problemas reales se pueden usar capas combinadas de diferentes maneras, como se observa en la Figura 2-6, de tal forma que se obtengan mejores soluciones.

Capa Convolucional La capa convolucional cumple la función principal en la arquitectura de las CNN, es por eso que en esta capa se requiere una mayor capacidad y tiempo de cómputo. Los parámetros de esta red son una serie de filtros, generalmente pequeños, los

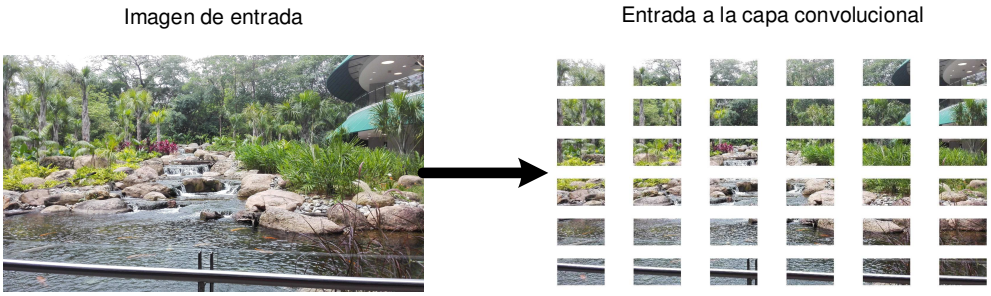


Figura 2-7: a) Imagen de Entrada. b) Forma en que se recorre la imagen en la capa convolucional.

cuales se encargan de recorrer el volumen de entrada a lo largo y ancho, como se ejemplifica en la Figura 2-7, y se realiza producto punto entre la entrada del filtro y la entrada en alguna posición, lo cual da como resultado un mapa de activación en 2 dimensiones. Desde otra perspectiva, cada resultado del producto punto puede ser interpretado como la salida de una neurona, donde esta se enfoca en solo una pequeña región de la entrada. La extensión que ocupa esta conexión es un híper-parámetro conocido como campo de recepción de la neurona, y corresponde al tamaño del filtro. Cada neurona de la capa de convolucion tendrá una dimensión volumétrica donde el ancho y la altura corresponderán al campo de recepción y la profundidad será la misma profundidad de entrada.

En cuanto al volumen de salida de la capa convolucional existen tres hiper-parámetros que controlan la dimensionalidad de este, estos parámetros son:

1. Profundidad (*depth*): Hace referencia al número de filtros que se usan para encontrar características en la entrada. Cuando hay un conjunto de neuronas que están enfocadas a la misma región se conoce como columna profunda (*depth column*).

2. Paso (*stride*): hace referencia al desplazamiento del filtro en la entrada, es decir, el número de pixeles de desplazamiento o avance.

3. Relleno de ceros (*Zero-Padding*): Es de gran utilidad llenar de ceros el borde de la imagen de entrada debido a que permite controlar el tamaño espacial de los volúmenes

de salida, este hiper-parámetro hace referencia al tamaño de dicho relleno.

El volumen de salida W_{out} se puede calcular en función del tamaño del volumen de entrada W_{in}, el campo de recepción F, el paso de los filtros S, y el tamaño del relleno de ceros P. donde:

$$W_{out} = \frac{W_{in} - F + 2 \cdot P}{S} \tag{2.1}$$

Capa de activación ReLU En la capa de activación ReLU (de sus siglas en inglés *Rectified Linear Units*) se utilizan neuronas que tienen una función de activación lineal acotada, lo cual mejora las propiedades no lineales de la función de decisión y en general de toda la red. Dicha función esta descrita por:

$$f(x) = \max(0, x) \tag{2.2}$$

Capa de *Pooling* Las capas de *Pooling* son generalmente utilizadas en la salida de la capa convolucional o entre capas Convolucionales, esto con el fin de mantener o reducir la dimensionalidad del sistema (como se observa en la Figura 2-8), debido a que en la capa de convolución la dimensión espacial y el numero de parámetros del sistema puede aumentar severamente por las neuronas en cada capa de convolución, lo cual puede conllevar a un sobre entrenamiento. Pooling es el proceso mediante el cual se reduce una matriz de dimensión $W1xH1$ a una de dimensión $W2xH2$, donde $W1 > W2$ y $H1 > H2$, usando ventanas de dimensiones m x n que recorren dicha matriz con un paso S determinado. Las operaciones de *Pooling* más comunes son *Pooling* Máximo y *Pooling* Promedio, donde la primera consiste en remplazar cada ventana por el valor máximo de esta, como se muestra en la Figura 2-8, y la segunda consiste en remplazar la ventana por el valor promedio de sus valores.

La dimensionalidad después de la capa de *Pooling* se puede calcular con los parámetros

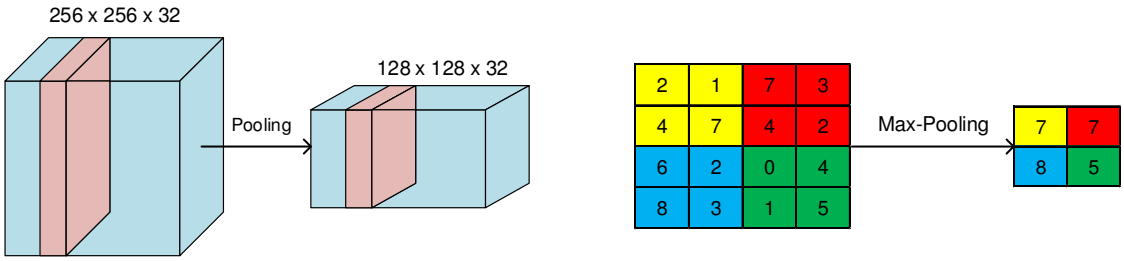


Figura 2-8: a) Proceso de *Pooling*. b) Ejemplo de *Max-Pooling*.

F y S:

$$W_2 = \frac{W_1 - F}{S} + 1 \tag{2.3}$$

$$H_2 = \frac{H_1 - F}{S} + 1 \tag{2.4}$$

$$D_2 = D_1 \tag{2.5}$$

La capa de *Pooling* es de suma importancia en el procesamiento, ya que al reducir la dimensionalidad se reduce la necesidad de una alta capacidad de cómputo y se reduce el sobre-entrenamiento.

Capa Completamente Interconectada Esta capa es la capa de salida de las CNN, se encarga de realizar la clasificación con las características obtenidas de las capas anteriores, esta capa puede ser un MLP o un *Autoencoder*, las cuales suelen tener funciones de activación tipo softmax en la capa de salida.

2.4. Algoritmos de Entrenamiento

2.4.1. *Backpropagation* (BP)

Backpropagation es un método de entrenamiento de redes neuronales que emplea ciclos de propagación hacia atrás de errores junto con métodos de optimización como lo son el gradiente descendente. En una primera fase, se aplica un estímulo de entrada a la RNA y se compara la salida de la red con la salida deseada, el resultado de esta comparación es llevado a cada una de las capas de la red y es comparado con la salida en cada capa.

2.4.2. Métodos de Optimización

Gradiente descendente Gradiente descendente es uno de los algoritmos de optimización más usados para optimizar RNA's. Es una forma de minimizar u optimizar una función de costo parametrizada a través de la actualización hacía atrás de sus parámetros. Se utiliza una tasa de aprendizaje que determina el tamaño de los pasos que conducen a un mínimo. A continuación, se presentan las ecuaciones que definen este procedimiento:

$$L(W) \approx \frac{1}{N}\sum_{i}^{N} fw\left(X^{(i)}\right) + \lambda r(W) \tag{2.6}$$

Donde N es el número de datos, $fw\left(X^{(i)}\right)$ es la perdida para cada entrada $X^{(i)}$ y $r(W)$ es una función de regularización con peso λ. Muchos de los *frameworks* utilizados para el desarrollo de *Deep Learning* utilizan variaciones de este algoritmo para el entrenamiento de sus redes, por ejemplo, *Caffe* en sus librerías cuenta con algoritmos como: gradiente descendente estocástico, AdaGrad, AdaDelta, Adam y NAG.

Gradiente descendente estocástico El gradiente descendente estocástico (SGD) es una simplificación drástica del algoritmo de gradiente descendente clásico, en donde cada iteración se calcula el gradiente tomando como base una muestra aleatoria. A continuación, se presentan las ecuaciones para la actualización de los pesos:

$$V_{t+1} = \mu V_t - \alpha \nabla L(W_t) \tag{2.7}$$

$$W_{t+1} = W_t + V_{t+1} \tag{2.8}$$

Donde V es el valor de la actualización, W es valor de los pesos, $L(W_t)$ corresponde a la función de costo, μ y α, son parámetros de aprendizaje que corresponden al momentum y la tasa de aprendizaje, y pueden ser ajustados para obtener mejores resultados. Este procedimiento depende de la muestra seleccionada en cada iteración. Se espera que el gradiente llegue a un punto óptimo sin importar el ruido introducido por esta simplificación. Este algoritmo es de gran utilidad cuando el proceso de entrenamiento requiere demasiado tiempo de ejecución (Bottou, 2012).

AdaGrad AdaGrad (Duchi, Hazan, y Singer, 2011), es un algoritmo de optimización basado en el SGD, en el cual la tasa de aprendizaje no es fija, si no que se actualiza con cada iteración, adaptándose a la geometría de los datos, es decir, cuando los datos se repiten con mayor frecuencia la tasa de aprendizaje tendrá pequeñas actualizaciones, mientras que cuando la frecuencia sea baja habrá una actualización mayor de los parámetros (Ruder, 2016). A continuación, se presentan las ecuaciones para la actualización de los pesos y de la tasa de aprendizaje:

$$(W_{t+1})_i = (W_t)_i - \alpha \frac{(\nabla L(W_t))_i}{\sqrt{\sum_f^t (\nabla L(W_f))_i^2}} \tag{2.9}$$

Donde W es valor de los pesos, $L(W_t)$ corresponde a la función de costo y α es la tasa

de aprendizaje.

AdaDelta Este algoritmo es una derivación, o extensión del algoritmo AdaGrad, en el cual se mantienen las ideas de mejorar la disminución de la tasa de aprendizaje a través del entrenamiento y la necesidad de seleccionar manualmente una tasa de aprendizaje global (Zeiler, 2012). AdaDelta, a diferencia del AdaGrad que acumula el cuadrado de los gradientes pasados, restringe la ventana de gradientes pasados a un tamaño fijo (Ruder, 2016). A continuación, se presentan las ecuaciones para la actualización de los pesos y de la tasa de aprendizaje:

$$(v_t)_i = \frac{RMS((W_{t+1})_i)}{RMS(\nabla L(W_t))_i} (\nabla L(W_t))_i \tag{2.10}$$

$$RMS(\nabla L(W_t))_i = \sqrt{E \mid g^2 \mid + \varepsilon} \tag{2.11}$$

$$E \mid g^2 \mid_t = \delta E \mid g^2 \mid_{t-1} + (1-\delta) g_t^2 \tag{2.12}$$

$$(W_{t+1})_i = (W_t)_i + \alpha (V_t)_i \tag{2.13}$$

Donde V es el valor de la actualización, W es valor de los pesos, $L(W_t)$ corresponde a la función de costo, g_t es el gradiente de los parámetros a la iteración t, $E \mid g^2 \mid_t$ es el valor esperado del gradiente de los parámetros, α es la tasa de aprendizaje y δ es una constante que controla la variación de los parámetros en la iteración anterior.

Adam Adam es un método de optimización estocástica que solo requiere gradientes de primer orden. Este calcula las tasas adaptativas individuales de aprendizaje para diferentes parámetros desde los primeros y segundos momentos del gradiente. Su nombre viene del inglés "*adaptive moment estimation*" (Kingma y Ba, 2014). A continuación, se presentan las

ecuaciones para la actualización de los pesos y de la tasa de aprendizaje:

$$(m_t)_i = \beta_1 (m_{t-1})_i + (1-\beta_1)(\nabla L(W_t))_i \tag{2.14}$$

$$(v_t)_i = \beta_2 (v_{t-1})_i + (1-\beta_2)(\nabla L(W_t))_i^2 \tag{2.15}$$

$$(W_{t+1})_i = (W_t)_i - \alpha \frac{\sqrt{1-(\beta_2)_i^t}}{1-(\beta_2)_i^t} \cdot \frac{(m_t)_i}{\sqrt{(v_t)_i}+\varepsilon} \tag{2.16}$$

Donde m y V son los valores de actualización, W es valor de los pesos, $L(W_t)$ corresponde a la función de costo, β_1 y β_2 son constantes usadas para obtener mejores resultados y α es la tasa de aprendizaje.

Gradiente acelerado de Nesterov El NAG, es un método de optimización de primer orden con una mejor tasa de convergencia que otros algoritmos de gradiente descendente (Sutskever, Martens, Dahl, y Hinton, 2013). Este método calcula una estimación del siguiente valor de la actualización de los pesos y con base a esta estimación calcula el gradiente completo, para obtener así una mejor tasa de aprendizaje (Ruder, 2016). A continuación, se presentan las ecuaciones para la actualización de los pesos:

$$V_{t+1} = \mu V_t - \alpha \nabla L(W_t + \mu V_t) \tag{2.17}$$

$$W_{t+1} = W_t + V_{t+1} \tag{2.18}$$

Donde V es el valor de la actualización, W es valor de los pesos, $L(W_t)$ corresponde a la función de costo, μ y α, son parámetros de aprendizaje que corresponden al momentum y la tasa de aprendizaje, y pueden ser ajustados para obtener mejores resultados.

2.5. Algoritmos de *Deep Learning*

Deng Li, en su artículo "*An Overview of Deep-Structured Learning for Information Processing*" define *Deep Learning* como una amplia clase de técnicas de *machine learning* y arquitecturas, las cuales usan muchas capas de procesamiento no lineal de información que poseen una jerarquía (Deng, 2011).

"*Deep Learning* permite modelos computacionales que se componen de múltiples capas de procesamiento para aprender representaciones de datos con múltiples niveles de abstracción. Estos métodos han mejorado dramáticamente el estado del arte en cuanto el reconocimiento de voz, reconocimiento de objetos visuales, detección de objetos y en muchos otros campos, como el descubrimiento de fármacos y el genoma. *Deep Learning* descubre complejas estructuras en grandes conjuntos de datos utilizando el algoritmo de Backpropagation para indicar cómo una red debe cambiar sus parámetros internos que son utilizados para procesar la representación en cada capa y desde la capa anterior..." – (LeCun y cols., 2015).

El aprendizaje en las *Deep Networks* es muy difícil debido a la cantidad de neuronas y capas ocultas presentes en una red. Cuando se realiza el entrenamiento con gradiente descendente la propagación del error se vuelve inestable, por lo que la red puede quedar atrapada o estancada en un mínimo local. Debido a este problema se han planteado estrategias a los algoritmos de entrenamiento. Algunos de estos algoritmos se presentan en esta sección.

2.5.0.1. Algoritmo *Greedy Layer Wise*

Este algoritmo originalmente fue propuesto para entrenar DBN's (Hinton y cols., 2006), y consistía en entrenar la red por capas, es decir una capa a la vez. Como se mencionó anteriormente, las RBM's son la unidad básica de construcción de una DBN. En este algo-

ritmo se entrena por separado cada RBM que compone la red, tomando un vector de datos como entrada de la primera capa, y la salida de ésta como la entrada de la segunda y así sucesivamente hasta obtener la salida de la última capa. Este modelo puede ser implementado en cualquier *Deep Network* para inicializar los pesos de cada neurona en cada una de las capas de la red, facilitando así su entrenamiento convencional. Finalmente, la red puede ser completamente entrenada utilizando un proceso de aprendizaje supervisado más fino, o simplemente FINE TUNING (Bengio, 2009).

2.5.0.2. *Fine Tuning*

Una vez realizado el entrenamiento por capas, es posible realizar un entrenamiento global de la red, en el cual se espera obtener una mejor sintonización de los pesos de la red. Después del pre-entrenamiento, toda la red puede ser fuertemente optimizada por gradiente descendente con respecto a otros criterios de entrenamiento (Bengio y cols., 2006).

2.5.0.3. Entrenamiento de un *Autoencoder*

El entrenamiento de un *Autoencoder* es muy similar al propuesto por Hinton para las DBN's. Primero, se entrena la primera capa con el vector de entrada para reducir el error de reconstrucción. Después, con la salida de la primera capa se entrena la segunda, y este proceso se repite iterativamente por cada una de las capas hasta llegar a la última capa oculta. En la capa de salida se realiza un entrenamiento supervisado con la salida de la última capa oculta y las etiquetas de los datos de entrenamiento. Finalmente, se realiza un proceso de *fine tuning* utilizando las etiquetas de los datos de entrenamiento para mejorar la precisión de la red (Vincent y cols., 2010).

2.5.0.4. Entrenamiento de una CNN

Para entrenar CNN's se usa *Backpropagation* de manera convencional. Una vez definida la arquitectura de la red, se inician los pesos de cada una de las neuronas de la red con valores aleatorios y se realiza una propagación hacia adelante, con el resultado de la red ante esta entrada se genera una función de costo, la cual indica que tan lejos o cerca está la red de proporcionar un resultado adecuado; después se hace una propagación hacia atrás para determinar que neuronas son más influyentes en el resultado y posteriormente mediante un algoritmo de optimización se actualizan los pesos de la red.

2.5.1. Frameworks

Para el desarrollo de *Deep Networks* existen diferentes marcos de trabajo, o *frameworks*, los cuales se encargan de facilitar las tareas de diseño al momento de programar, recopilando funciones o algoritmos que se ejecutan en un nivel de programación más bajo como lo es C++. Debido a los tiempos de cómputo que conlleva entrenar estas redes, estos *frameworks* han incorporado interfaces con CUDA, la cual permite procesar información mediante GPUs. A continuación, se presentan algunos de los *frameworks* más populares para el desarrollo de *Deep Networks*.

2.5.1.1. Cuda – CuDNN

CUDA es una plataforma de computación paralela y un modelo de programación desarrollado por NVIDIA, que incrementa el rendimiento de la computación aprovechando la potencia de la unidad de procesamiento gráfico (GPU). Con millones de GPUs habilitadas por CUDA vendidas hasta la fecha, los desarrolladores de software, científicos e investigadores están

encontrando amplios usos para la computación GPU con CUDA[1].

NVIDIA *CUDA Deep Neural Network* (CuDNN) es una librería acelerada por GPU para redes neuronales profundas. CuDNN proporciona implementaciones altamente afinadas para rutinas estándar tales como capas de convolución, agrupación, normalización y activación hacia delante y hacia atrás. CuDNN es parte del SDK de aprendizaje profundo de NVIDIA[2].

2.5.1.2. Deeplearning4j

Deeplearning4j (DL4J) es una librería open-source de *Deep Learning* para Java y Scala, diseñada para ser usada en múltiples GPUs y CPUs. Este framework se enfoca en desarrollos *plug and play*, lo que permite una forma de prototipado rápido para no desarrolladores y que adicionalmente permite importar modelos de otros *frameworks*. Deeplearning4j es compatible con cualquier lenguaje de JVM, y está programado en C, C++ y Cuda. Deeplearning4j permite crear *Deep Networks* a través de redes superficiales las cuales conforman las capas de la red. Este *framework* cuenta con modelos de RBM, *Autoencoders*, redes Convolucionales y redes recurrentes; y permite crear redes híbridas entre estas arquitecturas[3].

Casos de estudio en los campos de la visión artificial, procesamiento de lenguaje y entre otros, presentan resultados poco favorables frente a los obtenidos en otros *frameworks*, esto debido a la cantidad de parámetros que presentan sus modelos. Esto hace que DL4J sea una herramienta muy útil en otros casos donde se requiera un alto grado de paralelización, como lo es el caso de las redes recurrentes. Deeplearning4j es ejecutable sobre cualquier sistema operativo, cuenta con una programación no simbólica, de alta flexibilidad al momento de implementar nuevos sistemas. Para poder ejecutar DL4J se requiere tener instalado una versión superior a la 1.7 del JDK, una versión superior de Cuda 7.5 y una capacidad de

[1] http://www.nvidia.com/object/cuda_home_new.html
[2] https://developer.nvidia.com/deep-learning-frameworks
[3] https://deeplearning4j.org/

computo Cuda de 3.0.

2.5.1.3. TensorFlow

TensorFlow (TF) es una librería open-source para computación numérica usando gráficas de flujos de datos. Su arquitectura flexible permite trabajar en múltiples CPU's o GPU's. Fue desarrollado por investigadores e ingenieros que trabajaban en el equipo de *Google Brain* en conjunto del equipo de desarrollo de máquinas inteligentes de *google* con propósitos orientados a la investigación en *machine learning* y *Deep Networks*. TensorFlow permite que los usuarios puedan agregar sus propias librerías en Python, o escribiéndolas a un nivel más bajo de programación como C++[1]. En cuanto a flexibilidad de modelado TF es más débil que otros frameworks, esto debido a que sus modelos son construidos como una gráfica estática, lo que dificulta su cómputo. TF no implementa convolución 3D, lo que lo convierte en un *framework* poco óptimo para reconocimiento de vídeo. TensorFlow es ejecutable sobre Linux y Mac, cuenta con un tipo de programación simbólica, se programa en Python y C++, y tiene baja flexibilidad al momento de implementar nuevos sistemas. Para poder utilizar este framework se requiere tener instalado una versión superior a la 7.0 de Cuda, una capacidad de cómputo Cuda de 3.0 y una versión igual o superior a la 3.0 de CuDNN.

2.5.1.4. Torch

Torch es un *framework* de computación científica con soporte para algoritmos de *Machine Learning* que usan GPU's. Su lenguaje está basado en scripts y es implementado en C y Cuda. Soporta arreglos N-dimensionales, rutinas de operaciones matriciales, interfaz con C, rutinas de álgebra lineal, modelos de redes neuronales, rutinas de optimización y soporte de GPU. Es considerado un *framework* poco flexible, debido a que para crear un nuevo tipo

[1] https://www.tensorflow.org/

de capa o red es necesario implementar una actualización completa de los algoritmos de gradiente ascendente y descendente. Torch es ejecutable sobre cualquier sistema operativo, cuenta con un tipo de programación no simbólica y tiene su propio lenguaje de programación: Lua. Para poder utilizar este *framework* se requiere tener instalado una versión superior a la 6.5 de Cuda, una capacidad de computo Cuda de 3.0 y una versión igual o superior a la 3.1 de CuDNN.

2.5.1.5. Keras

Keras es una librería de redes neuronales de alto nivel, escrita en Python y con la capacidad de ser utilizada en TensorFlow y Theano. Fue desarrollada teniendo en mente la rápida experimentación, es decir la capacidad de llegar de la idea al resultado con el menor número de retrasos. Keras permite un rápido y fácil prototipado, soporta redes Convolucionales y redes recurrentes y corre tanto en CPU como en GPU. Keras es ejecutable sobre cualquier sistema operativo, cuenta con un tipo de programación simbólica y se programa en Python, tiene una flexibilidad media al momento de implementar nuevos sistemas. Para poder utilizar este framework se requiere tener instalado una versión superior a la 7.0 de Cuda, una capacidad de computo Cuda de 3.0 y una versión igual o superior a la 3.0 de CuDNN.

2.5.1.6. *Microsoft Cognitive Toolkit* (CNTK)

El CNTK es un *toolkit* unificado de *Deep Learning* que describe las redes neuronales como una serie de etapas computacionales. CNTK permite combinar diferentes modelos como redes *Feed-forward*, Redes Convolucionales y redes recurrentes. Implementa aprendizaje por gradiente descendente estocástico con diferenciación automática y paralelización a través de múltiples GPU's[1]. CNTK es ejecutable sobre Linux y Windows, cuenta con un tipo de programación no simbólica, se programa en C++, y tiene alta flexibilidad al momento de implementar

nuevos sistemas. Para poder utilizar este framework se requiere tener instalado una versión superior a la 7.5 de Cuda y una capacidad de computo Cuda de 3.0.

2.5.1.7. Caffe

Caffe es un *framework* de *Deep Learning* diseñado para facilidad de expresión, velocidad y modularidad. Fue desarrollado por el *Berkeley Vision and Learning Center*. Permite cambiar entre el uso de CPU y GPU a través de *flags*. Cuenta con una gran comunidad de desarrolladores que contribuyen constantemente a través de *feedbacks*. Caffe puede procesar 60 millones de imágenes por día con su red convolucional, lo que actualmente la convierte en una de las implementaciones más rápidas[1].

Caffe está diseñado bajo la concepción de ser lo más modular posible, permitiendo crear gran variedad de sistemas con las diferentes funciones, modelos de capas y neuronas que han sido desarrolladas e implementadas. Caffe cuenta con una interfaz para Python y Matlab, que facilita el prototipado y diseño de sistemas que pueden contar con las diferentes arquitecturas y algoritmos implementados en este *framework*.

En cuanto arquitectura, Caffe usa arreglos 4-dimensionales llamados "*blobs*" para almacenar y enviar información entre las diferentes capas. Estos *blobs* son interfaces de memoria que pueden contener parámetros, actualizaciones de parámetros o lotes de información y permiten alternar tareas entre CPU's y GPU's. Caffe carga la información de memoria a un *blob* a través de la CPU, mientras que para computación en GPU se inician núcleos (*kernels*) CUDA, y los *blobs* se pasan de capa a capa de manera transparente.

Las capas de las redes neuronales en Caffe están compuestas por capas Caffe (o en el inglés "*Caffe layers*"), estas capas son las encargadas de procesar internamente los *blobs*

[1] https://github.com/microsoft/cntk
[1] caffe.berkeleyvision.org/

y realizar las operaciones de la red como el *forward* y *backward pass*, que se encargan respectivamente de convertir las entradas en salidas y de calcular el gradiente con respecto a la salida. (Jia y cols., 2014)

A pesar de ser uno de los mejores *frameworks* para visión artificial, su capacidad (o soporte) para el uso de redes recurrentes o procesamiento del lenguaje es débil debido a su arquitectura del *framework*.

Caffe es ejecutable sobre Linux y Mac, cuenta con un tipo de programación No simbólica, se programa en Matlab, Python y C++, y tiene baja flexibilidad al momento de implementar nuevos sistemas. Para poder utilizar este *framework* se requiere tener instalado una versión superior a la 6.5 de Cuda, una capacidad de computo Cuda de 3.0 y una versión igual o superior a la 2.0 de CuDNN.

Los requerimientos de cada uno de los *frameworks* se encuentran resumidos en la siguiente tabla:

2.6. Hardware

Para el entrenamiento de *Deep Networks* es necesario el uso de GPU's debido a la cantidad de operaciones que deben ser efectuadas de un instante de tiempo dado, es importante contar con el hardware requerido para el desarrollo e implementación de estas tecnologías, ya que de lo contrario no se obtendrían buenos resultados. Para el desarrollo de este trabajo se cuenta con la tarjeta de desarrollo NVidia-Jetson TK1.

Tabla 2.1: Frameworks

	DL4J	**TensorFlow**	**Torch**	**Keras**	**CNTK**	**Caffe**
Plataforma	Multi-plataforma	Linux, Mac	Multi-plataforma	Multi-plataforma	Windows, Linux	Linux, Mac
Programacion	No Simbolica	Simbolica	No Simbolica	Simbolica	No Simbolica	No Simbolica
Lenguaje	Java, Scala	Python, C++	Lua	Python	C++	C++, Python, MATLAB
Modelos Pre-Entrenados	Si	Si	Si	Si	Si	Si
Flexibilidad	Alta	Baja	Baja	Media	Alta	Baja
Requerimentos						
Entorno	Intellij IDEA, Eclipse	Pip, Virtualenv, Anaconda	LuaJIT, LuaRocks, QtTorch, QtLua	Pip, Virtualenv, Anaconda	CNTK	CMD, Command Prompt
Java	JDK 1.7+	java8	No	No	No	No
Cuda	7.5 +	7.0 +	6.5+	7.0 +	7.5+	6.5+
CuDNN	No	¿=3.0	¿=3.1	¿=3.0	No	¿=2.0
C.C Cuda	3	3	3	3	3	3
Python	No	2.7, 3.3+	No	2.7, 3.3+	No	2.7, 3.3+

2.6.1. NVidia – Jetson TK1

La tarjeta NVidia-Jetson TK1 es un sistema embebido de desarrollo creado por NVidia, concebido bajo la idea de tener una GPU portable, pero con las mismas características de una GPU de escritorio. Además, permite el uso de periféricos y de sistemas de memoria externos, lo que le da una funcionalidad muy parecida a la de una Raspberry [1]. La tarjeta cuenta con:

- GPU: NVIDIA Kepler “GK20a” con 192 SM3.2 núcleos CUDA (hasta 326 GFLOPS).
- C.C. CUDA: 3.0
- CPU: NVIDIA ”4-Plus-1”2.32GHz ARM quad-core.
- DRAM: 2GB DDR3L 933MHz.
- Almacenamiento interno: 16GB.

2.7. Desarrollo de Software

Al estudiar las características de la Jetson TK1 y observando la tabla 2.1 se aprecia que los únicos *frameworks* que son compatibles y que cumplen con todos los requisitos de hardware son: Caffe-rc3 y TensorFlow v0.8, para este último se encontraron registros en internet de casos en los que se pudo instalar con éxito este *framework*, a pesar de que la Jetson no cumplía en su totalidad los requisitos. Sin embargo, al momento de instalar TensorFlow hubo problemas de incompatibilidad por la versión de java que debía ser instalada y debido a que TensorFlow utiliza funciones que están incorporadas en CUDA 7.5 (el cual no es compatible

[1] http://www.nvidia.com/object/jetson-tk1-embedded-dev-kit.html

con la Jetson TK1), por lo cual se debe llevar a cabo un proceso en donde se instalan solo aquellas funciones que son usadas por el *framework*. Debido a esto, se seleccionó Caffe como el *framework* de desarrollo para este trabajo.

Para el desarrollo de la aplicación se seguirá como guía de diseño la metodología RUP, se utilizó Qt como entorno de desarrollo de la GUI y Python como lenguaje de programación para la interfaz entre Qt y Caffe.

2.7.1. RUP

El Proceso Racional Unificado o RUP (por sus siglas en inglés *Rational Unified Process*) es una metodología que fija un estándar para el desarrollo de software y es una de las metodologías más utilizadas para el análisis, diseño, interpretación y documentación de sistemas orientados a objetos. RUP no es una metodología que fija una serie de pasos estrictos, por lo contrario, propone una serie de metodologías que se adaptan a la etapa y estado en la que se encuentra el proyecto. En esta metodología se proponen 4 fases de trabajo, la primera es la fase de inicio, donde se estudia y define los alcances del proyecto, se identifican los riesgos, se propone una arquitectura de software y se plantea un plan de fases. La segunda es la fase de elaboración, en esta identifican los casos de uso, se realizan las especificaciones de estos, se realiza un análisis del problema y se plantea una solución preliminar. La tercera es la fase de desarrollo, en esta se debe completar la funcionalidad del sistema, verificar que requerimientos están pendientes, administrar cambios y actualizaciones en los requerimientos y se realizan mejoras al proyecto. La última fase es la de transición, en esta se asegura la disponibilidad del software, se detectan y corrigen errores y se genera el soporte técnico necesario (White, 2004).

2.7.2. Qt

Qt es un *framework* multiplataforma libre y de código abierto utilizado para diseñar interfaces gráficas de usuario o interfaces para comunicar diferentes dispositivos o herramientas, como lo son algunos sistemas embebidos, sistemas móviles, etc. C++ es su lenguaje de programación de forma nativa[1], aunque es compatible con otros lenguajes a través del uso de *bindings* (adaptación de librerías para ser utilizadas en otros lenguajes diferentes al lenguaje en que han sido escritas), como lo es el caso de PyQt, el cual permite utilizar librerías de Qt desde Python.

2.7.3. Python

Python es un lenguaje de programación de alto nivel interpretado para programación de propósitos generales. Soporta programación orientada a objetos, programación imperativa y programación funcional. Se caracteriza por tener una síntesis bastante intuitiva y legible, usa tipado dinámico y es multi-plataforma[2].

2.8. Discusión

Teniendo en cuenta que se desea implementar D*eep Networks* en un hardware embebido Jetson TK1 con capacidad de procesamiento paralelo a través de un núcleo de CUDA, se realizó una revisión bibliográfica para estimar los requerimientos funcionales y apropiados tal que permitan entrenar y validar el funcionamiento de las arquitecturas *Autoencoders* y redes CNN. Estas arquitecturas, la cuales fueron presentadas en la sección 2.3, serán desarrolladas

[1] https://www.qt.io/
[2] https://www.python.org/

tal que sea posible utilizar los diferentes métodos de entrenamiento presentados en la sección 2.5.

Por otro lado, se desarrollará una aplicación basada en la metodología RUP que permite generar una estructura de software conforme a los requerimientos y usos que tendrá esta. El software Qt, ha sido seleccionado como entorno de desarrollo para el diseño de la interfaz gráfica de usuario debido a sus características multi-plataforma y compatibilidad con otros lenguajes, tal como Python el cual se puede integrar a la interfaz gráfica y las librerías del *framework*, a través de PyQt y PyCaffe, respectivamente.

Capítulo 3

Metodología

A partir de la revisión realizada en los *frameworks* y los sistemas de software compatibles con ellos para diseñar una interfaz, este capítulo presenta el desarrollo metodológico necesario para integrar el *framework* Caffe en una interfaz de usuario diseñada en Qt. Para el desarrollo de la interfaz es necesario construir un sistema de 3 etapas: Diseño de una Interfaz de Gráfica de Usuario (GUI) en Qt, donde el código fuente es desarrollado utilizando la librería PyQt y finalmente, la interfaz de este código fuente con el *framework* Caffe a través de PyCaffe.

La metodología de desarrollo se basa en la descripción de cada una de las etapas necesarias para que el *framework* sea utilizable en una interfaz amigable sin que sea necesaria la comprensión de módulos de procesamiento de Caffe, donde se integran diferentes arquitecturas *Deep Networks* con los diferentes algoritmos de aprendizaje *Deep Learning*.

3.1. Interfaz Gráfica de Usuario – GUI

Para el desarrollo de la interfaz gráfica de usuario o GUI, se utilizó el entorno de diseño gráfico Qt Designer versión 4.8.6, el cual permite crear plantillas y bosquejos con diferentes elementos de gran utilidad. La GUI está compuesta de una ventana principal que contiene 2 pestañas, tal como se presenta en la Figura 3-1, donde la primera pestaña contiene un conjunto de paneles que permiten crear la arquitectura de la red y efectuar su entrenamiento, mientras la segunda contiene diferentes opciones de visualización de cada una de las capas y la validación del sistema.

3.1.1. Pestaña de entrenamiento

Esta pestaña cuenta con 6 paneles que permiten la configuración de los parámetros de entrenamiento y las características de la red. El primer panel corresponde a la selección de base de datos y cuenta con dos campos de texto y un botón para cada uno de estos, los cuales permiten: abrir un explorador, seleccionar el archivo donde se encuentra el conjunto de entrenamiento y seleccionar el archivo del conjunto de validación. Una vez seleccionados cada uno se mostrarán sus rutas en los campos de texto correspondiente. Adicionalmente, junto a cada campo de texto hay una casilla que permite ingresar el tamaño del lote de muestras para el entrenamiento en cada iteración; finalmente, se cuenta con una entrada de texto en la cual se debe escribir el nombre con el que se va a guardar la red. En la sección A1 de la Figura 3-1 se muestra el panel de selección de base de datos.

El segundo panel presentado en la sección A2 de la Figura 3-1 tiene la función de ajustar los parámetros para realizar el entrenamiento. Estos parámetros son guardados en el archivo solver.prototxt de Caffe para posteriormente ser ejecutados por el mismo framework. Dentro de las propiedades de este panel se encuentran: seleccionar el tipo de gradiente que se

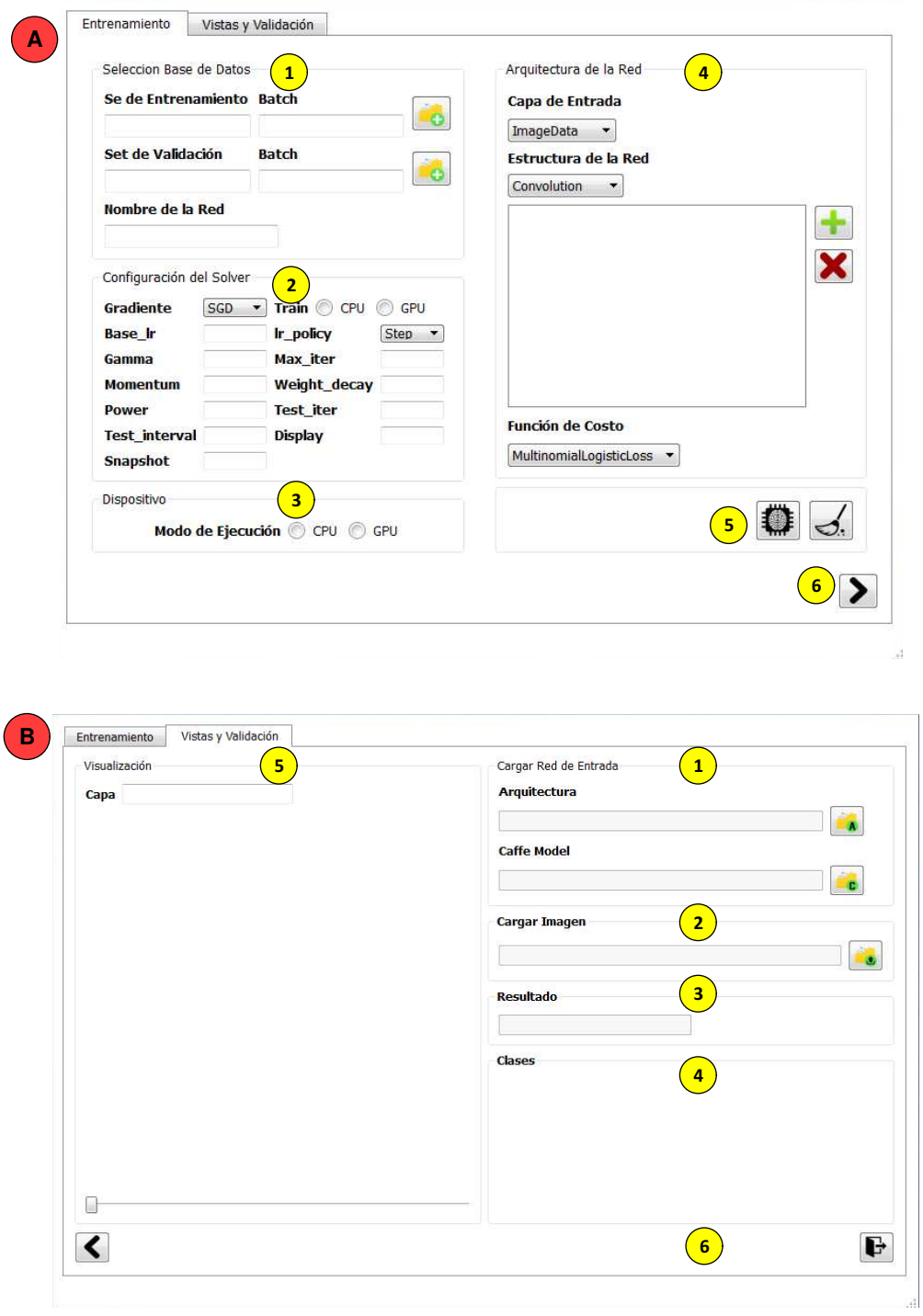


Figura 3-1: A)1. Selección base de datos, 2. Configuración solver, 3. Modo de ejecución, 4. Arquitectura de la red, 5. Botones de entrenar y limpiar, 6. Botón siguiente. B) 1. Cargar modelo entrenado, 2. Cargar imagen, 3. Resultado, 4. Probabilidad de la clase, 5. Panel de visualización.

desea utilizar para el entrenamiento, definir donde realizar el procesamiento del gradiente (CPU o GPU) y configurar cada uno de los parámetros a través de una línea de entrada de texto; estos parámetros serán cargados al solver.prototxt de Caffe al momento de presionar el botón de entrenamiento y se guarda una copia con la configuración con la cual se realizó el entrenamiento. La descripción de cada uno de los parámetros se presenta en la Tabla 3.2. Adicionalmente, es posible seleccionar la arquitectura de procesamiento (CPU o GPU) donde se va a ejecutar el forward pass de la *Deep Network* o arquitectura de red seleccionada, obteniendo así 4 diferentes modos de operación (ver Tabla 3.1), los cuales pueden ser configurados a conveniencia. Estos modos de operación se encuentran en el tercer panel, el cual se presenta en la sección A3 de la Figura 3-1.

Tabla 3.1: Modos de operación de Caffe.

Modo Op.\ Procedimiento	Backward pass – solver	Forward pass
1	GPU	GPU
2	GPU	CPU
3	CPU	GPU
4	CPU	CPU

El cuarto panel se encarga de configurar la arquitectura de la red. Este panel se divide en 3 secciones, las cuales están separadas según el tipo de capas que ofrece Caffe. La primera sección permite seleccionar la capa de entrada, la cual se presenta a través de una lista desplegable donde se encuentran los diferentes tipos de capa de entrada. La segunda sección define la estructura de la red haciendo uso de una lista desplegable donde se encuentran las capas disponibles del framework; adicionalmente, la segunda sección cuenta con una lista editable que permite visualizar las capas que han sido creadas y la manera en que están conectadas; cada uno de los parámetros de cada capa puede ser editado en el momento que se adiciona la capa. Al momento de presionar el botón adicionar se presentará una ventana emergente que contiene todos los parámetros de cada una de las capas (Ver Figura 3-2), estando habilitados aquellos correspondientes a la capa en cuestión; el botón eliminar cumple la función de remover la última capa editada. En la última sección se define la función de costo

Tabla 3.2: Parámetros de entrenamiento.

Parámetro	Descripción
Gradiente	Indica que algoritmo Backpropagation se utilizara al momento del entrenamiento
Procesamiento	Selecciona el tipo de procesamiento que tendrá el entrenamiento.
base_lr	Rata de entrenamiento inicial. Tipo flotante
lr_policy	Indica cómo debe cambiar la rata de entrenamiento a través del tiempo. Tipo String, valores: Step, multistep, fixed, exp, poly, sigmoid
gamma	Indica que tanto debe cambiar la rata de aprendizaje en la siguiente iteración
stepsize	Indica la frecuencia con la que se debe pasar al siguiente paso
stepvalue	Indica el valor de conteo de iteraciones antes de pasar al siguiente paso
max_iter	Es uno de los parámetros de parada
momentum	Indica que tanto valor debe tener el peso anterior el cálculo actual
weight_decay	Indica el factor de regularización o penalización de pesos grandes.
snapshot	Indica Caffe cada cuanto debe generar un reporte del estado actual del entrenamiento
snapshot_prefix	Directorio y nombre del modelo entrenado
Net	Indica la ubicación de la red que se va a entrenar
test_iter	Indica cuantas iteraciones de prueba se deben hacer por cada intercalo de test_interval
test_interval	Indica la frecuencia con la que se ejecutará la fase de prueba de la red.
display	Indica la frecuencia con la que Caffe debe dar resultados

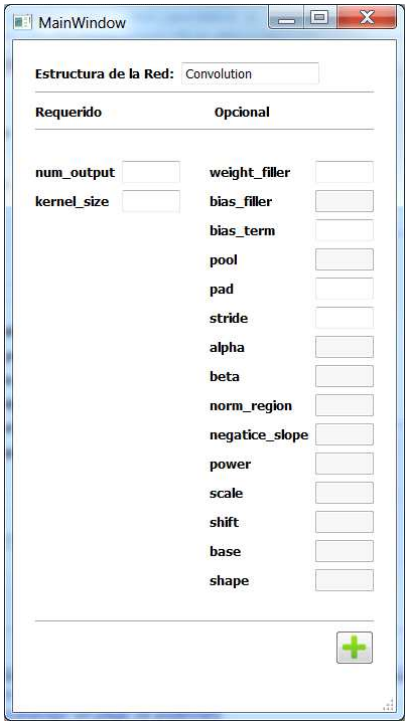


Figura 3-2: Ventana de configuración de parámetros.

de la red para poder ejecutar su entrenamiento; esta sección cuenta con una lista desplegable de funciones disponibles para el proceso de adaptación y aprendizaje. El catálogo de capas disponibles en cada sección se presenta en la Tabla 3.3.

En el quinto panel se encuentran dos botones, uno encargado de iniciar el entrenamiento de la red creada y el otro limpia todos los campos de la GUI. Una vez finalizado el proceso de entrenamiento se presenta una gráfica emergente que muestra la evolución de la función de costo definida y la precisión del sistema en cada iteración.

Finalmente, en el sexto panel se encuentra un botón que permite cambiar la pestaña actual a la pestaña donde se encuentran las vistas de las salidas de cada capa en aquellas arquitecturas que usan filtros o están compuesta de imágenes; la última pestaña contiene la salida de validación del sistema.

Tabla 3.3: Catálogo de capas.

	Capa	parametro	parametro	tipo	Descripcion
Entrada	Data	requerido	source	string	Recibe bases de datos en formato LMDB
			batch_size	int	
	HDF5Data	requerido	source	string	Recibe bases de datos en formato HDF5
			batch_size	int	
Estructura, funciones y utilidades	convolution	requerido	num_output	int	Realiza convolucion
			kernel_size	int	
		opcional	weight_filler	string	
			bias_term	bool	
			pad	int	
			stride	int	
	Pooling	requerido	kernel_size	int	Realiza pooling
		opcional	pool	string	
			pad	int	
			stride	int	
	Deconvolution	requerido	num_output	int	Convolucion transpuesta
			kernel_size	int	
		opcional	weight_filler	string	
			bias_term	bool	
			pad	int	
			stride	int	
	Im2col	-	-	-	conversion imagen a columan
	InnerProduct	requerido	num_output	int	capa completamente conectada
		opcional	weight_filler	string	
			bias_filler	string	
			bias_term	bool	
	Dropout	-	-	-	
	Embed	requerido	num_output	int	Capa util con codificacion one-hot
	LRN	opcional	alpha	int	Local Response Normalization
			beta	int	
			norm_region	string	
	MVN	-	-	-	Mean Variance Normalization
	BatchNorm	-	-	-	Batch Norm Layer
	ReLU	opcional	negatice_slope	bool	Rectificador Lineal
	ELU	-	-	-	Unidades lineales exponenciales
	Sigmoid	-	-	-	Funcion Sigmoidal
	TanH	-	-	-	
	Power	opcional	power	int	Potenciacion
			scale	double	
			shift	double	
	Log	opcional	base	int	Logaritmo
			scale	double	
			shift	double	
	Reshape	opcional	shape	int	reconfigura la dimension de un bloob
	Softmax	-	-	-	
Funciones de costo	MultinomialLogisticLoss	-	-	-	Costo logistico multinomia
	SoftmaxWithLoss	-	-	-	Calcula el costo logistico multinomia con softmax
	EuclideanLoss	-	-	-	Costo euclediano
	Accuracy	-	-	-	calcula la precision del sistema

3.1.2. Pestaña de salidas intermedias y validación

Esta pestaña cuenta con 6 paneles que permiten seleccionar una red previamente entrenada, cargar una imagen para su validación o clasificación, y visualizar la respuesta y el comportamiento de la red.

El primer panel es el encargado de cargar una red previamente entrenada. El panel cuenta

con dos botones que permiten abrir un explorador para seleccionar el archivo .prototxt que contiene la arquitectura de la red a validar y el archivo .caffemodel que contiene todos los parámetros y pesos de la red entrenada; dos campos de texto permiten visualizar la ruta donde se encuentran estos archivos. Este panel se encuentra representado en la sección B1 de la Figura 3-1.

El segundo panel cuenta con un campo de texto que muestra la dirección y nombre de la imagen que va a ser utilizada para validar o probar el sistema. Para seleccionar la imagen hay un botón que abre un explorador y permite seleccionar una imagen. Este panel se encuentra representado en la sección B2 de la Figura 3-1.

El tercer panel cuenta con un campo de texto que es utilizado para visualizar cuál es el valor de la etiqueta detectada para la imagen de entrada. Este panel se encuentra representado en la sección B3 de la Figura 3-1.

El cuarto panel presenta un histograma donde se presenta el valor de la probabilidad de cada clase al momento de realizar la clasificación. Este panel se encuentra representado en la sección B4 de la Figura 3-1.

En el quinto panel se presentan 3 elementos: una entrada de texto, un espacio para visualizar imágenes y una barra deslizable. En el espacio para visualizar imágenes se cargará la imagen seleccionada, con la barra deslizable se podrá cambiar la imagen para observar cómo cambia a través de las capas y el campo de texto mostrará la capa actualmente seleccionada.

3.2. Caffe

Caffe es un framework para el desarrollo e implementación de técnicas de *Deep Learning*, el cual ha sido construido pensando en la facilidad de expresión definiendo y guardando

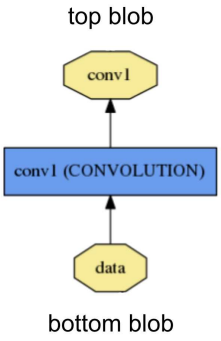


Figura 3-3: Entrada y salida de una capa en Caffe.

sus modelos, arquitecturas y parámetros en esquemas de texto sin formato. Este framework cuenta con una gran cantidad de algoritmos y funciones los cuales han sido implementados de manera óptima para su ejecución en CPU y en GPU, siendo uno de los frameworks más veloces en el desarrollo de aplicaciones en visión artificial (Jia y cols., 2014). La creación de arquitecturas en Caffe se hace de manera modular agregando desde capas de entrada hasta capas de salida con un amplio y variado catálogo de opciones, como el utilizado en este trabajo (ver Tabla 3.3). Aunque cuenta con el respaldo de una gran comunidad científica, tiene desventajas al momento de crear, desarrollar, implementar y validar nuevos sistemas, debido a que no se hace de manera intuitiva y trabajar directamente sobre él resulta engorroso.

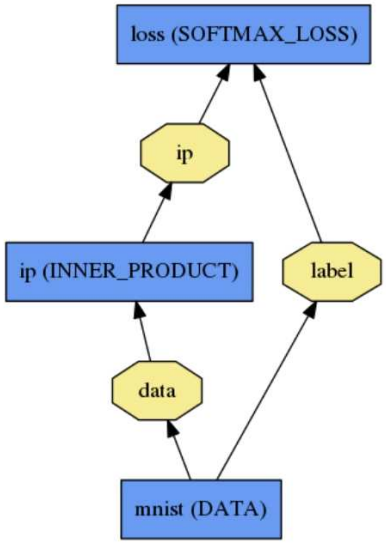
Las capas de Caffe son la esencia de diseño en este framework, ya que estas representan desde capas reales como las capas de una red convolucional, las capas de neuronas, las capas ReLU, hasta capas que representan funciones, operaciones o entrada y salida de datos. Las capas en Caffe toman como entrada un blob inferior (bottom blob) y tienen como salida un blob superior (top blob) como se muestra en la Figura 3-3. Un blob representa la información que pasa a través de la red en un arreglo unificado de memoria, el cual puede ser desde los nombres de cada capa hasta los parámetros propios de la arquitectura de la *Deep Network* como sus pesos sinápticos, las dimensiones de cada capa, entre otros.

Cada una de las capas en Caffe debe cumplir con 3 funciones esenciales para el buen

funcionamiento del sistema. La primera función es la inicialización de las conexiones de la capa y de sus parámetros, la segunda es la capacidad de computar una entrada en una salida y la tercera es tomar la salida y calcular el gradiente descendente hacia la entrada.

Cuando se diseña una red a través de Caffe, se habla de una serie de capas conectadas en un grafo acíclico dirigido o DAG (ver Figura 3-4a, DAG - directed acyclic graph) evitando bucles en la estructura de la red. Una arquitectura de red en modelos composicionales se representa por un conjunto de capas conectadas entre sí para procesar un lote de información. Caffe define las redes capa a capa bajo su propio esquema de modelo, la arquitectura se define desde la capa encargada de recibir la información hasta la última capa que se encarga de calcular la función de costo del sistema. Caffe internamente se encarga de verificar que las capas estén conectadas correctamente para detectar y evitar problemas al momento de diseñar la arquitectura. La red se define como una serie de capas especificando la conexión existente entre cada capa y sus parámetros o funciones en un lenguaje plano sin formato. Las arquitecturas, representadas por modelos en Caffe, se definen en el esquema de buffer de protocolo de texto plano ".prototxt" mientras que el resultado del aprendizaje, representado por los pesos sinápticos de cada capa, son guardados como un buffer de protocolo binario en archivos de extensión ".caffemodel".

El procedimiento por el cual Caffe obtiene una salida ante la presencia de una entrada es conocido como forward pass. En este procedimiento la entrada pasa a través de primera, en la cual se ejecutan las diferentes funciones y operaciones convirtiéndose en la entrada de la siguiente capa hasta llegar a la salida en la última capa. Cuando se realiza el proceso de entrenamiento esta salida pasa por una última capa en Caffe encargada de calcular la función de costo del sistema. Para esto cuenta con 4 diferentes capas Caffe presentadas como Funciones de Costo en la Tabla 3.3. El aprendizaje en Caffe está dirigido en 2 partes, la primera es el modelo de la arquitectura que se encarga de generar las salidas, evaluar la función de costo y calcular los gradientes; la segunda es el solver, el cual se encarga



(a) DAG

```
name: "LogReg"
layer {
  name: "mnist"
  type: "Data"
  top: "data"
  top: "label"
  data_param {
    source: "input_leveldb"
    batch_size: 64
  }
}
layer {
  name: "ip"
  type: "InnerProduct"
  bottom: "data"
  top: "ip"
  inner_product_param {
    num_output: 2
  }
}
layer {
  name: "loss"
  type: "SoftmaxWithLoss"
  bottom: "ip"
  bottom: "label"
  top: "loss"
}
```

(b) Implementacion en Caffe

Figura 3-4: clasificador de regresión logística

de supervisar la optimización del modelo coordinando la inferencia directa de la red y los gradientes hacia atrás para formar actualizaciones de parámetros que intentan mejorar la pérdida.

Adicionalmente, Caffe cuenta con una interfaz diseñada para Python llamada PyCaffe, la cual permite trabajar Caffe con cierto nivel de integridad y modularidad tal que permite manejar todos los archivos, funciones, procesos y parámetros que requiere y genera Caffe en funciones y métodos diseñados en Python, que pueden ser utilizados y ejecutados en un mismo script.

En la Figura 3-5 se presenta la implementación en PyCaffe del modelo presentado en la Figura 3-4. Para utilizar las librerías y componentes de PyCaffe es necesario importar Caffe al inicio del script, tal como se muestra en la línea 1 de la imagen. Para crear la estructura del clasificador de regresión logístico se debe instanciar el objeto “layers” y “params”, que contienen respectivamente el conjunto de capas disponibles en Caffe y los parámetros de

```
impot caffe
from caffe import layers as L, params as P

def LogReg(lmdb, batch_size):

    n = caffe.NetSpec()
    n.mnist, n.label = L.Data(batch_size=batch_size,
                              backend=P.Data.LMDB,
                              source=lmdb,
                              ntop=2)
    n.ip = L.InnerProduct(n.mnist,
                          num_output=2,
    n.loss =  L.SoftmaxWithLoss(n.ip, n.label)
    return n.to_proto()
```

Figura 3-5: Clasificador de regresión logística en PyCaffe.

configuración para cada una de las capas. Se crea una función que tiene como parámetros de entrada la base de datos y el tamaño del lote de entrenamiento, en donde se instancia el objeto “NetSpec()” el cual permite crear la arquitectura capa por capa, definiendo el flujo de blobs y los parámetros de cada capa.

De manera similar ocurre con los parámetros de entrenamiento guardados en el solver, donde para configurarlo en PyCaffe se debe invocar el objeto “Caffe_pb2.SolverParameter()” y asignar un valor a cada uno de los paramentos, como se muestra en la Figura 3-6.

Sin un previo conocimiento en el framework, el diseño de *Deep Networks* en Caffe resulta complejo. El diseño de la aplicación a través de Python y la librería PyCaffe se convierte en una alternativa clara y sencilla donde es posible diseñar, entrenar y validar *Deep Networks* de manera gráfica sin tener que interactuar directamente con algún lenguaje de programación, por ejemplo, en la Figura 3-7 se muestra la implementación del clasificador de regresión logística usando la aplicación.

Finalmente, este trabajo presenta una aplicación que propone una metodología para la creación, desarrollo y validación de diferentes arquitecturas de una manera gráfica e intuitiva utilizando el framework Caffe a través de una interfaz desarrollada en Qt y enlazada usando PyCaffe. La metodología propuesta se resume en la figura 3-8.

```
1  from caffe.proto import caffe_pb2
2  s = caffe_pb2.SolverParameter()
3  s.train_net = train_net_path
4  s.test_net.append(test_net_path)
5  s.test_interval = 500
6  s.test_iter.append(100)
7  s.max_iter = 10000
8  s.type = "SGD"
9  s.base_lr = 0.01
10 s.momentum = 0.9
11 s.weight_decay = 5e-4
12 s.lr_policy = 'inv'
13 s.gamma = 0.0001
14 s.power = 0.75
15 s.display = 1000
16 s.snapshot = 5000
17 s.snapshot_prefix = 'mnist/custom_net'
18 s.solver_mode = caffe_pb2.SolverParameter.GPU
```

Figura 3-6: Configuración Solver en PyCaffe.

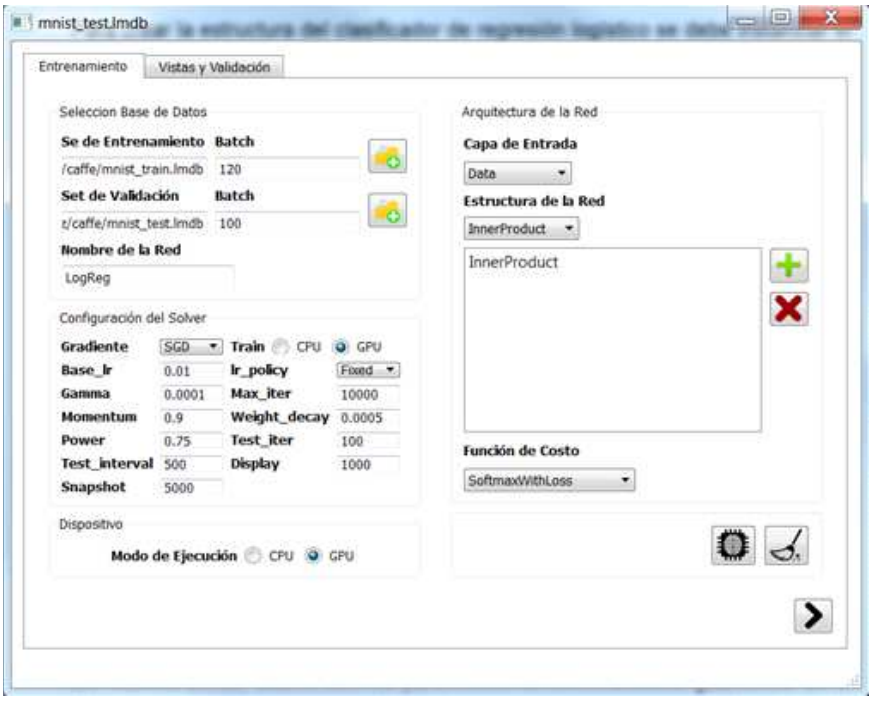


Figura 3-7: Clasificador logístico de regresión implementado en la GUI.

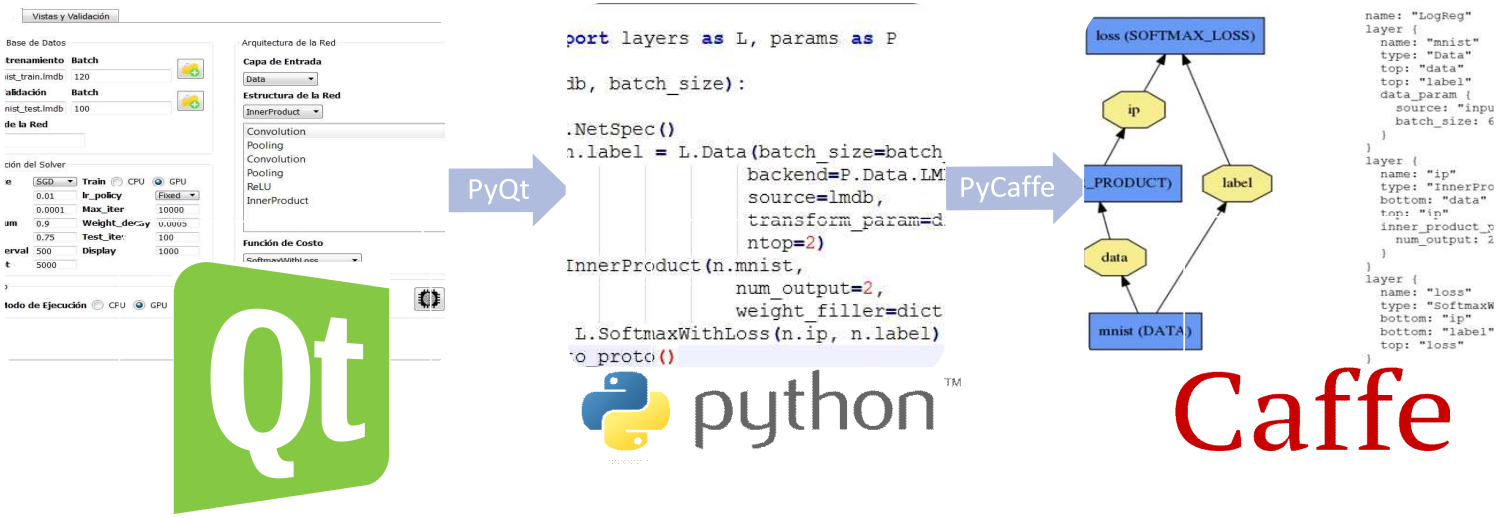


Figura 3-8: metodología propuesta para el desarrollo de *Deep Networks*.

Capítulo 4

Pruebas y Resultados

4.1. Introducción

Luego de hacer una descripción metodológica del aplicativo desarrollado en la plataforma Jetson TK1 y los lenguajes de programación utilizados para el *framework* de *Deep Networks* en la misma, este capítulo se centra en la presentación de resultados obtenidos a través de la plataforma. El objetivo es demostrar la versatilidad de la herramienta desarrollada tal que se puedan verificar los alcances y limitaciones de la misma. Tal como se presentó en el capítulo anterior, la herramienta está en la capacidad de configurar diferentes algoritmos de entrenamiento así como diferentes arquitecturas. Algunas de las bases de datos comúnmente utilizadas en este tipo de arquitecturas son ejecutadas a través de la metodología propuesta usando diferentes algoritmos de entrenamiento.

4.2. Configuración de las pruebas

Para realizar las pruebas de funcionamiento se definió un protocolo de operación en el cual se utilizaron 3 bases de datos, implementando 3 arquitecturas y 5 algoritmos de aprendizaje. Las bases de datos seleccionadas para las pruebas fueron:

- MINST: Es una base de datos compuesta de 60000 imágenes de entrenamiento y 10000 imágenes de validación que contienen números manuscritos del 0 al 9. Las imágenes de esta base de datos han sido normalizadas y centradas usando su centro de gravedad, tienen una dimensión de 28x28 pixeles y se eliminaron los niveles de gris, con el fin de realizar el proceso de entrenamiento utilizando valores lógicos, es decir 1 y 0[1].

- Cifar10: Esta base de datos consiste de 60000 imágenes a color de 32x32 pixeles, clasificadas en 10 clases, donde cada una de estas clases tiene 6000 imágenes. La base de datos se distribuye en 50000 imágenes de entrenamiento y 10000 imágenes de validación. Las clases de la base de datos son excluyentes y no existe ningún tipo de solapamiento entre clases[2].

- ImageNet: Es una base de datos diseñada para el desarrollo de paquetes software especializados en reconocimiento de objetos por visión artificial. Esta base de datos se actualiza de imágenes que han sido publicadas en internet y que poseen una etiqueta para facilitar su identificación[3].

Cada una de las bases de datos fue probada utilizando 3 diferentes arquitecturas, ajustadas según la complejidad de clasificación. Estas arquitecturas fueron:

[1] http://yann.lecun.com/exdb/mnist/
[2] https://www.cs.toronto.edu/~kriz/cifar.html
[3] http://www.image-net.org/

- DBN: Se utilizó una DBN de 5 capas con estructura 500-1000-500-1000-500 y una capa de salida de 10 neuronas para realizar la clasificación. La figura 4-1a muestra la arquitectura implementada en la aplicación y la figura 4-1b muestra la estructura de red desarrollada.

- Autoencoder – MLP: Se utilizó una *Deep Network* con la arquitectura de un Autoencoder 1000-500-250-10, aunque este fue entrenado como una MLP debido a las dificultades que presenta Caffe al momento de entrenar Autoencoders. La figura 4-1c muestra la arquitectura implementada en la aplicación y la figura 4-1d muestra la estructura de red desarrollada.

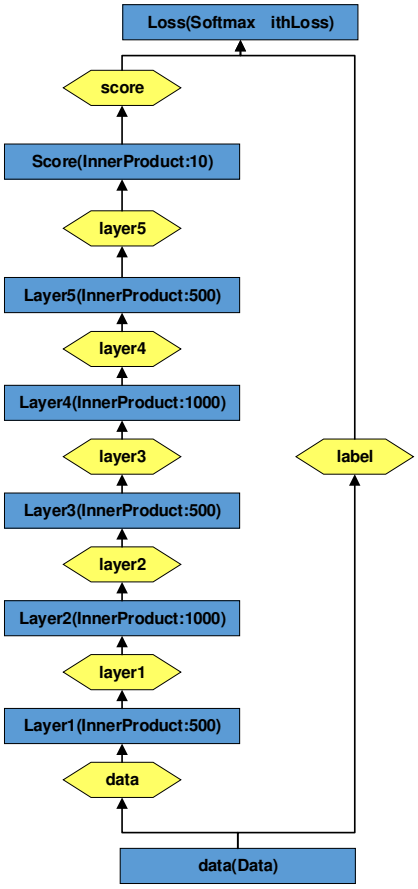
- LeNet: Se utilizó una réplica ajustada de la red convolucional propuesta por Yan LeCunn en 1998 (Lecun, Bottou, Bengio, y Haffner, 1998), esta red cuenta con la siguiente estructura: entrada - convolución – *pooling* – convolución – ReLu – *pooling* – FC – ReLu – salida. La figura 4-2a muestra la arquitectura implementada en la aplicación y la figura 4-2b muestra la estructura de red desarrollada.

- ConvNet: Partiendo de LeNet se creó una nueva red convolucional cambiando algunas de las capas, la arquitectura propuesta cuenta con la siguiente estructura: entrada – convolución – *pooling* – convolución – *pooling* – ReLu – FC – activación sigmoidal – salida. La figura 4-2c muestra la arquitectura implementada en la aplicación y la figura 4-2d muestra la estructura de red desarrollada.

4.3. Protocolo de pruebas

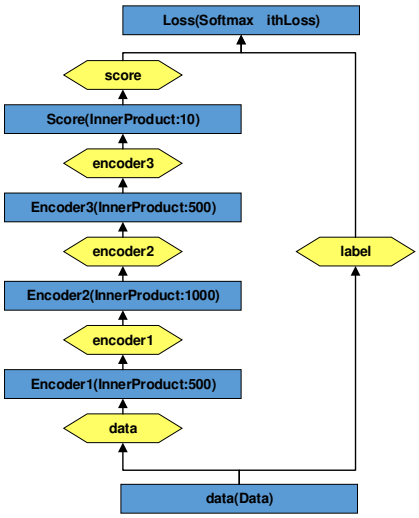
Cada una de las arquitecturas propuestas fueron desarrolladas con la GUI través de los paneles diseñados, tal como se muestra en las figuras 4-1 y 4-2. La interfaz permitió obtener

(a) DBN en el panel de arquitecturas



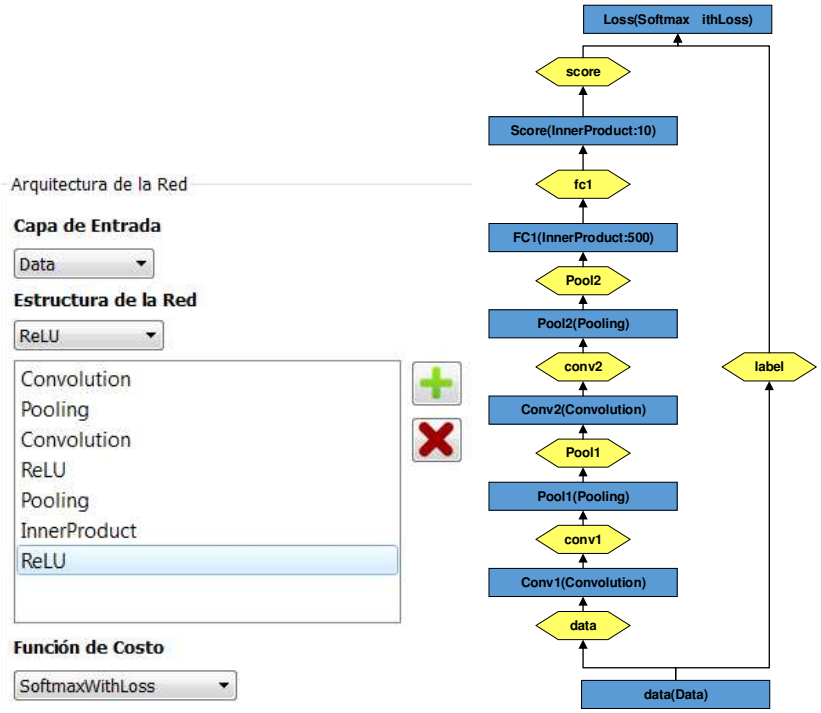
(b) DBN en caffe

(c) Autoencoder en el panel de arquitecturas

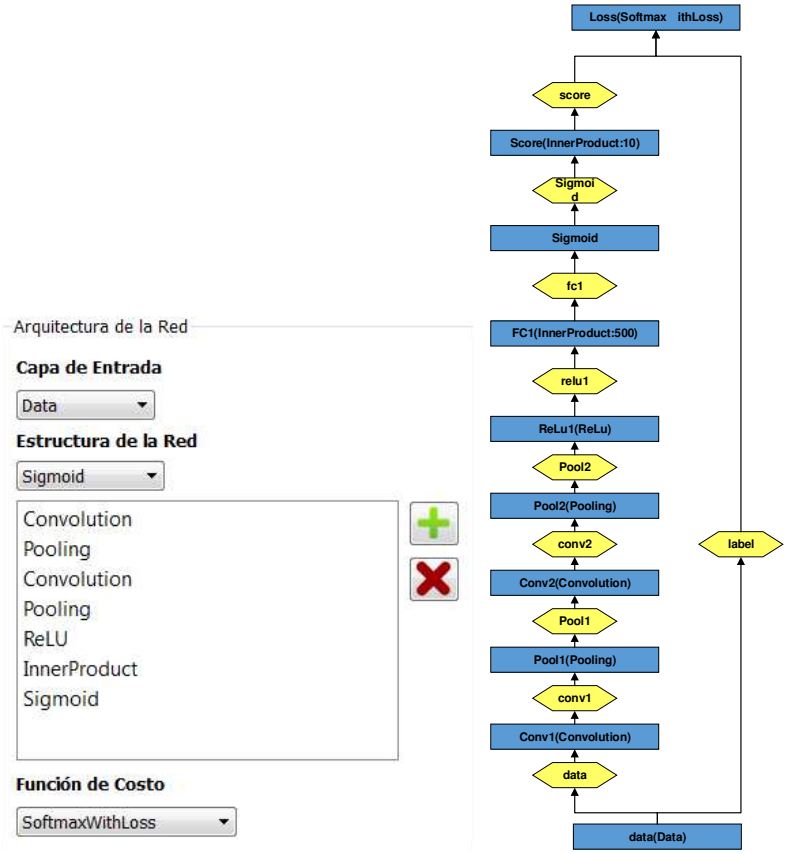


(d) Autoencoder Caffe

Figura 4-1: Diagramas de arquitecturas



(a) lenet en el panel de arquitecturas **(b)** lenet Caffe



(c) ConvNet en el panel de arquitecturas **(d)** ConvNet Caffe

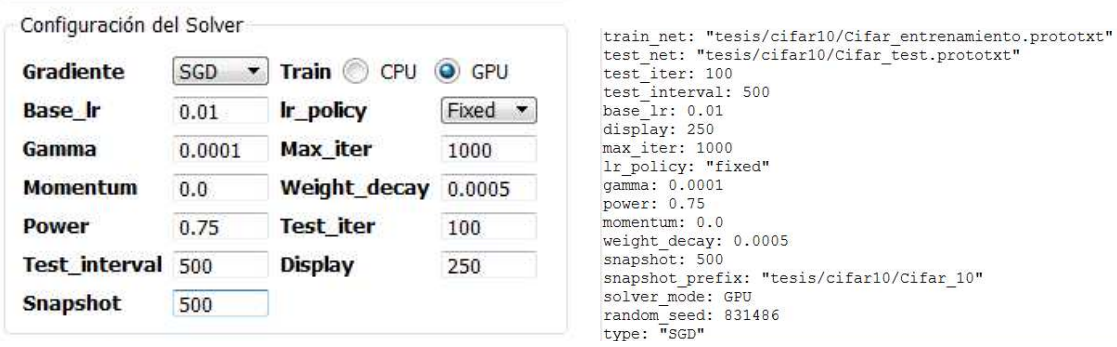
Figura 4-2: Diagramas de arquitecturas 2

dichas arquitecturas y generar el archivo prototxt para iniciar los procesos de entrenamiento de las *Deep Networks* a través de Caffe.

El protocolo de pruebas siguió los siguientes pasos:

1. Seleccionar base de datos.
2. Diseñar la arquitectura de la *Deep Network*.
3. Configurar los parámetros del *solver* para el entrenamiento.
4. Iniciar el entrenamiento de la arquitectura.
5. Repetir cada experimento cambiando el algoritmo de entrenamiento.

Cada una de las arquitecturas fue entrenada utilizando los algoritmos de aprendizaje disponibles en el *framework* y utilizando la misma configuración para el resto de parámetros del *solver* en Caffe, a excepción del número máximo de iteraciones el cual fue de 500 para el MNIST y 1000 para Cifar10. En la figura 4-3 se presenta la configuración del *solver* usando la aplicación y el *solver.prototxt* generado.



(a) *Solver* en la aplicación **(b)** *Solver* generado

Figura 4-3: Configuracion *Solver*

Una vez definidas las arquitecturas y los *solvers*, se procedió con el entrenamiento de cada sistema. El entrenamiento en la aplicación tiene 2 etapas, la primera realiza el entrena-

miento como tal de la red a la cual se optimiza su función de costo y la segunda es una fase de test la cual permite calcular la precisión total del sistema. En la tabla 4.1 y en la tabla 4.2 se presentan la organización de los entrenamientos según el set de entrenamiento, arquitectura y algoritmo de aprendizaje, y también el valor final de la función de costo, la precisión y el tiempo de entrenamiento.

Tabla 4.1: Resultados obtenidos en cada entrenamiento utilizando MNIST

Base de Datos	Arquitectura	Algoritmo	Precision	Loss	Tiempo (s)
MNIST	DBN	SGD	0.89	**0.32**	770
		AdaGrad	0.89	1.65	779.9
		AdaDelta	0.82	0.89	798.4
		Adam	0.87	0.75	731.9
		Nesterov	**0.9**	0.35	**351**
	AE	SGD	0.9	**0.4**	**204**
		AdaGrad	**0.91**	0.37	468
		AdaDelta	0.75	1.35	502
		Adam	0.89	0.59	460
		Nesterov	0.9	0.36	207
	CONV	SGD	0.97	0.11	**433.9**
		AdaGrad	**0.98**	**0.08**	511.2
		AdaDelta	0.77	1.32	519.1
		Adam	0.18	2.27	510.1
		Nesterov	0.97	0.12	435.3

Finalmente, se verifica el correcto funcionamiento de la ventana de vistas y validación para ello se seleccionaron 2 modelos, el primero fue el obtenido al realizar el entrenamiento del MNIST con LeNet y el segundo fue un modelo pre-entrenado de Caffe llamado CaffeNet, el cual fue entrenado utilizando la base de datos ImageNet y tiene la capacidad de identificar más de 500 clases. En la figura 4-4 se muestra el funcionamiento de la aplicación con el modelo pre-entrenado de Caffe ” CaffeNet”.

Para poder realizar la validación de una arquitectura en Caffe es necesario hacer unos pequeños cambios al .prototxt usado para el entrenamiento, esto con el fin de que Caffe reconozca el archivo como una arquitectura de prueba y no una de entrenamiento. Para este

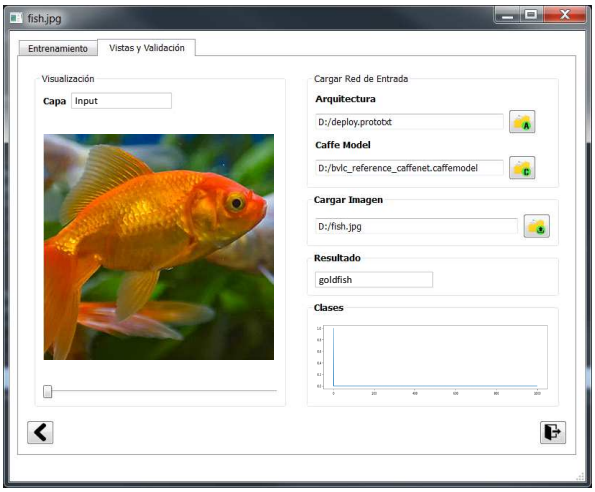
Tabla 4.2: Resultados obtenidos en cada entrenamiento utilizando CIFAR10

Base de Datos	Arquitectura	Algoritmo	Precision	Loss	Tiempo (s)
CIFAR10	DBN	SGD	NA	NA	NA
		AdaGrad			
		AdaDelta			
		Adam			
		Nesterov			
	AE	SGD	NA	NA	NA
		AdaGrad			
		AdaDelta			
		Adam			
		Nesterov			
	CONV	SGD	**0.52**	**1.64**	**1719.5**
		AdaGrad	0.12	2.29	2158
		AdaDelta	0.35	2.09	2179
		Adam	0.12	2.66	2151
		Nesterov	**0.52**	**1.64**	1833

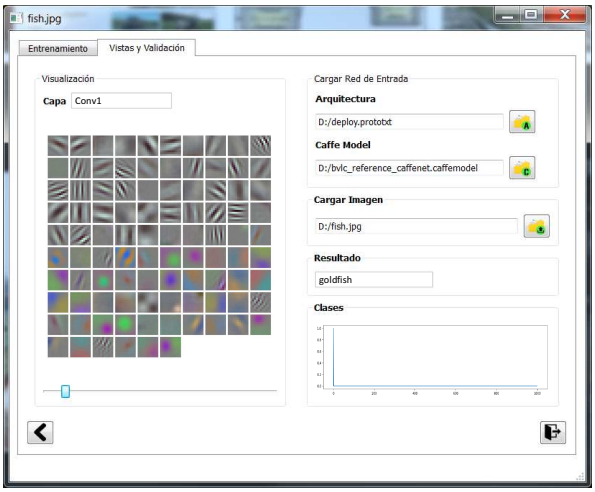
proposito se anexa un script que realiza esta operacion.

4.4. Análisis de Resultados

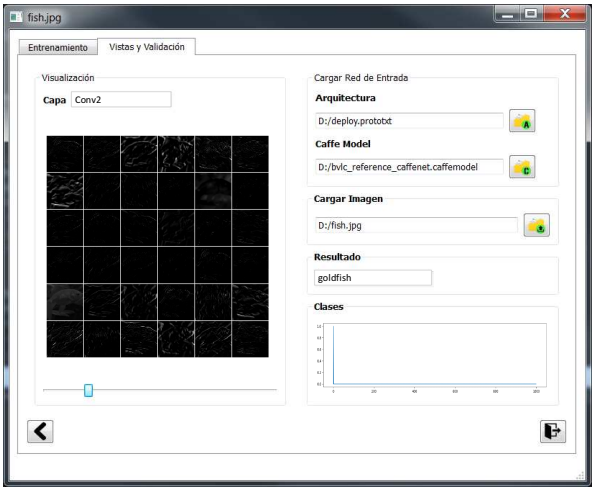
En esta sección se presenta el análisis de los resultados obtenidos para cada caso desde diferentes puntos de vista o comparación, por ejemplo, la relación arquitectura algoritmo y base de datos arquitectura. Cabe resaltar que el objetivo de este trabajo no es optimizar resultados, sino presentar una aplicación para el diseño y validación de arquitecturas *Deep Network* usando una plataforma de bajo costo de procesamiento heterogéneo.



(a) Capa de entrada



(b) Filtros de la capa Conv1



(c) Valor de los BLOBs a la salida de la capa Conv1

Figura 4-4: Prueba de la pestaña de validación

4.4.1. Análisis Base de datos - arquitectura - algoritmo

Como se muestra en la tabla 4.1, para la base de datos MNIST se utilizó 3 arquitecturas descritas anteriormente, y cada una de las arquitecturas fue entrenada utilizando los algoritmos disponibles en el *framework*.

Para la DBN, el mejor resultado en cuanto a precisión fue obtenido utilizando el algoritmo de Nesterov, con un valor de 90 %, aunque todos los algoritmos a excepción del AdaDelta estuvieron cercanos a ese valor; en cuanto a la función de costo los menores valores se obtuvieron con el SGD y el Nesterov, respectivamente; con respecto al tiempo de procesamiento el mejor resultado se obtuvó con el algoritmo Nesterov.

En cuanto el Autoencoder, a excepción del AdaDelta el resto de los algoritmos presentaron una precisión alrededor del 90 %; El menor valor de función de costo se obtuvó utilizando el algoritmo Nesterov, El SGD y AdaGrad tuvieron valores cercanos; con respecto al tiempo de procesamiento el SGD y el Nesterov presentan los mejores resultados.

Para la red convoluciónal el SGD, AdaGrad y Nesterov se obtuvó una precisión entre 97 % y 98 %, valor de la función de costo entre 0.08 y 0.12; el menor valor del tiempo de procesamiento lo obtuvieron los algoritmos SGD y Nesterov. En la gráfica 4-5 se observa que a pesar de que con el algoritmo Adam se obtuvó una precisión final de 18 %, al momento de la iteración 300 este alcanzo un umbral de 98 % de precisión, lo cual pudo haber sido causado por un desbordamiento de memoria por parte del dispositivo donde por cuestiones de seguridad elimina todas las tareas que se encuentran en ejecución, una alternativa para tratar este tipo de inconvenientes es utilizar uno de los *Snapshots* generados automáticamente por el *framework* Caffe durante el entrenamiento, el intervalo de iteraciones puede ser ajustado con el parámetro *Snapshot*.

Del análisis realizado sobre el MNIST se puede concluir que para la DBN la mejor alter-

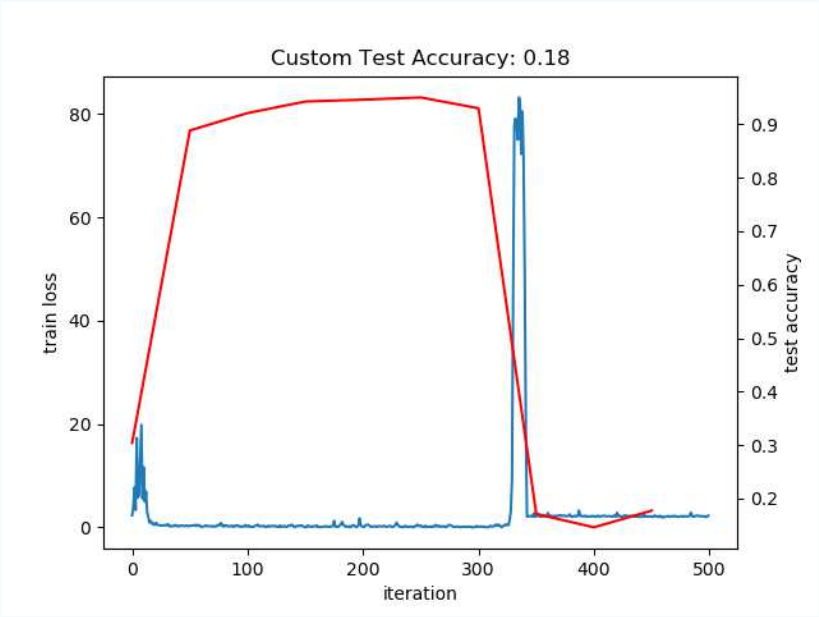


Figura 4-5: Entrenamiento usando algoritmo Adam. rojo: Precisión; azul: Función de costo.

nativa de entrenamiento es con el algoritmo Nesterov ya que a pesar de que tiene precisión y función de costo similares a las obtenidas con el SGD, el Nesterov emplea la mitad del tiempo de procesamiento del SGD. Para el AE, los algoritmos SGD y Nesterov presentan los mejores resultados en cuanto a precisión, función de costo y tiempo de procesamiento. Finalmente para la red convoluciónal el mejor algoritmo para el entrenamiento fue el AdaGrad, el cual presenta la mayor precisión y el menor valor de función de costo.

Se observa entonces que la arquitectura que mejor se adapta al problema de clasificación del MNIST es la red convoluciónal, aunque las otras arquitecturas implementadas también presentan resultados que resultan favorables desde el punto de vista de la facilidad y sencillez del diseño.

Para el CIFAR10 se realizó el mismo protocolo de pruebas con los cambios mencionados en la sección 4.2. En la tabla 4.2, para las arquitecturas DBN y AE se registraron “NA” en los campos de precisión, función de costo y tiempo; esto debido a que al momento de realizar el entrenamiento el sistema se volvía inestable sin presentar convergencia hacia alguna so-

lución. Estos estancamientos son causados por la complejidad de la base de datos, la cual requiere una red con mayor capacidad de procesamiento y mayor capacidad de abstracción de características.

Para la red convolucional los mejores resultados se obtuvieron con los algoritmos SGD y Nesterov, con los cuales se obtuvo una precisión del 52% y un valor final de función de costo de 1.64, la diferencia estuvo en el tiempo de procesamiento, donde el SGD obtuvo una diferencia mínima respecto al Nesterov resolviendo el problema en 1719.5 segundos y 1833 segundos respectivamente

Debido a la complejidad del CIFAR10, es requerido utilizar una arquitectura robusta y con una alta capacidad de cómputo y extracción de características como lo son las redes convolucionales. En la prueba realizada con la red convolucional no se obtuvieron resultados óptimos a comparación con los obtenidos utilizando el MNIST, esto debido a que la red a pesar de tener cierto nivel de complejidad sigue siendo simple, además de tener en cuenta que los parámetros de entrenamiento no fueron ajustados de manera óptima y el número de iteraciones es muy reducido comparado con el utilizado por Karen Simonyan (Simonyan y Zisserman, 2014), donde obtuvo una precisión del 92% utilizando 370.000 iteraciones. Sin embargo, no se descarta la posibilidad de obtener buenos resultados utilizando un Autoencoder correctamente entrenado, ya que el implementado en estas pruebas fue entrenado como una MLP debido a la dificultad que presenta Caffe para llevar a cabo el entrenamiento descrito en la sección 2.5.0.3.

Capítulo 5

Conclusiones y Trabajos Futuros

5.1. Observaciones Generales

Se implementaron cinco algoritmos de aprendizaje para el entrenamiento de redes neuronales tipo *Deep Networks*. Los algoritmos de aprendizaje implementados son: gradiente descendente estocástico, Adagrad, AdaDelta, Adam y Nesterov, los cuales están orientados a resolver el proceso de aprendizaje de las arquitecturas Deep Belief Network, Autoencoders y Redes Convolucionales. La implementación de este sistema de redes neuronales fue realizado sobre una sistema embebido de procesamiento gráfico Jetson TK1 de NVidia y a través del *framework* de *Deep Learning Caffe*.

Para determinar que tipo de arquitecturas y algoritmos de aprendizaje que podían ser implementadas sobre la Jetson TK1 y el framework Caffe se realizó una investigación la cual queda reportada en este documento. Adicionalmente, se hizo una exploración de la evolución de las redes neuronales tradicionales hacia las *Deep Networks* desde las MLP hasta las arquitecturas mas complejas conocidas como redes convolucionales.

Se diseño una aplicación software que propone una metodología para la creación, entrenamiento y validación de diferentes arquitecturas de una manera gráfica e intuitiva, que además permite utilizar un gran número capas y neuronas con diferentes funciones a través del *framework* Caffe, el cual es controlado con una interfaz desarrollada en Qt y enlazada usando PyCaffe.

Con la aplicación desarrollada y el conocimiento adquirido se afrontaron 2 problemas prácticos que consistían en la clasificación de imágenes y detección de objetos en imágenes, para lo cual se utilizaron las bases de datos MNIST y CIFAR10. La aplicación se utilizó de manera exitosa en el diseño y entrenamiento de las diferentes arquitecturas implementadas, donde se logró realizar diferentes pruebas variando los parámetros de entrenamiento y las capas que componían cada arquitectura. Tanto para el MNIST como para el CIFAR10 las redes convolucionales presentaron la mejor respuesta al entrenamiento con los algoritmos SGD y Nesterov, obteniendo los mejores valores en cuanto a precisión, valor final de la función de costo y el tiempo de procesamiento.

Las redes Convolucionales son una excelente alternativa en aplicaciones o soluciones a problemas que involucran campos de la visión artificial debido a su robustez y capacidad de extracción de características. Combinando diferentes tipos de capas se pueden crear redes potentes con fines específicos, y combinando arquitecturas se pueden obtener sistemas más complejos con aplicabilidad a problemas igualmente complejos, pero que requieren una alta capacidad de procesamiento y extracción de características.

Con la aplicación desarrollada y el conocimiento registrado en este libro se da el primer paso en la adopción y apropiación de una nueva tecnología que es tendencia a nivel mundial en el ámbito científico y de desarrollo. Estas tecnologías permiten abordar y resolver problemas de alta complejidad de una manera eficiente con respecto a otras tecnologías como las redes neuronales convencionales, las cuales presentan muchas limitaciones al momento de afrontar problemas de este tipo.

5.2. Trabajos Futuros

La aplicación desarrollada permite crear, entrenar y validar redes, sin embargo, estas 3 operaciones no son las únicas que se pueden realizar en Caffe. En cuanto al entrenamiento aun hace falta implementar una función que permita realizar el *fine tuning* de la red. Otra función que debe ser tenida en cuenta para una mejora de la aplicación es la capacidad de utilizar redes siamesas, estas son redes que surgen a partir de otra red, es decir que se podría tomar solo una parte de un modelo entrenado y adaptarlo en otro modelo conservando los pesos y parámetros, lo que es técnicamente conocido como *Transfer Learning*. Con estas 2 funciones se podría implementar un algoritmo para entrenar Autoencoders usando pycaffe.

En la actualidad existen un gran número de *frameworks* y tecnología a nivel de hardware que eliminan las dificultades que han existido en el pasado, esta aplicación podría ser implementada utilizando un framework diferente a Caffe, y ejecutándolo en plataformas diferentes a la tk1,con una mayor capacidad de cómputo y libertad al momento del diseño.

Referencias

Anderson, J. (2007). *Redes neuronales* (Alfaomega, Ed.). Alfaomega.

Arel, I., Rose, D. C., y Karnowski, T. P. (2010, noviembre). Research frontier: Deep machine learning–a new frontier in artificial intelligence research. *Comp. Intell. Mag.*, *5*(4), 13–18.

Baldi, P. (2012, 02 Jul). Autoencoders, unsupervised learning, and deep architectures. En I. Guyon, G. Dror, V. Lemaire, G. Taylor, y D. Silver (Eds.), *Proceedings of icml workshop on unsupervised and transfer learning* (Vol. 27, pp. 37–49). Bellevue, Washington, USA.

Bengio, Y. (2009, enero). Learning deep architectures for ai. *Found. Trends Mach. Learn.*, *2*(1), 1–127.

Bengio, Y., Lamblin, P., Popovici, D., y Larochelle, H. (2006). Greedy layer-wise training of deep networks. En *Proceedings of the 19th international conference on neural information processing systems* (pp. 153–160). Cambridge, MA, USA: MIT Press.

Bottou, L. (2012). Stochastic gradient descent tricks. En G. Montavon, G. B. Orr, y K.-R. Muller (Eds.), *Neural networks: Tricks of the trade (2nd ed.)* (Vol. 77, p. 421-436). Springer.

Caicedo, E., y Lopez, J. (2010). *Redes neuronales artificiales* (P. E. U. del Valle, Ed.). Programa Editorial Universidad del Valle.

Deng, L. (2011, October). An overview of deep-structured learning for information processing. En *Proc. asian-pacific signal & information proc. annual summit & conference (apsipa-asc)* (p. 1-14).

Duchi, J., Hazan, E., y Singer, Y. (2011, julio). Adaptive subgradient methods for online learning and stochastic optimization. *J. Mach. Learn. Res.*, *12*, 2121–2159.

Fischer, A., y Igel, C. (2012). An introduction to restricted boltzmann machines. En L. Álvarez, M. Mejail, L. G. Déniz, y J. C. Jacobo (Eds.), *Ciarp* (Vol. 7441, p. 14-36). Springer.

Gonzalez, J. R. H., y Sierra, F. J. C. (2000). *Redes neuronales artificiales: fundamentos, modelos y aplicaciones* (A.-W. Iberoamericana, Ed.). Addison-Wesley Iberoamericana.

Haykin, S. (1999). *Neural networks: A comprehensive foundation* (P. Hall, Ed.). Prentice Hall.

Hernandez, D. (2014). Meet the man google hired to make ai a reality. *https://www.wired.com/2014/01/geoffrey-hinton-deep-learning/*.

Hinton, G. E., Osindero, S., y Teh, Y.-W. (2006, julio). A fast learning algorithm for deep belief nets. *Neural Comput.*, *18*(7), 1527–1554.

Hinton, G. E., y Salakhutdinov, R. R. (2006, julio). Reducing the dimensionality of data with neural networks. *Science*, *313*(5786), 504-507.

Jia, Y., Shelhamer, E., Donahue, J., Karayev, S., Long, J., Girshick, R., ... Darrell, T. (2014). Caffe: Convolutional architecture for fast feature embedding. En *Proceedings of the 22nd acm international conference on multimedia* (pp. 675–678). New York, NY, USA: ACM.

Kingma, D. P., y Ba, J. (2014). Adam: A method for stochastic optimization. *CoRR*, *abs/1412.6980*.

LeCun, Y., y Bengio, Y. (1998). The handbook of brain theory and neural networks. En M. A. Arbib (Ed.), *The handbook of brain theory and neural networks* (pp. 255–258). Cambridge, MA, USA: MIT Press.

LeCun, Y., Bengio, Y., y Hinton, G. (2015, 27 de mayo). Deep learning. *Nature*, *521*(7553), 436–444.

Lecun, Y., Bottou, L., Bengio, Y., y Haffner, P. (1998). Gradient-based learning applied to document recognition. En *Proceedings of the ieee.*

Le Roux, N., y Bengio, Y. (2008, junio). Representational power of restricted boltzmann machines and deep belief networks. *Neural Computation*, *20*(6), 1631–1649.

Mo, D. (2012). A survey on deep learning: one small step toward AI - Mo. *A survey on deep learning: one small step toward AI - Mo*.

Nielsen, M. (2015). *Neural networks and deep learning* (M. Nielsen, Ed.). Neural Networks and Deep Learning.

Ruder, S. (2016). An overview of gradient descent optimization algorithms. *CoRR*, *abs/1609.04747*.

Ruiz, C. A., y Basualdo, M. S. (2004). *Redes neuronales artificiales* (L. Noriega, Ed.). Limusa Noriega.

Salakhutdinov, R. (2015). *Learning deep generative models* (Tesis Doctoral no publicada). University of Toronto, Toronto, Ont., Canada, Canada. (AAINR61080)

Salakhutdinov, R., y Hinton, G. (2009, julio). Semantic hashing. *Int. J. Approx. Reasoning*, *50*(7), 969–978.

Salakhutdinov, R., y Murray, I. (2008). On the quantitative analysis of deep belief networks. En *Proceedings of the 25th international conference on machine learning* (pp. 872–879). New York, NY, USA: ACM.

Simonyan, K., y Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *CoRR*, *abs/1409.1556*.

Sutskever, I., Martens, J., Dahl, G., y Hinton, G. (2013, mayo). On the importance of initialization and momentum in deep learning. En S. Dasgupta y D. Mcallester (Eds.), *Proceedings of the 30th international conference on machine learning (icml-13)* (Vol. 28, p. 1139-1147). JMLR Workshop and Conference Proceedings.

Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., y Manzagol, P.-A. (2010, diciembre). Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *J. Mach. Learn. Res.*, *11*, 3371–3408.

Wang, H., y Raj, B. (2015). A survey: Time travel in deep learning space: An introduction to deep learning models and how deep learning models evolved from the initial ideas. *CoRR*, *abs/1510.04781*.

White, P. R. S. (2004). Rational unified process best practices for software [Manual de software informático].

Zeiler, M. D. (2012). ADADELTA: an adaptive learning rate method. *CoRR*, *abs/1212.5701*.