Instituto Tecnológico
y de Estudios Superiores de Occidente

Reconocimiento de validez oficial de estudios de nivel superior según acuerdo secretarial 15018, publicado en el Diario Oficial de la Federación del 29 de noviembre de 1976.

Departamento de Electrónica, Sistemas e Informática

Maestría en Sistemas Computacionales

Biblioteca de Evaluación y Entrenamiento de Redes Neuronales Artificiales Implementada en Rust y OpenCL

TRABAJO RECEPCIONAL que para obtener el **GRADO** de
MAESTRO EN SISTEMAS COMPUTACIONALES

Presenta: **VICTOR RAÚL ESCAMILLA OCHOA**

Director **MTRO. PEDRO ARTURO CORNEJO TORRES**

Tlaquepaque, Jalisco. Enero de 2021.

AGRADECIMIENTOS

Primeramente, quiero agradecer a mi esposa, Daleth. Sin tu apoyo, este trabajo no sería el mismo. Sin esas tardes de risas y distracciones. Sin tus plantas en la terraza. Sin nuestros gatos. Agradezco todas las palabras de aliento y el soporte que me diste cuando estaba cansado. Porque todo es más fácil si lo hacemos juntos. Te amo.

Agradezco también a todos los docentes del Iteso que me apoyaron durante estos dos años de trabajo, especialmente al Mtro. Pedro Arturo Cornejo Torres por todos sus consejos y asesoría durante este proyecto, al CONACYT por la beca recibida con número 498401, al ITESO por los descuentos aplicados para apoyo a la industria tecnológica y a Intel por el apoyo recibido por parte del sistema de tutorías.

Finalmente, quiero agradecer a la comunidad de Rust por ser un espacio incluyente y motivador en donde personas de todo el mundo contribuyen a desarrollar herramientas para el futuro de los sistemas computacionales.

DEDICATORIA

Dedico este trabajo a mi madre, que alimentó la curiosidad que tenía desde pequeño por la ciencia; a mi padre, que me enseñó a ser responsable, a estar a tiempo; y finalmente a mi hermano, quien ha llenado de alegría mi vida desde que *vivíamos en una nave espacial*.

Noemí, Martín y César. Los amo.

RESUMEN

El presente trabajo consiste en el desarrollo de una *framework* para el diseño, evaluación y entrenamiento de Redes Neuronales con soporte de OpenCL para acelerar los cálculos mediante del uso de la Unidad de Procesamiento Gráfico (GPU por sus siglas en inglés) y desarrollado en Rust, un lenguaje de alto nivel diseñado para la programación de sistemas con un desempeño equiparable a C y C++.

En primer lugar, se exploran algunas de las librerías de *machine learning* más utilizadas en la industria actualmente (Tensorflow, Theano, CNTK, PyTorch y Keras), así como otros proyectos de *machine learning* más recientes propuestos por la comunidad de Rust (Rust Tensorflow, Juice, Rust-autograd y tch-rs). De este análisis se obtiene una comparación breve entre las soluciones existentes que sirve para establecer puntos de mejora en el estado del arte del área de *machine learning* en Rust.

Posteriormente se analizan las bases teóricas sobre las cuales se fundamenta el modelado, entrenamiento y evaluación de redes neuronales: los perceptrones, las funciones de activación, los algoritmos detrás del entrenamiento de una red neuronal (*feedforward-backpropagation*), técnicas para el procesamiento de grandes cantidades de datos, generalización de una red neuronal y el sobreentrenamiento. Se detallan también las características de OpenCL y Rust que permiten implementar dichas bases teóricas a lo largo de este trabajo.

Con la finalidad de establecer un marco de referencia que ayude a determinar el éxito de la biblioteca que se desarrolla en este trabajo, se toman dos problemas bastante populares en el área de *machine learning* (compuerta XOR y detección de dígitos escritos a mano a partir del dataset MNIST), cuya solución radica en el uso de redes neuronales clásicas. Ambas soluciones son implementadas con la biblioteca propuesta en este trabajo y se procede a comparar el desempeño obtenido contra los mismos modelos de redes neuronales implementados con Tensorflow y Keras.

Finalmente, se proporciona una discusión a partir de los resultados obtenidos durante los experimentos realizados y se analizan posibles mejoras en el desempeño de la biblioteca, así como trabajo a futuro.

TABLA DE CONTENIDO

LISTA DE FIGURAS

LISTA DE TABLAS

LISTA DE ECUACIONES

LISTA DE ACRÓNIMOS Y ABREVIATURAS

AI	*Artificial Intelligence*
ANN	*Artificial Neural Networks*
API	*Application Programming Interface*
CPU	*Central Processing Unit*
FPGA	*Field-Programmable Gate Array*
GC	*Garbage Collector*
GPU	*Graphics Processing Unit*
LSTM	*Long Short-Term Memory*
MKL	*Math Kernel Library*
MLP	*Multi Layer Perceptron*
MSE	*Mean Squared Error*
ReLU	*Rectified Linear Unit*
SDK	*Software Development Kit*
SGD	*Stochastic Gradient Descent*
SIMD	*Single Instruction Multiple Data*

1. INTRODUCCIÓN

En este capítulo se presentan brevemente los antecedentes de las soluciones basadas en redes neuronales artificiales, así como la justificación para crear una nueva biblioteca para el modelado, entrenamiento y evaluación de estas, y los motivos para emplear Rust y OpenCL para el desarrollo.

1.1. Antecedentes

El uso comercial de soluciones de *machine learning* se ha disparado en los últimos años y las Redes Neuronales Artificiales (ANN por sus siglas en inglés) juegan un papel muy importante. Productos tales como los vehículos autónomos de Tesla, asistentes personales como Siri, Cortana y Alexa, controles de logística como Amazon Go; no serían rentables sin la interacción con algún tipo de ANN. Empresas tecnológicas importantes como lo son Google, Amazon, Facebook, Microsoft y Oracle, participan activamente ya sea en la investigación y creación de herramientas de software para Inteligencia Artificial (AI por sus siglas en inglés) o en el diseño de infraestructura para facilitar su desarrollo y consumo. El entender cómo funcionan las ANN y cómo aplicarlas en la resolución de problemas actuales está cada vez más presente en el desarrollo de los estudiantes de Sistemas Computacionales.

1.2. Justificación

El desarrollar desde cero una biblioteca para el modelado, evaluación y entrenamiento de redes neuronales es una tarea muy ambiciosa, sobre todo si se toma en cuenta que existen muchas soluciones para este problema desarrolladas no solo por la comunidad Open Source, sino también por grandes empresas tecnológicas como lo son Microsoft, Google y Facebook. Sin embargo, es por este mismo hecho que no muchos estudiantes tienen la oportunidad de realizar proyectos de este tipo.

El contar con una biblioteca de redes neuronales desarrollada y mantenida por estudiantes del ITESO, permite a sus contribuidores explorar de primera mano diferentes maneras de resolver los problemas que se presentan durante la implementación de los algoritmos relacionados a las redes neuronales y profundizar en conceptos que suelen pasar desapercibidos al utilizar las bibliotecas existentes. Un ejemplo de estos conceptos es la labor que representa traducir dichos algoritmos de *machine learning* para ser procesados por un GPU de forma eficiente, algo que se explora durante el desarrollo de este proyecto al agregar el soporte de OpenCL.

1.3. Problema

El proceso de entrenamiento de redes neuronales implica una gran cantidad de operaciones matemáticas que resultan exhaustivas para la Unidad Central de Procesamiento (CPU por sus siglas en inglés); operaciones que son repetitivas y altamente paralelizables. Es por este motivo que la gran mayoría de bibliotecas dedicadas al modelado y entrenamiento de redes neuronales cuentan con soporte para realizar el cómputo en el GPU.

Si bien existen múltiples proyectos para el modelado y evaluación de redes neuronales así como otras tecnologías de *machine learning*, la gran mayoría se limita a dar soporte a GPUs de Nvidia[1], limitando

el desarrollo de aplicaciones a un hardware específico. Gracias a tecnologías como OpenCL, es posible escribir programas que permitan delegar cálculos repetitivos y paralelizables para ser ejecutados en el GPU independientemente del hardware que se utilice e incluso correr en el CPU haciendo uso de instrucciones *Single Instruction Multiple Data* (SIMD) en el procesador.

El propósito de esta investigación es el establecer las bases para la creación de una nueva biblioteca para el entrenamiento y la evaluación de redes neuronales artificiales con soporte para OpenCL y que quedará disponible para estudiantes y académicos del ITESO.

1.4. Objetivos

1.4.1. Objetivo General:

Desarrollar una biblioteca para el modelado y entrenamiento de redes neuronales artificiales con soporte de redes *feedforward-backpropagation*, implementada en Rust y que sea capaz de acelerar el entrenamiento mediante el uso del GPU a través de OpenCL. La biblioteca debe alcanzar una precisión equiparable a la obtenida al modelar los mismos problemas con los mismos hiperparámetros en Keras. Esto con el propósito de establecer el código base para el desarrollo de una nueva solución de *machine learning* a disposición de estudiantes y académicos del ITESO.

1.4.2. Objetivos Específicos:

- Diseñar una Interfaz de Programación de Aplicaciones (API por sus siglas en inglés) que sea clara, expresiva y fácil de extender el soporte a redes neuronales más complejas.
- Desarrollar una biblioteca auxiliar que contenga todos los elementos de álgebra lineal necesarios para la implementación de la red neuronal, y que cuente con soporte para realizar el procesamiento a través del CPU o GPU.
- Proveer un mecanismo para inspeccionar el estado de la red neuronal durante la etapa de entrenamiento, de manera que permita evaluar la precisión de la red neuronal.
- Implementar dos problemas clásicos de redes neuronales (MNIST y XOR) con Keras así como con la biblioteca desarrollada para este trabajo y comparar la precisión obtenida en ambas soluciones.

1.5. Novedad científica, tecnológica o aportación

Un gran beneficio que surge a partir de la creación de una biblioteca de redes neuronales desde cero es la posibilidad de elegir el lenguaje de programación a utilizar. Para este trabajo se decide implementar la solución en Rust, un lenguaje de programación de sistemas muy expresivo que cuenta con un desempeño equiparable al de C/C++, por lo que es un lenguaje adecuado para resolver este tipo de problemas.

En los últimos años, la aceptación de Rust en distintas áreas del desarrollo de software ha crecido rápidamente debido a su modelo único de manejo de memoria, su expresividad y su alto desempeño. Desde el 2016 y hasta el momento, Rust ha sido el lenguaje más votado en la categoría "*Most loved language*" (El lenguaje más amado) en la Encuesta Anual de Desarrolladores organizada por Stack Overflow[2], demostrando el alto interés de la comunidad de desarrolladores de software de distintas áreas en construir herramientas en este lenguaje. Considerando que Rust fue publicado oficialmente en el 2015, esto es un logro excepcional.

Muchos proyectos Open Source escritos originalmente en C o C++ están siendo reescritos en Rust paulatinamente por la comunidad, logrando mejoras en la velocidad y consumo de memoria de las aplicaciones[3]. A pesar de esto, Rust sigue siendo un lenguaje relativamente nuevo, por lo que muchas bibliotecas externas que forman parte fundamental del ecosistema de otros lenguajes de programación de sistemas se encuentran aún en etapas inmaduras o simplemente no existen. Un gran ejemplo de esto es la falta de bibliotecas para el desarrollo de soluciones en el área de *machine learning*.

Actualmente los proyectos enfocados en el modelado y entrenamiento de redes neuronales desarrollados en Rust están incompletos, abandonados o no cuentan con soporte para agilizar el procesamiento mediante el GPU. Al implementar este trabajo en Rust se pretende abrir el camino para más desarrollos en el ámbito de *machine learning* en dicho lenguaje, que cada vez va ganando más fuerza en la industria.

2. ESTADO DEL ARTE O DE LA TÉCNICA

En esta sección, se listan algunas de las herramientas que se utilizan actualmente para el desarrollo de redes neuronales artificiales en la industria, así como distintos proyectos que han surgido ante la necesidad de una biblioteca enfocada al desarrollo de soluciones de *machine learning* nativamente en Rust.

2.1. TensorFlow

Más que una biblioteca, Tensorflow es una plataforma open source para la construcción y producción de aplicaciones de *machine learning* desarrollada por Google. Cuenta con un ecosistema bastante sólido de herramientas y librerías, así como una comunidad de desarrollo muy amplia. Actualmente es el proyecto con mayor actividad y presencia entre todas las bibliotecas que se describen en este trabajo[4].

Tensorflow fue desarrollado en C++ y cuenta con APIs oficiales tanto para Python como para C++, aunque tiene soporte no oficial para otros lenguajes como lo son Java, Go e incluso Rust, como se analizará más adelante. Las aplicaciones desarrolladas con Tensorflow son compatibles con Windows, Linux, MacOS, iOS, Android y otros dispositivos móviles.

El procesamiento en Tensorflow se divide en dos fases: fase de construcción y fase de cómputo. Durante la fase de construcción, se definen todas las operaciones a realizar mediante la creación de un grafo en donde cada nodo representa una operación con sus respectivos datos de entrada y datos de salida, y cada arista del grafo representa un movimiento de dichos datos a los siguientes nodos del grafo. Durante la fase de cómputo, Tensorflow toma el grafo definido previamente, lo divide en varias secciones independientes y optimizadas para ser evaluadas en paralelo y se procede a ejecutar de forma distribuida entre los distintos dispositivos disponibles que pueden ser tanto CPU como GPU (CUDA). Dichos dispositivos pueden encontrarse distribuidos entre varios servidores de cómputo.

Una característica que distingue a Tensorflow es que su modelo basado en grafos es bastante flexible, lo que permite implementar una amplia variedad de redes neuronales, así como desarrollos en otros dominios de *machine learning*.

2.2. Theano

Theano es una biblioteca de Python para la evaluación optimizada de expresiones matemáticas de álgebra multidimensional desarrollada en la Universidad de Montreal donde es ampliamente utilizado para llevar las asignaturas de *machine learning* así como para la publicación de múltiples investigaciones en el campo. Theano es básicamente un compilador de expresiones matemáticas que permite al desarrollador diseñar cualquier modelo de *machine learning[5]*.

De forma similar a Tensorflow, Theano permite al usuario describir el modelo como un grafo de cómputo y posteriormente procede a procesar dicho grafo. El procesamiento en Theano consiste en 5 etapas: Canonicalización, estabilización, especialización, transferencia a GPU y generación de código. Durante la

fase de canonicalización, Theano transforma el grafo de entrada del usuario a una forma estándar, detectando y removiendo expresiones duplicadas. En la fase de estabilización, Theano reemplaza llamadas a funciones matemáticas que puedan causar problemas de estabilidad numérica (por ejemplo, expresiones que tienden a infinito) por aproximaciones numéricas. La siguiente fase, especialización, se encarga de sustituir expresiones matemáticas por implementaciones más rápidas que generen los mismos resultados. Un ejemplo de esto es convertir operaciones de matrices a sus equivalentes en GEMM. Theano cuenta con soporte opcional para habilitar el cómputo en GPU, durante la fase de transferencia a GPU se reemplazan todas aquellas operaciones que se realizan en el *host* por operaciones equivalentes en el GPU. Finalmente, durante la fase de generación de código, Theano transforma las expresiones resultantes del grafo de cómputo en código ya sea de C o CUDA, lo compila y lo carga de forma dinámica en módulos de Python.

Theano fue publicado en el 2008, lo que lo convierte en la biblioteca con más tiempo en el mercado entre las que se mencionan en este trabajo. Hasta el momento, Theano continúa recibiendo actualizaciones y sigue siendo relevante en el ámbito de machine learning.

2.3. Microsoft Cognitive Toolkit (CNTK)

Escrita en C++ y publicada oficialmente en el 2015, CNTK es la propuesta open source de Microsoft para el desarrollo de soluciones comerciales deep learning. A diferencia del resto de bibliotecas descritas en esta sección, CNTK se enfoca principalmente en el modelado, entrenamiento y evaluación de redes neuronales[6]. Con CNTK es posible trabajar con redes neuronales feedforward, redes neuronales convolucionales, redes neuronales recurrentes, *Long Short-Term Memory* (LSTM), entre otras y se basa completamente en el algoritmo *Stochastic Gradient Descent* (SGD). CNTK cuenta con una API disponible en varios lenguajes como Python, C++ y C#.

CNTK también puede ser utilizado para entrenar y evaluar soluciones de *machine learning* a través de su propio lenguaje de modelado de redes neuronales llamado BrainScript. Con BrainScript es posible definir prácticamente cualquier tipo de red neuronal utilizando expresiones, variables, funciones y otros conceptos de programación en una sintaxis similar a la de un lenguaje de *scripting*. Una gran ventaja de BrainScript es que permite mantener el código relacionado al modelo de la red neuronal completamente separado del código relacionado a la distribución y ejecución de la red neuronal. Esto favorece la portabilidad de modelos entre proyectos.

De manera similar a Tensorflow y Theano, en CNTK primero es necesario realizar la representación de la red neuronal como un grafo en el que cada hoja representa un valor de entrada o parámetro de la red y cada nodo representa una operación a realizar sobre sus nodos hijos, después internamente se realizan ciertas optimizaciones de acuerdo con el grafo obtenido y finalmente se procede a realizar el cómputo. CNTK puede realizar el procesamiento tanto en CPU como en GPU (CUDA) así como distribuir la carga de trabajo entre varios servidores. Hasta el momento no cuenta con soporte para OpenCL.

2.4. PyTorch

PyTorch es una biblioteca open source desarrollada por Facebook para Python y con soporte para C++ para el desarrollo de soluciones de *machine learning*. Su propósito principal es otorgar al desarrollador una biblioteca que sea capaz de realizar cómputos relacionados a *machine learning* de forma rápida sin comprometer la usabilidad de esta. Tanto Tensorflow, CNTK y Theano favorecen la velocidad de cómputo

y la flexibilidad para implementar distintos algoritmos de *machine learning* sobre la simplicidad de la API, mientras que PyTorch promete mantener un equilibrio entre velocidad de cómputo y facilidad de uso.

Otras bibliotecas analizadas en este trabajo dividen su modo de operación en varias fases, las cuales consisten en su mayor parte en tres operaciones: definir un grafo de cómputo, optimizarlo y posteriormente realizar el cómputo de dicho grafo. Este modo de operación permite tener visibilidad de todas las operaciones a realizar en el modelo a entrenar, siendo posible así identificar optimizaciones tempranas que de otra forma no sería posible. Sin embargo, este beneficio se paga con mayor tiempo de compilación y dificultad al momento de depurar errores en los modelos a entrenar. Estas desventajas se vuelven mucho más evidentes durante las fases preliminares de un proyecto de *machine learning*.

En PyTorch no se define un grafo de cómputo, sino que se realizan las operaciones de forma inmediata. Para conseguir un cómputo eficiente, PyTorch acelera el procesamiento mediante el uso de GPU (CUDA) y utiliza un algoritmo llamado diferenciación automática, el cual consiste en determinar la derivada exacta de una función dada en tiempo constante (O(1)). La combinación de GPU y diferenciación automática hacen de PyTorch una biblioteca con un desempeño que compite con el resto de bibliotecas mencionadas en este trabajo[7], haciendo de PyTorch una excelente opción para el desarrollo de prototipos e investigaciones preliminares a la producción de soluciones de *machine learning*.

2.5. Keras

Una biblioteca de alto nivel para redes neuronales escrita en Python. Tal como dice en su sitio web, Keras es una interfaz desarrollada para humanos, no máquinas[8]. La API de Keras provee distintas abstracciones de alto nivel y una semántica muy expresiva para la construcción de modelos de *machine learning* de forma clara. Dichas abstracciones sirven como interfaz entre el desarrollador y otras bibliotecas como Theano, Tensorflow y CNTK, las cuales corren en segundo plano y son las responsables de realizar todas las operaciones de bajo nivel (creación de grafos de cómputo, optimizaciones, distribución de carga en GPU, etcétera).

Debido a su facilidad de uso y a que es compatible con las bibliotecas más populares en la comunidad, Keras se ha ganado un lugar esencial en el nicho, siendo una herramienta predilecta para el desarrollo y experimentación de prototipos durante la investigación de modelos de *machine learning*.

2.6. Soluciones existentes en Rust

Características como el alto desempeño de Rust, su capacidad de realizar operaciones de bajo nivel, su seguridad y eficiencia en el manejo de memoria y concurrencia, hacen de Rust un lenguaje bastante prometedor para el desarrollo de herramientas en el área de *machine learning*. Sin embargo, actualmente el ecosistema para proyectos de machine learning nativos en Rust se encuentra en sus fases preliminares. Esto ha sido una gran área de oportunidad para la comunidad open source, que se ha dedicado a cubrir los huecos existentes en el ámbito creando nuevas herramientas nativas para Rust o contribuyendo en la migración de proyectos existentes hacia este nuevo lenguaje. En esta sección se exploran los proyectos más relevantes en el ecosistema de machine learning desarrollados en Rust de acuerdo con la comunidad[9].

2.6.1. Rust TensorFlow

Rust Tensorflow es un proyecto open source cuyo propósito es el proporcionar *bindings* de la API de Tensorflow en Rust y que permite realizar la evaluación en modelos previamente entrenados con TensorFlow, pero de forma idiomática en Rust. Para utilizar este paquete es necesario compilar el proyecto teniendo instalado Python, la biblioteca de Tensorflow con todas sus dependencias externas, y opcionalmente CUDA para el procesamiento de GPU. Actualmente no cuenta con soporte para OpenCL.

Si bien, la posibilidad de utilizar una biblioteca tan poderosa como lo es TensorFlow de forma idiomática en Rust resulta bastante prometedora, en la actualidad este paquete se encuentra en una etapa de desarrollo temprana, no cuenta con una API estable y su soporte actual no permite modelar soluciones de machine learning, sino que se limita a la ejecución de modelos previamente entrenados[10].

2.6.2. Juice

Juice (previamente conocido como Leaf) es un proyecto open source dedicado al desarrollo de una framework que contiene herramientas para implementar distintas técnicas de machine learning en Rust. En cuanto a redes neuronales, Juice cuenta con soporte para redes clásicas de perceptrón multicapa y redes convolucionales, empleando el algoritmo SGD. Juice es capaz de procesar el entrenamiento mediante el uso de GPU con dispositivos compatibles NVIDIA ya que se puede compilar con soporte para CUDA. El proyecto sigue recibiendo actualizaciones ocasionalmente, pero sus desarrolladores principales abandonaron el proyecto en el 2016[11]. Actualmente, Juice no tiene soporte para OpenCL.

2.6.3. rust-autograd

Rust-autograd es un proyecto desarrollado completamente en Rust enfocado en el desarrollo de una librería de modelado y evaluación de grafos de cómputo, similar a Tensorflow en ese sentido. El propósito de Rust-autograd es demostrar que Rust es capaz de realizar este tipo de operaciones relacionadas a machine learning de manera eficiente y con una semántica simple[12]. Internamente, Rust-autograd emplea una biblioteca llamada ndarray para realizar las operaciones de álgebra lineal en arreglos n-dimensionales. La biblioteca ndarray es bastante utilizada en otros proyectos de Rust, y es considerada equivalente a numpy en Python. En cuanto a redes neuronales, es posible implementar redes feedforward, convolucionales y LSTM.

Actualmente rust-autograd no cuenta con soporte para cómputo en GPU, pero tiene soporte opcional para Intel MKL (*Math Kernel Library*) a través de ndarray. MKL es una biblioteca desarrollada por Intel y agnóstica al lenguaje dedicada a optimizar operaciones matemáticas aprovechando las instrucciones SIMD, así como otras operaciones exclusivas de procesadores Intel.

2.6.4. tch-rs

El proyecto tch-rs consiste en la creación de *bindings* para Rust de la librería de C++ de PyTorch (libtorch 1.6). El propósito de esta colaboración es el mantener la API lo más cercana a la librería original de C++, por lo que el resultado es una biblioteca que no coincide con las prácticas idiomáticas de Rust. Sin embargo, esto no ha evitado que la comunidad siga apoyando a este desarrollo, y en la actualidad es el proyecto con las contribuciones más constantes y recientes de todas las soluciones de Rust listadas en este trabajo[13].

A diferencia de Rust-tensorflow, tch-rs no solo permite ejecutar modelos previamente entrenados, sino que cuenta también con las APIs de libtorch para el modelado y entrenamiento de soluciones de machine learning, lo que lo convierte en el proyecto de *bindings* más completo de Rust hasta el momento.

Tabla 1. Comparación de bibliotecas existentes enfocadas al modelado y evaluación de redes neuronales.

Biblioteca	**Implementación**	**API**	**CUDA**	**OpenCL**
TensorFlow	C++	Python, C++	Sí	-
Theano	Python	Python	Sí	-
CNTK	C++	Python, C++, C#, BrainScript	Sí	-
PyTorch	C++ (libtorch)	Python	Sí	-
Keras	Python	Python	Sí	-
Rust-tensorflow	C++ (tensorflow)	Rust	Sí (parcialmente)	-
Juice	Rust	Rust	Sí	-
Rust-autograd	Rust	Rust	-	-
tch-rs	C++ (libtorch)	Rust	Sí	-

3. MARCO TEÓRICO/CONCEPTUAL

En este capítulo se presentan las bases teóricas y conceptuales sobre los algoritmos detrás del entrenamiento y evaluación de redes neuronales artificiales, sus aplicaciones y las herramientas elegidas para el desarrollo.

3.1. Redes Neuronales Artificiales

Las Redes Neuronales Artificiales son modelos matemáticos que permiten aproximar casi cualquier función matemática en base a un conjunto de entradas y han demostrado ser de particular beneficio para resolver problemas de clasificación. Para comprender cómo es que funcionan, es necesario retroceder en el tiempo y analizar la estructura que inspiró el desarrollo de estas: el perceptrón.

3.1.1. Perceptrón

El perceptrón surge como un modelo de toma de decisiones que se basa en la forma en que las neuronas biológicas se activan o desactivan para interactuar entre sí. Si bien las neuronas biológicas funcionan de manera completamente diferente, los perceptrones fueron una idea innovadora en su momento y contribuyeron a establecer las bases de los modelos de redes neuronales artificiales que existen en la actualidad.

El perceptrón, en la interpretación moderna, es una función la cual recibe uno o más valores de entrada y a partir de dichos valores genera un único valor de salida el cual puede ser 0 o 1. En otras palabras, se puede decir que el perceptrón es capaz de tomar la decisión de activar (1) o desactivar (0) el valor de salida dependiendo de los valores que se reciben en la entrada. Para lograr esto, un perceptrón es modelado como un grafo con un único nodo el cual tiene varias aristas de entrada (x) y una única arista de salida (z). Cada arista de entrada tiene establecido un peso (w), el cual determina qué tanto afecta el valor de dicha arista a la activación de salida del perceptrón. Esta representación básica se puede observar en la Ilustración 1.

Para tomar la decisión de responder 0 ó 1, el perceptrón multiplica cada valor de entrada por su peso correspondiente, realiza la suma del resultado de cada arista y si la suma es menor o igual a cierto valor predefinido el resultado será 0, de forma contraria el perceptrón regresará 1. A este valor predefinido se le conoce también como bias (b), y determina qué tan sensible es el perceptrón para cambiar su resultado.

El combinar varios perceptrones de forma que la salida de uno o varios perceptrones se conviertan en la arista de entrada de otros, permite realizar modelos que serían capaces de tomar decisiones más complejas. Estos modelos con múltiples perceptrones son un ejemplo de red neuronal conocido como *Multi Layer Perceptron* (MLP) en donde cada perceptrón representa una neurona. Esta representación se puede observar en la Ilustración 2.

La idea detrás del modelo de perceptrones es que debido a que cualquier cambio en los valores de los pesos de cada arista y bias en la red afectan directamente al resultado del perceptrón, debería ser posible

aproximar cualquier función matemática siempre y cuando se encuentren los valores ideales para pesos y bias en la red para esta función en específico, representado en la Ilustración 3. Al proceso de encontrar dichos valores ideales de pesos y bias se le conoce como entrenamiento de la red neuronal y será analizado en secciones posteriores.

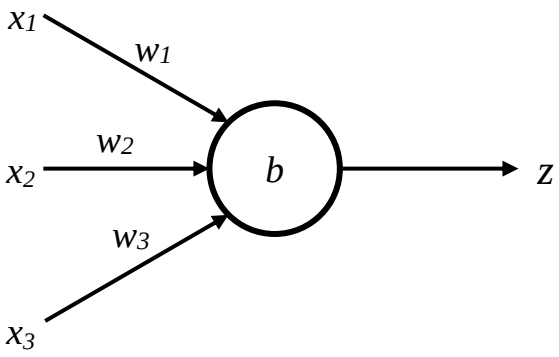


Ilustración 1. Forma base del perceptrón.

Ecuación 1. Evaluación del perceptrón es igual a la sumatoria la multiplicación de cada peso (w) por su valor de entrada (x) más el bias (b) en la neurona.

$$z = \begin{cases} 0 & si \sum_j w_j * x_j \leq b \\ 1 & si \sum_j w_j * x_j > b \end{cases}$$

$$z = \begin{cases} 0 & si \sum_j w_j * x_j + b \leq 0 \\ 1 & si \sum_j w_j * x_j + b > 0 \end{cases}$$

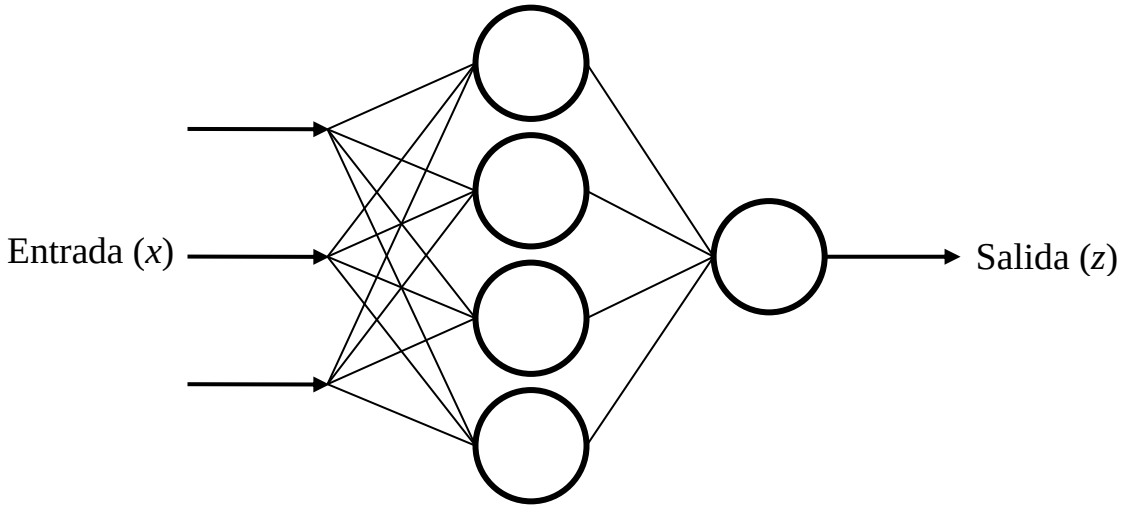


Ilustración 2. Forma perceptrón multicapa.

Ecuación 2. El cambio en la salida de MLP es aproximadamente igual a la sumatoria del cambio en cada uno de los pesos (w) multiplicado por la derivada parcial de la función de salida de la capa (z) con respecto al peso más la multiplicación del bias (b) con la derivada parcial de la función de salida de la capa con respecto al bias.

$$\Delta z \approx \sum_j \frac{\partial z}{\partial w_j} \Delta w_j + \frac{\partial z}{\partial b} \Delta b$$

Como se mencionó anteriormente, cualquier cambio en los pesos o bias en la red afectan directamente la salida. Sin embargo, en una arquitectura de perceptrones la salida de cada neurona siempre es 0 ó 1, de manera que es posible que cambios pequeños en los pesos o bias puedan cambiar el resultado de una neurona alterando drásticamente el resultado de la red neuronal. Estos cambios bruscos hacen que entrenar una red MLP sea un proceso lento y complicado. Para mitigar estos cambios drásticos en el resultado de la red se introdujeron las funciones de activación, dando origen a las redes neuronales clásicas.

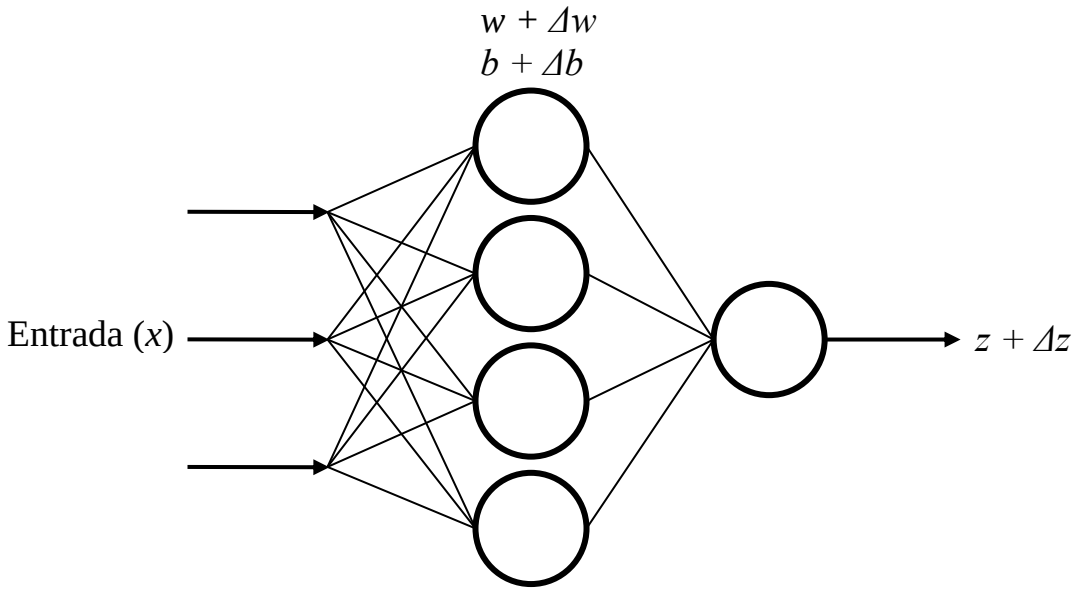


Ilustración 3. Un cambio en los pesos o bias en las neuronas generan un cambio en la salida del perceptrón multicapa.

3.1.2. Funciones de activación

Las funciones de activación son funciones que se aplican sobre el resultado de cada neurona en una red neuronal antes de propagar su salida hacia las siguientes neuronas. El propósito de las funciones de activación es el suavizar los cambios que ocurren en la salida de la red neuronal en función de variaciones en los valores de los pesos y los bias dentro de cada neurona. Existen varias funciones de activación, sin embargo, las más comunes son sigmoide, tangente hiperbólica y unidad lineal rectificada (ReLU, por sus siglas en inglés).

3.1.2.1. Sigmoide

La función sigmoide, también conocida como función logit (Ecuación 3). Esta función describe una curva en forma de S que está acotada a valores entre 0 y 1, como se muestra en la Ilustración 4.

3.1.2.2. Tangente hiperbólica

La función tangente hiperbólica está dada por la Ecuación 4. Esta función describe una curva en forma de S similar a la función sigmoide, pero que está acotada a valores entre -1 y 1, como se muestra en la Ilustración 5.

3.1.2.3. Rectificador (ReLU)

La función ReLU simplemente regresa el mismo valor de entrada si este es positivo, en caso contrario regresa cero, como se muestra en la Ilustración 6.

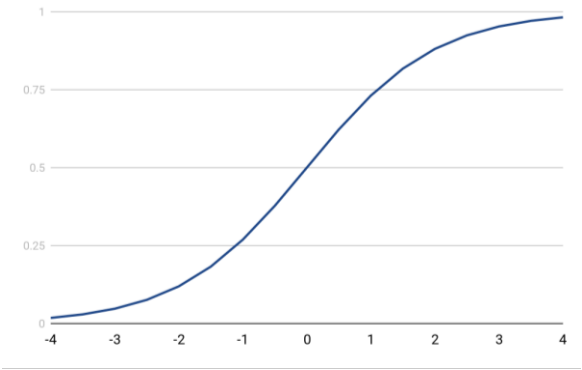


Ilustración 4. Gráfica de la función sigmoide.

Ecuación 3. Función sigmoide y su derivada.

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

$$\sigma'(z) = \sigma(z) * (1 - \sigma(z))$$

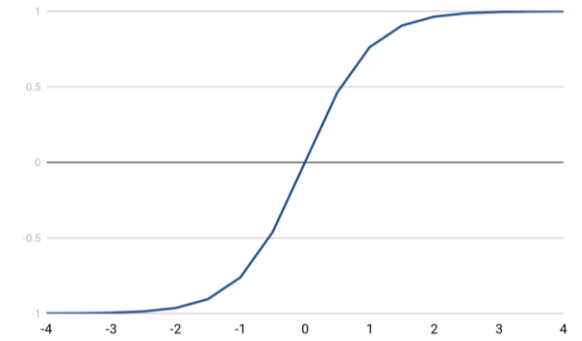


Ilustración 5. Gráfica de la función tangente hiperbólica.

Ecuación 4. Función tangente hiperbólica y su derivada.

$$tanh(z) = 2\sigma(2z) - 1 = \frac{2}{1+e^{-2z}} - 1$$

$$tanh'(z) = 1 - tanh^2(z)$$

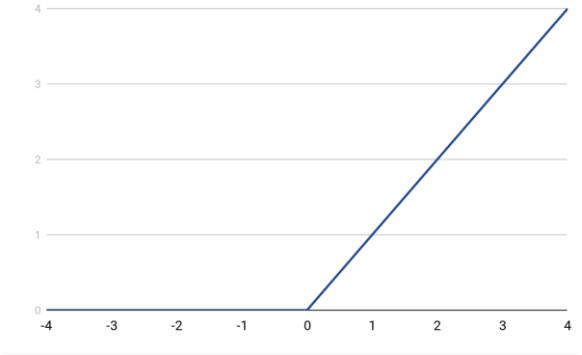


Ilustración 6. Gráfica de la función ReLU.

Ecuación 5. Función ReLU y su derivada.

$$relu(z) = \begin{cases} 0 & si\ z < 0 \\ z & si\ z \geq 0 \end{cases}$$

$$relu(z) = \begin{cases} 0 & si\ z < 0 \\ 1 & si\ z \geq 0 \end{cases}$$

3.1.3. Redes neuronales clásicas (*feedforward*)

A partir de las redes MLP surgen las redes neuronales *feedforward* también conocidas como redes neuronales clásicas. El modelo de estas redes consiste en un grafo dirigido en donde cada nodo representa una neurona que tiene aristas de entrada y aristas de salida. De forma similar a las redes MLP, cada arista en la red tiene peso y cada neurona tiene un valor de bias.

Las neuronas de una red neuronal clásica realizan la misma operación que los perceptrones para obtener su valor de salida, multiplicar los valores de las aristas de entrada (z) por sus respectivos pesos (w) y sumar el bias (b) de la neurona. La diferencia es que al resultado se le aplica una función de activación (a).

Ecuación 6. Evaluación feedforward (a) equivale a evaluar la función de activación (f) en el resultado (z) de la sumatora de la multiplicación de cada uno de los pesos (w) por su respectivo valor de entrada (x) más el bias en la neurona (b).

$$z = \sum_j w_j * x_j + b$$
$$a = f(z)$$

Las neuronas de una red neuronal clásica están agrupadas en capas. Cada arista de salida de las neuronas de una capa se conecta a cada una de las neuronas de la siguiente capa, convirtiéndose en sus respectivas aristas de entrada. A su vez, las capas de una red neuronal se clasifican en tres tipos, descritos en la Ilustración 7:

- *Capa de entrada*: La capa de entrada representa los datos de entrada de la red neuronal y es la primera capa en la red. En esta capa realmente no se tienen neuronas, sino solo datos de entrada para la red neuronal que servirán como entrada para las aristas de las capas ocultas.

- *Capas ocultas:* Cada red neuronal contiene una o más capas ocultas. Las neuronas en estas capas simplemente se limitan a procesar sus aristas de entrada y propagar los resultados hacia sus aristas de salida. Las neuronas en esta capa son las más importantes, ya que son las que representan los cálculos internos de la red neuronal que se irán ajustando durante el entrenamiento para que el resultado de la evaluación de la red sea lo esperado.

- *Capa de salida:* Las neuronas en esta capa representan el resultado de la evaluación de la red neuronal, también llamada predicción. Es la última capa de la red.

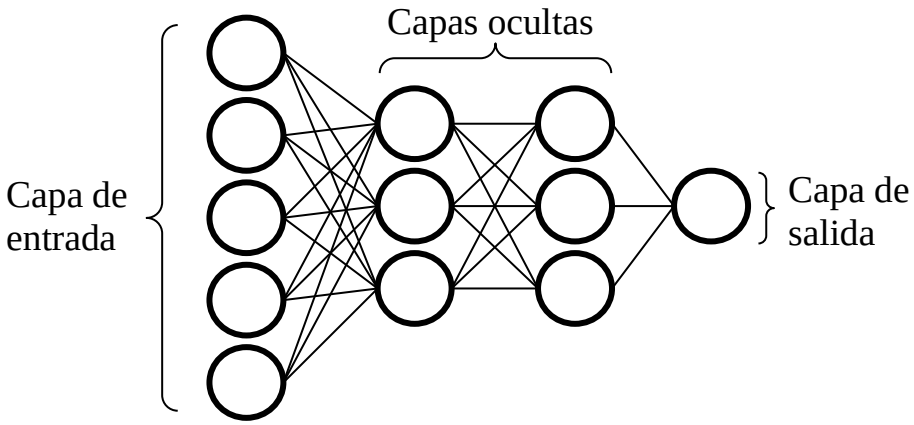


Ilustración 7. Estructura de una red neuronal clásica.

Por ejemplo, suponiendo que se desea crear una red neuronal capaz de determinar si una imagen contiene o no un perro, el proceso de evaluación ocurriría de la siguiente manera:

- La entrada de la red neuronal sería la imagen por analizar. Por lo tanto, cada píxel en la imagen representaría una neurona en la capa de entrada.

- La capa de salida sería una única neurona, la cual indicaría si la imagen tiene un perro (neurona activa) o no (neurona no activa).

- Las capas ocultas representan las operaciones que se llevarán a cabo dentro de la red neuronal para obtener la salida deseada. A diferencia de las capas de entrada y salida, no existe alguna regla que defina el número de neuronas dentro de cada capa oculta en la red. Para elegir el número de neuronas por capa, es necesario experimentar con diferentes configuraciones hasta encontrar una adecuada.

Para conseguir que una red neuronal aprenda a resolver un problema en específico, es necesario entrenarla a partir de muchos ejemplos de entradas catalogadas con su resultado esperado, las cuales serán evaluadas por la red neuronal quien a su vez actualizará sus pesos y bias de acuerdo con lo aprendido durante el entrenamiento. Este proceso se puede apreciar en las Ilustraciones 8 y 9.

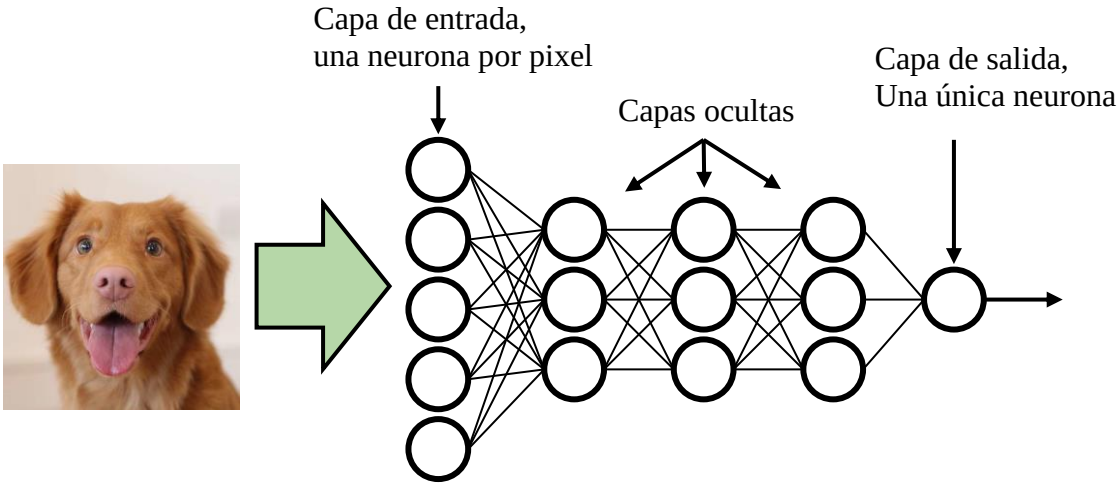


Ilustración 8. Ejemplo de modelo de red neuronal.

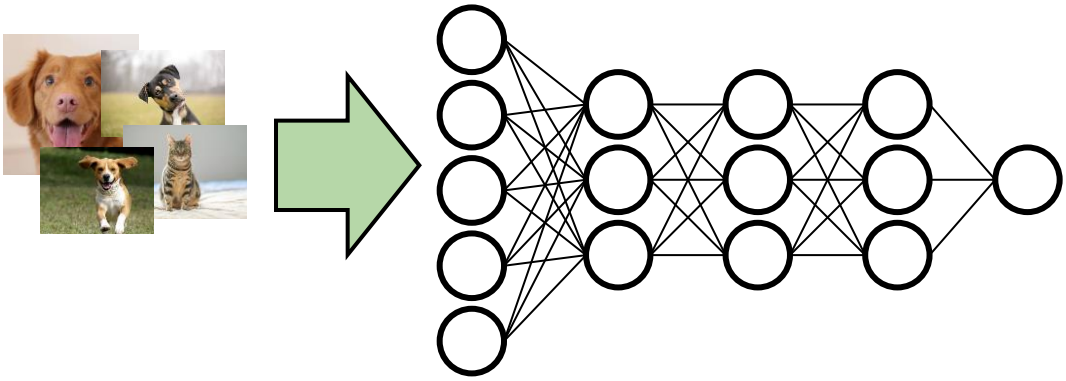


Ilustración 9. Ejemplo de entrenamiento de red neuronal.

3.1.3.1. Entrenamiento de una red neuronal clásica

Para calcular la salida de una red neuronal clásica, se toman los valores de la capa de entrada y se evalúa cada una de las neuronas de la primera capa oculta como se vio anteriormente: multiplicando las entradas por los pesos, sumando el bias de la neurona y aplicando la función de activación al resultado. El resultado de la primera capa oculta sirve entonces como entrada para la siguiente y el proceso se repite hasta llegar a la capa de salida. El resultado de la capa de salida representará el resultado de la evaluación de toda la red. A este proceso se le conoce como *feedforward*, descrito en la Ilustración 10.

Ecuación 7. La activación en la capa actual (l) es igual al resultado de aplicar la función de activación (f) al producto punto de los pesos (w) de cada neurona (k) de la capa actual (l) y la activación (a) de cada neurona (k) de la capa anterior (l-1) más el bias (b) en la capa actual.

$$a_j^l = \sum_j f\left(w_{jk}^l * a_k^{l-1} + b_j\right)$$

La red neuronal comienza con valores predefinidos (normalmente aleatorios) para los pesos y bias en cada una de sus neuronas, por lo que es muy probable que al realizar *feedforward* el resultado en la capa de salida no sea el esperado. Sin embargo, este resultado erróneo es de bastante utilidad para establecer qué tan lejos se encuentra el modelo del resultado deseado. En base a esta diferencia es posible determinar el incremento o decremento necesario en los pesos y bias de la red para lograr un resultado adecuado. Al proceso de propagar el error de la capa de salida hacia las capas anteriores para ajustar los pesos y bias se le conoce como *backpropagation* (descrito en la Ilustración 11) y es el algoritmo que permite que las redes neuronales aprendan.

Anteriormente se menciona que el propósito del entrenamiento de una red neuronal es el ajustar los pesos y bias para obtener los resultados deseados. Para establecer matemáticamente qué tan cerca se encuentra la salida de una red neuronal del resultado deseado se define una función de costo la cual se espera aproximar a cero. Existen muchas funciones de costo, un ejemplo bastante común es el error cuadrático medio (MSE por sus siglas en inglés).

Ecuación 8. El MSE (C) del resultado obtenido durante feedforward (a) equivale a la mitad de la sumatoria de la diferencia de los resultados esperados (y) y los resultados obtenidos (a) con cada dato de entrada (x) elevada al cuadrado. Así mismo, la derivada parcial del MSE con respecto al resultado obtenido (a) equivale a la diferencia entre dicho resultado obtenido (a) y el resultado esperado (y).

$$a = f(w \cdot x + b)$$

$$C(w,\, b) \;=\; \frac{1}{2} \sum_x (y - a)^2$$

$$\frac{\partial C}{\partial a} = a - y$$

Como se puede apreciar en la ecuación, MSE garantiza que el costo siempre sea un valor entre 0 y 1. Si el resultado deseado (y) y el resultado obtenido (a) son muy cercanos, el costo tiende a 0. En caso contrario el costo tiende a 1. Para aproximar la función de costo a cero, backpropagation utiliza una técnica conocida cómo gradiente descendiente, la cual consiste en utilizar derivadas parciales de la función de costo para determinar la dirección en la que se necesitan desplazar los valores de pesos y bias de la red para encontrar el mínimo más cercano.

El proceso de backpropagation se puede expresar en cuatro ecuaciones principalmente:

Ecuación 9. Error en la última capa.

$$\delta^L = \frac{\partial C}{\partial a^L} * f'\left(z^L\right)$$

El error en la última capa (δ^L) equivale al producto de la derivada parcial de la función de costo (∂C) con respecto a la activación en la última capa (∂a^L) y la evaluación de la derivada de la función de activación ($f'\left(z^L\right)$).

Ecuación 10. Error en las capas intermedias.

$$\delta^l = \left(\left(w^{l+1}\right)^T \cdot \delta^{l+1}\right) * f'\left(z^l\right)$$

El error en cualquier capa (δ^l) equivale al producto punto de los pesos transpuestos de la capa siguiente ($\left(w^{l+1}\right)^T$) y el error en la capa siguiente (δ^{l+1}) multiplicado por la evaluación de la derivada de la función de activación ($f'\left(z^l\right)$).

Ecuación 11. Cambio en los bias.

$$\frac{\partial C}{\partial b^l} = \delta^l \qquad b^l = \frac{\partial C}{\partial b^l} * \lambda$$

El cambio necesario en los bias la capa actual para aproximar la función de costo a cero equivale al error obtenido en la misma capa.

Ecuación 12. Cambio en los pesos.

$$\frac{\partial C}{\partial w^l} = a^{l-1} \cdot \delta^l \qquad w^l = w^l + \frac{\partial C}{\partial w^l} * \lambda$$

El cambio necesario en los pesos de la capa actual para aproximar la función de costo a cero equivale al producto punto de la activación en la capa anterior y el error obtenido en la capa actual.

La ejecución de *backpropagation* se divide en 3 pasos:

- ***Feedforward***: Durante este primer paso se ejecuta el algoritmo de *feedforward* como se vio anteriormente. La diferencia en este punto es que es necesario almacenar los valores de activación de cada neurona (a) así como el resultado obtenido antes de la activación, este último es conocido también como caché (z). Ambos datos serán utilizados en los siguientes pasos.

- **Cálculo de error en las capas**: El siguiente paso consiste en emplear las primeras dos ecuaciones descritas anteriormente para determinar el error obtenido en cada capa.

  La ecuación para el error en la última capa (Ecuación 9) necesita evaluar la activación de la última capa sobre la derivada parcial de la función de costo con respecto a la activación y el resultado será multiplicado por la evaluación de la derivada de la función de activación sobre la caché de la última capa. Por ejemplo, si la función de costo empleada es MSE, su derivada parcial equivaldría a restarle a la activación el resultado esperado.

Una vez obtenido el error en la última capa es posible determinar el error en las siguientes capas (Ecuación 10). Esta operación se va realizando en cada una de las capas desde la penúltima y hasta la primera, almacenando los valores de error (δ^l) ya que serán utilizados en el siguiente paso.

- **Actualización de pesos y bias**: El último paso es el calcular el cambio necesario en los pesos y bias en función del error obtenido en cada capa. Para esto se emplean las ecuaciones para el cambio en el bias (Ecuación 11) y el cambio en los pesos (Ecuación 12).

  Una vez obtenidos los cambios esperados en el bias y pesos, se procede a ajustar dichos valores sumando los cambios obtenidos a los valores actuales de bias y pesos. En el contexto de las redes neuronales, este cambio suele ser muy abrupto por lo que los valores de cambio son multiplicados por una constante conocida como tasa de aprendizaje (λ), la cual es encargada de reducir el valor del cambio, suavizando así las variaciones en los pesos y bias. La tasa de aprendizaje es un atributo configurable en el modelo de la red neuronal.

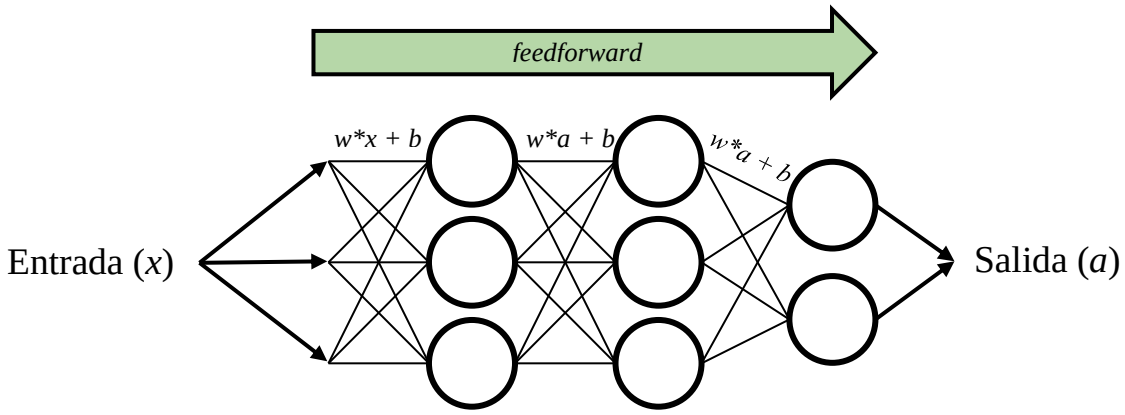


Ilustración 10. Ejemplo ejecución feedforward.

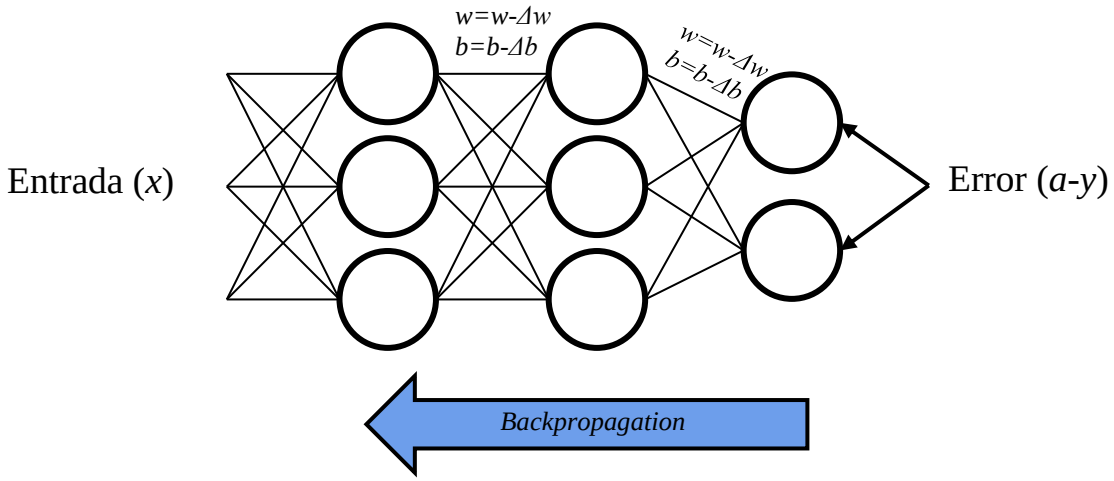


Ilustración 11. Ejemplo ejecución backpropagation.

Este proceso se realiza con cada una de las entradas de entrenamiento y se repite un número de veces previamente establecido esperando obtener un valor muy cercano a cero como resultado de la función de costo. Al número de veces que se repite este proceso se le conoce como número de épocas, y es otro atributo configurable en el modelo de la red neuronal.

Como se puede observar en las ecuaciones empleadas, el cálculo del gradiente descendiente, así como su propagación hacia atrás mediante *backpropagation* resulta en un procesamiento bastante exhaustivo para el procesador si se intenta realizar secuencialmente. Es por este motivo que se desarrollaron algunas variantes de este algoritmo para sacar provecho tanto de instrucciones SIMD como de dispositivos para el cómputo paralelo de lotes tales como GPUs.

3.1.3.1.1. Gradiente Descendiente - Procesamiento por lotes (*batch*)

El cálculo del gradiente descendiente por lotes consiste en considerar todos los datos de entrada como una única matriz de entrada. Las ecuaciones detalladas en backpropagation se pueden expresar como operaciones sobre matrices, de manera que en lugar de serializar las operaciones sobre cada elemento de los datos de entrenamiento es posible realizar todas las operaciones sobre la matriz de entrada.

La ventaja de este procesamiento por lotes es que, si se cuenta con suficiente memoria RAM para almacenar todos los datos de prueba y hardware capaz de realizar operaciones tales como producto punto de matrices en paralelo de forma eficiente, será posible entrenar una red neuronal mucho más rápido que de forma serializada.

Sin embargo, por lo general este procesamiento por lotes no es lo deseado, ya que difícilmente se tendrá suficiente memoria RAM para resolver los problemas tradicionales de redes neuronales. Otro problema de este procesamiento por lotes es que, si en algún punto ocurre algún problema en la máquina, el cómputo realizado hasta el momento se perderá.

3.1.3.1.2. Gradiente Descendiente - Procesamiento de mini-lotes (*mini-batch*)

Para solventar las desventajas del procesamiento por lotes, surge una idea bastante similar pero que, en lugar de operar sobre la totalidad de los datos, divide los datos de entrada en grupos más pequeños conocidos como mini-batch. El número de mini-batches es configurable.

Esta implementación tiene varias ventajas:

- No se requiere tanta memoria RAM ya que solo se cargará un mini-batch a la vez.
- Cada mini-batch actualizará los pesos y bias en la red, por lo que ocurren actualizaciones más rápidas y será posible detectar más rápido cuando la red llegue al punto deseado (el costo es muy cercano a cero).
- Debido a que cada mini-batch actualiza los valores en la red, es posible guardar un respaldo del estado de la red después de cada mini-batch. Si algo le ocurre a la máquina, será posible restaurar el entrenamiento a partir del último mini-batch.

3.1.3.1.3. Gradiente Descendiente Estocástico

El gradiente descendiente estocástico es una variante del gradiente descendiente por mini-batch, en el cual durante cada época se realiza un desordenamiento aleatorio de los datos de cada mini-batch para de esta forma generar un poco de variación en el entrenamiento de la red.

3.1.4. Generalización y sobreentrenamiento

La premisa del entrenamiento de una red neuronal es que una vez que la red haya sido entrenada con suficientes datos de entrada, esta será capaz de generalizar el problema y resolver correctamente entradas que no formaron parte del entrenamiento de la red. Sin embargo, esto no siempre es así. Si la red neuronal es entrenada durante más épocas de las necesarias con los mismos datos de entrada, es posible que al finalizar el entrenamiento la red solo sea capaz de resolver problemas dentro de los datos de entrada y no llegue a generalizar el problema. A esta situación se le conoce como sobreentrenamiento.

Para evitar el sobreentrenamiento, es necesario dividir los datos de entrada en al menos dos conjuntos: datos de entrenamiento y datos de validación. Los datos de entrenamiento son aquellos empleados para evaluar la red neuronal y realizar *backpropagation* para ir ajustando los pesos y bias en la red, mientras que los datos de validación no formarán parte de la ejecución de *backpropagation* pero servirán para determinar qué tanto ha mejorado la predicción de la red neuronal sobre datos ajenos al entrenamiento.

Si durante cada época en el entrenamiento de una red neuronal se monitorea el resultado de la función de costo tanto con los datos de entrenamiento como con los de validación, se puede detectar uno de tres escenarios:

- **El costo es mayor en el conjunto de datos de entrenamiento que en el de validación.** Esto quiere decir que el conjunto de los datos de validación es muy pequeño en comparación con el de entrenamiento o que los datos de validación contienen elementos que son mucho más sencillos de identificar. Es necesario realizar una mejor distribución de los datos.

- **El costo es igual en el conjunto de datos de entrenamiento que en el de validación.** Este es el escenario ideal, siempre y cuando la tendencia del costo indique que va disminuyendo y por lo tanto el modelo va mejorando. En este punto, la distribución entre datos de entrenamiento y datos de validación es excelente.

- **El costo es menor en el conjunto de datos de entrenamiento que en el de validación.** Esto quiere decir que el modelo ha sido sobreentrenado y por lo tanto le resulta bastante difícil generalizar y encontrar soluciones a entradas desconocidas. Esto se puede deber a que el número de épocas es muy elevado o a que se cuenta con una tasa de aprendizaje muy amplia.

El graficar los cambios en el costo de una red neuronal en función de los datos de entrenamiento y validación es un punto crucial durante las fases de investigación de modelos de redes neuronales debido a que es en este punto en el que se va experimentando con distintos valores para la taza de aprendizaje, el número de épocas, número de capas ocultas, número de neuronas por capa, distintas funciones de activación, etc. A todas estas variables se les conoce como hiperparámetros. Un ejemplo de este tipo de gráficas y su interpretación se puede observar en la Ilustración 12.

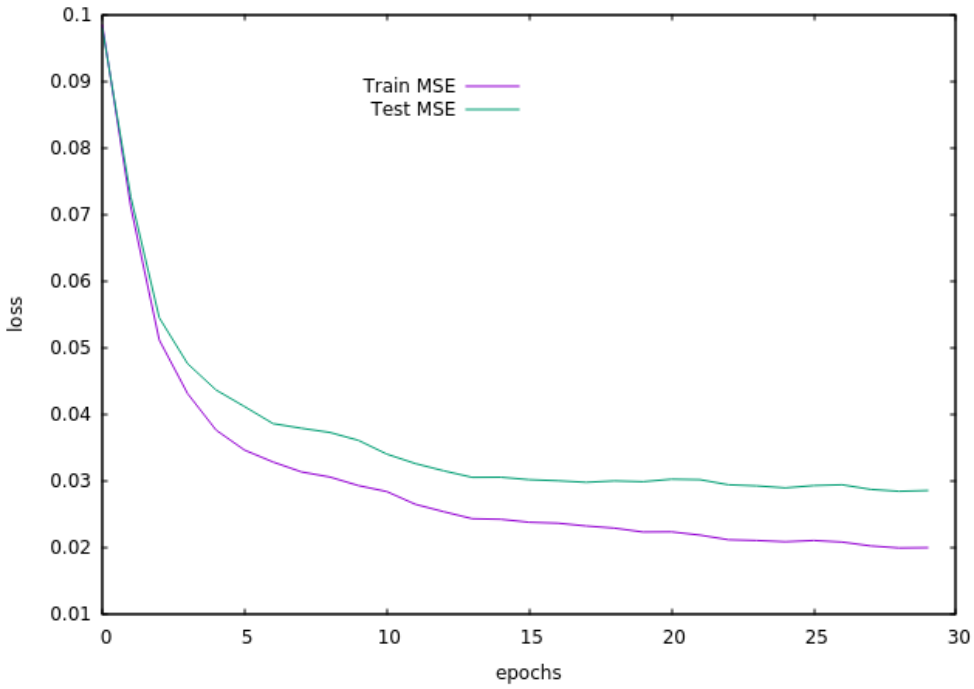


Ilustración 12. Grafica del cambio del costo en función del tiempo. En este caso el modelo ha sido sobreentrenado, por lo que el costo es mayor en los elementos de validación que en los elementos de entrenamiento.

3.2. OpenCL (Open Computing Language)

Dependiendo del contexto, OpenCL puede ser considerado un estándar de programación en paralelo, una biblioteca para procesar datos en el GPU o un lenguaje de programación de rutinas para GPU.

OpenCL es el nombre de un estándar de programación en paralelo publicado en el 2008 por grupo Khronos[14], responsables también de otros estándares de programación tales como OpenGL y Vulkan. El estándar de OpenCL establece reglas de comunicación entre la máquina huésped o *host* y dispositivos CPU y GPU, las cuales son implementadas por los distintos fabricantes de procesadores como Intel, AMD y Nvidia. La finalidad de OpenCL es otorgar al desarrollador una plataforma única para escribir programas capaces de procesar grandes cantidades de datos de forma paralela sacando partido del *hardware* disponible en el sistema, independientemente de su fabricante.

Todo desarrollo de OpenCL se divide en dos partes: la aplicación en la máquina huésped o *host* y el programa a ejecutar en el dispositivo o *device*. La aplicación del *host* es responsable de la lógica de negocio detrás de un desarrollo de OpenCL: inspeccionar el hardware disponible en la máquina, transferir datos hacia los dispositivos y encolar las rutinas a realizar en los mismos. La aplicación realiza todas estas operaciones mediante la biblioteca OpenCL (libopencl). Esta biblioteca tiene soporte oficial para C y C++, pero existen otros proyectos no oficiales que proporcionan interfaces para interactuar con libopencl mediante otros lenguajes como Python y Rust.

El programa en el *device* consiste en la definición de las rutinas o *kernels* a realizar en los dispositivos que implementen el estándar OpenCL. Se conoce también como OpenCL al lenguaje utilizado para escribir

dichos *kernels*. La aplicación *host* está a cargo entonces de cargar los distintos *kernels* de uno o varios programas y mandarlos a ejecutar en los dispositivos. Este programa puede ser compilado en tiempo de ejecución dentro de la aplicación *host*, por lo que es necesario tener instalado un compilador de OpenCL. Este lenguaje de OpenCL está basado en C, sin embargo, cuenta con una serie de ventajas con respecto a C[15]:

- **Portabilidad**: OpenCL permite desacoplar el código de la aplicación del dispositivo en el que se realizará el cómputo, por lo que es posible compilar y ejecutar el mismo programa en múltiples dispositivos de distintos fabricantes, incluso simultáneamente.

- **Operaciones de vectores estandarizadas:** Si bien la mayoría de los dispositivos en el mercado soportan operaciones vectoriales a nivel instrucción de procesador, cada fabricante implementa dichas instrucciones de forma distinta. Procesadores de distintos fabricantes pueden utilizar un número distinto de bytes para representar los mismos tipos de datos numéricos, tener distintas representaciones de datos como *little endian* y *big endian*, o incluso puede ser necesario utilizar más de una instrucción en algunos procesadores para lograr el mismo resultado obtenido con una sola instrucción en otro procesador. OpenCL estandariza todas las operaciones de vectores como operaciones nativas del lenguaje e internamente se encarga de resolver todos estos problemas de compatibilidad, por lo que el desarrollador no necesita preocuparse más por estas diferencias en el *hardware*.

- **Paralelismo de tareas:** Con OpenCL es posible definir tareas a realizar en cada uno de los dispositivos disponibles en el sistema (así pertenezcan a diferentes fabricantes) y ejecutarlas de forma paralela mediante una misma API.

3.2.1. Arquitectura de un sistema OpenCL

Para comprender cómo es que funciona OpenCL internamente, el estándar define una representación de alto nivel de la arquitectura de cualquier sistema desarrollado en OpenCL. Dicha arquitectura se puede observar en la Ilustración 13.

Cada sistema OpenCL cuenta con un único *host* (la máquina que ejecuta el programa del host), el cual puede contar con uno o varios dispositivos OpenCL conectados: CPU, GPU, *Field-Programmable Gate Arrays* (FPGA), entre otros. Cada dispositivo internamente contiene una o varias unidades de cómputo, que a su vez se encuentran divididas en varios elementos procesadores. Estos elementos procesadores representan de forma física el número máximo de operaciones que se pueden realizar simultáneamente dentro de un dispositivo.

En las siguientes secciones se explica a mayor detalle cómo es que todas estas partes entran en juego durante la ejecución de OpenCL.

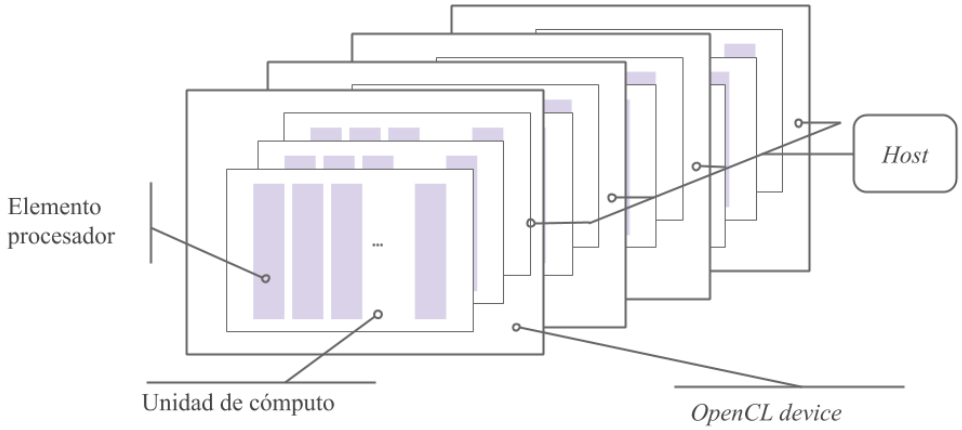


Ilustración 13. Representación de alto nivel de la arquitectura OpenCL.[22]

3.2.1.1. Plataformas y dispositivos

Como se mencionó anteriormente, cada fabricante de hardware provee la implementación de OpenCL compatible con su dispositivo. Estas implementaciones son distribuidas mediante paquetes conocidos como *Software Development Kit* (SDK) y son necesarias para poder operar en el dispositivo. OpenCL es agnóstico al tipo de *hardware* que existe en el sistema, por lo que es posible distribuir el cómputo entre dispositivos de diferentes fabricantes en una misma máquina, cada uno con su respectivo SDK instalado. Dentro del ámbito de OpenCL, a estas SDK se les conoce también como plataformas.

En tiempo de ejecución, OpenCL permite a la aplicación obtener la lista de plataformas disponibles e inspeccionar sus características. Cada plataforma cuenta con uno o más dispositivos asociados a la misma, los cuales también son inspeccionados en tiempo de ejecución. La aplicación puede seleccionar qué dispositivos serán empleados para el cómputo mediante la creación de un contexto. En la Ilustración 14 se puede observar de forma general el flujo de operación de OpenCL y sus diferentes elementos. Dichos elementos serán descritos en las siguientes secciones.

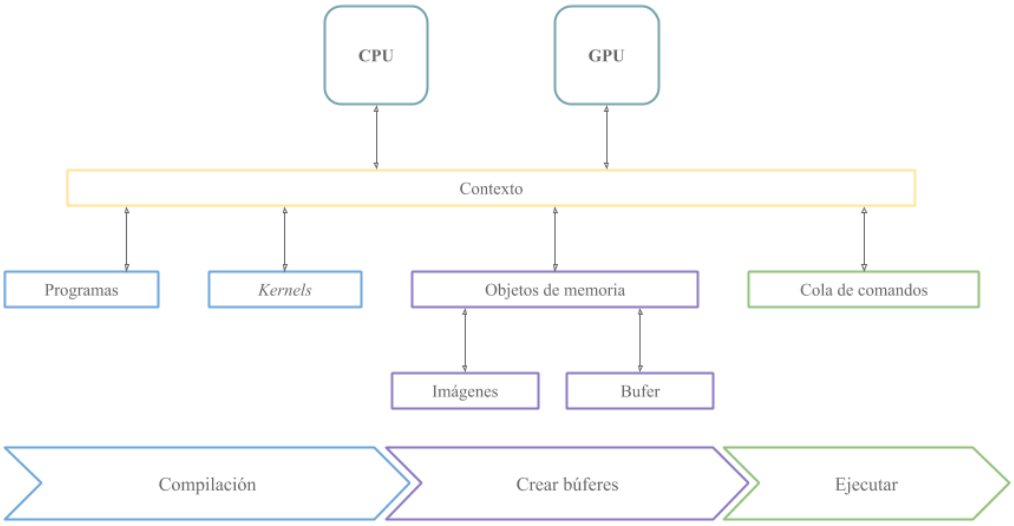


Ilustración 14. Vista general del flujo de OpenCL. [22]

3.2.1.2. Contexto

Un contexto es una estructura de OpenCL que representa a un subconjunto de dispositivos a utilizar y que sirve de referencia para la creación de otros objetos de OpenCL tales como colas de comandos, programas/*kernels* y búferes de memoria. Es posible tener más de un contexto simultáneamente, pero todos los dispositivos dentro de un contexto deben pertenecer a la misma plataforma.

3.2.1.3. Cola de comandos

La comunicación entre el *host* y los dispositivos OpenCL se lleva a cabo mediante el paso de mensajes por una o varias colas de comandos. Estas estructuras se crean a partir de un contexto de OpenCL y son utilizadas del lado del *host* para mandar a los dispositivos a procesar tres tipos de instrucciones (véase Ilustración 15):

- Ejecución de *kernels*.
- Comandos de sincronización.
- Lectura, escritura y mapeo de búferes.

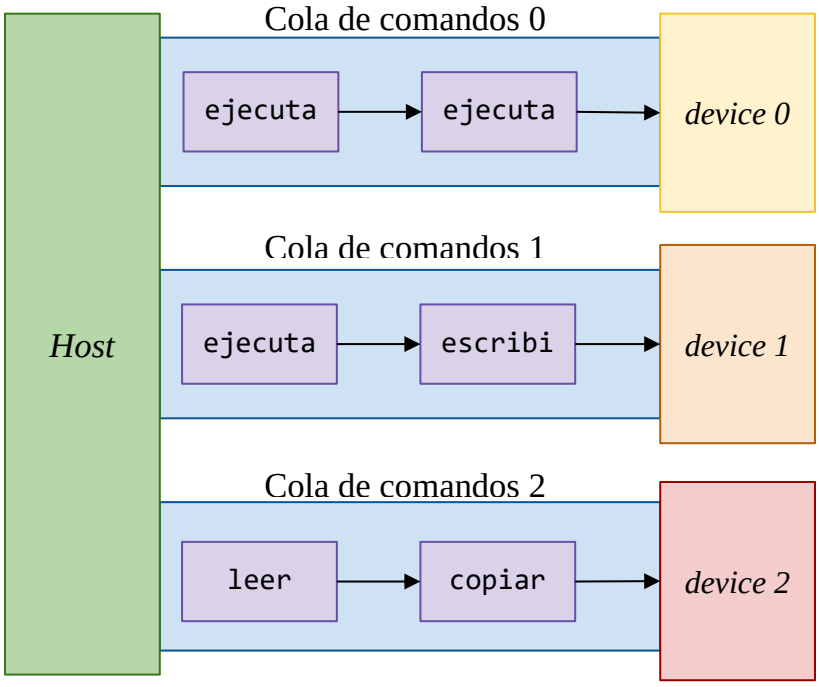


Ilustración 15. Procesamiento de la cola de comandos. [22]

3.2.1.4. Programas y kernels

Todo desarrollo en OpenCL se divide en dos partes: el programa del *host* y el programa del dispositivo. El programa del *host* funciona como cualquier otra aplicación escrita en C o C++. El código se compila, se obtiene un binario y al ejecutar la aplicación se comienza a correr el programa a partir de la función `main`. A pesar de tener muchas similitudes, el proceso para el programa del dispositivo funciona de manera diferente.

En primer lugar, el código del programa del dispositivo se compila en el tiempo de ejecución del programa del *host* y su resultado se mantiene en memoria. El programa dispositivo no tiene función `main`, sino que contiene la definición de uno o varios *kernels*. Un *kernel* es una función declarada con el decorador `__kernel`, la cual sirve como punto de entrada a la ejecución en un dispositivo OpenCL, de manera similar al `main` en el programa del *host*. La diferencia principal es que mientras en el programa del *host* existe un solo hilo de ejecución procesando la función *main*, en el programa del dispositivo se tienen muchos hilos de ejecución, cada uno procesando de forma paralela su propia instancia del *kernel* a ejecutar.

Para comenzar el procesamiento de un *kernel*, es necesario indicar a los dispositivos destino el nombre del kernel, el número de instancias a ejecutar (*work-items*), el número de grupos en el que serán repartidas dichas instancias (*work-groups*) y los argumentos que recibe el kernel, para posteriormente encolar la operación en la cola de comandos que se construye a partir del contexto.

3.2.1.4.1. Work-items

Cuando se manda a ejecutar un kernel a través de la cola de comandos, es necesario especificar el número de instancias del *kernel* que se desean ejecutar. Estas instancias reciben también el nombre *work-item*. La cantidad de *work-items* a ejecutar está representada por una tupla que puede tener una, dos o tres dimensiones y el número de *work-items* por dimensión está definido por cada fabricante de hardware (Tabla 2).

Cada uno de los *work-items* es ejecutado por un elemento procesador, la parte más granular dentro de la arquitectura de alto nivel de un sistema OpenCL.

3.2.1.4.2. Work-groups

Es posible agrupar varios *work-items* para formar *work-groups*. De igual manera que los *work-items*, los *work-groups* también se definen al momento de mandar a ejecutar un kernel a través de la cola de comandos. La definición de los *work-groups* también está representada por una tupla de una, dos o tres dimensiones, con la característica de que el número de *work-groups* en cada dimensión debe ser un factor del número de *work-items* definido en la misma dimensión (Tabla 3).

Cada *work-group* es procesado por una unidad de cómputo dentro de los dispositivos, por lo que todos los *work-items* dentro del mismo grupo pueden acceder a la memoria local dentro de cada unidad de cómputo, la cual es mucho más rápida que la memoria global.

3.2.1.4.3. Argumentos del *kernel*

Los *kernels* pueden recibir como argumentos datos primitivos o punteros a búferes de memoria en el dispositivo.

Los datos primitivos funcionan de forma similar a las funciones en C/C++, sin embargo, en OpenCL se cuenta con el tipo de dato vector. Este tipo fue diseñado con el propósito de sacar provecho de las operaciones SIMD de los procesadores modernos y realizar cálculos de forma paralela en varios elementos a la vez.

Los búferes de memoria son bloques de memoria de los dispositivos OpenCL definidos desde la aplicación del *host*. Estos búferes son de gran utilidad para transferir grandes cantidades de datos entre el host y los dispositivos OpenCL, como se analizará en las siguientes secciones.

Tabla 2. Ejemplo de distribución de work-items.

Tupla work-items	**Descripción**
(2)	2 *work-items* en la dimensión 0 2 *work-items* en total
(2, 6)	2 *work-items* en la dimensión 0 6 *work-items* en la dimensión 1 12 *work-items* en total
(2, 6, 3)	2 *work-items* en la dimensión 0 6 *work-items* en la dimensión 1 3 *work-items* en la dimensión 2 36 *work-items* en total

Tabla 3. Ejemplo de distribución de work-groups.

Tupla work-items	**Tupla work-groups**	**Descripción**
(2)	(1)	1 *work-group* en la dimensión 0 1 *work-group* en total 2 *work-items* por grupo
(2, 6)	(1, 2)	1 *work-group* en la dimensión 0 2 *work-groups* en la dimensión 1 2 *work-groups* en total 6 *work-items* por grupo
(2, 6, 3)	(1, 2, 3)	1 *work-group* en la dimensión 0 2 *work-groups* en la dimensión 1 3 *work-groups* en la dimensión 2 6 *work-groups* en total 6 *work-items* por grupo

3.2.1.5. Búferes de memoria

Es posible reservar bloques de memoria en los dispositivos OpenCL desde la aplicación del *host*. Estos bloques de memoria son conocidos como búferes y sus principales usos son el transferir datos entre el *host* y el dispositivo, así como ser utilizados como argumentos de kernels.

Existen dos tipos de búferes en OpenCL, búfer de datos y búfer de imágenes. Un búfer de datos puede almacenar cualquier colección de tipo nativo de OpenCL tales como enteros, flotantes o vectores, mientras que los búferes de imágenes son objetos de memoria optimizados para operar con imágenes.

Los búferes pueden pertenecer a una de cuatro regiones de memoria definidas por el estándar de OpenCL. Dichas regiones de memoria se encuentran representadas en las Ilustraciones 16 (representación física) y 17 (representación lógica). Las regiones de memoria definidas en OpenCL son las siguientes:

- *Global*: Esta región memoria está disponible para lectura y escritura desde cualquier *work-item* en cualquier *work-group*. La memoria global se utiliza para copiar datos desde la memoria del host hacia el dispositivo y tenerla disponible para cualquier ejecución de kernel.

- *Constant*: La región de memoria constante es una subregión de la memoria global. La diferencia es que una vez que el host inicializa la memoria constante, esta no cambiará durante la ejecución del *kernel*.

- *Local*: La región local es una memoria mucho más rápida que la memoria global, pero más pequeña. Esta memoria solo se encuentra compartida entre los *work-items* que pertenezcan al mismo *work-group*. No es posible acceder a la memoria local de un *work-group* desde otro *work-group*. Tampoco se puede acceder desde el host a esta región de memoria. El propósito de esta región es almacenar búferes de datos en común para todos los *work-items* y de esta forma realizar lecturas de memoria mucho más rápidas dentro del contexto de un *work-group*.

- *Private*: La región privada de memoria es mucho más rápida que la memoria local, pero mucho más pequeña. Esta memoria sólo está disponible dentro de la ejecución de cada *work-item* y es la encargada de almacenar las variables que se definan dentro de la ejecución del mismo. El *host* tampoco tiene acceso a esta región de memoria.

Para transferir los contenidos de un búfer entre el *host* y algún dispositivo, es necesario establecer comunicación con el mismo mediante el uso de la cola de comandos, indicando el búfer en cuestión, así como la operación a realizar. Algunas de las operaciones que se pueden realizar sobre los búferes son las siguientes:

- *Read*: Lee un búfer de datos desde el dispositivo hacia el host.
- *Write*: Escribe un búfer de datos desde el host hacia el dispositivo.
- *ReadImage*: Similar a Read, pero optimizada para búferes de imágenes.
- *WriteImage*: Similar a Write, pero optimizada para búferes de imágenes.
- *CopyBuffer*: Copia el contenido de un búfer de datos de origen hacia uno de destino.
- *CopyImage*: Similar a CopyBuffer, pero optimizada para búferes de imágenes.

- *MapBuffer*: Mapea una región de la memoria del dispositivo a memoria en el host.
- *MapImage*: Similar a MapBuffer, pero optimizada para búferes de imágenes.
- *Unmap*: Revierte el proceso de MapBuffer o MapImage.

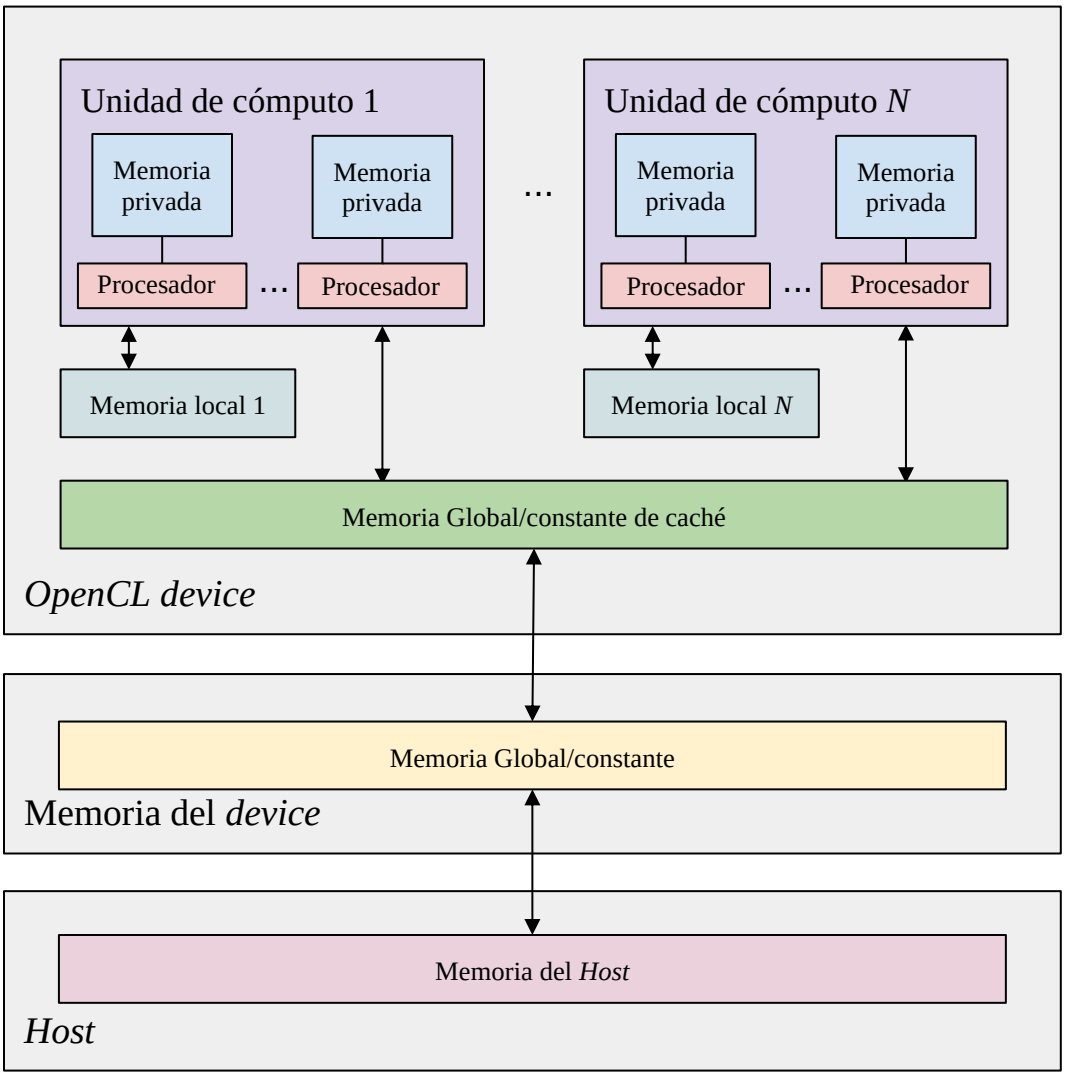


Ilustración 16. Representación física de los tipos de memoria en OpenCL. [22]

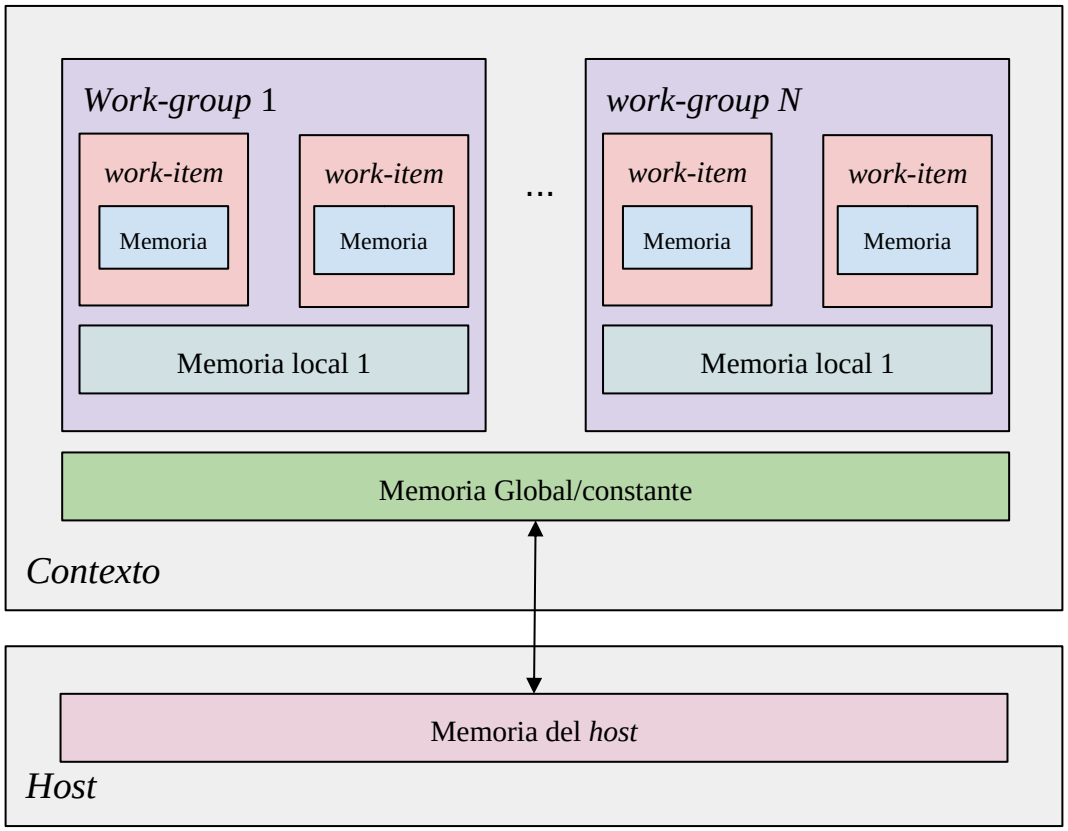


Ilustración 17. Representación lógica de la memoria de OpenCL. [22]

3.3. Rust

Rust es un lenguaje de programación de sistemas desarrollado por Mozilla con un desempeño equiparable al de lenguajes de bajo nivel como lo son C/C++[16]. En el 2012, Mozilla inició el desarrollo de su actual motor de navegador (Servo) escrito completamente en Rust, con el objetivo de mejorar la velocidad y rendimiento de Firefox sacando provecho del hardware *multi core* presente en los dispositivos actuales. Rust 1.0 se liberó oficialmente en el 2015 y actualmente se encuentra en la versión Rust 1.46.0.

Rust fue diseñado con la idea brindar un lenguaje capaz de mantener una administración de memoria eficiente, detectar en tiempo de compilación accesos incorrectos a memoria, *deadlocks*, condiciones de carrera y otros problemas comunes de concurrencia, al mismo tiempo que se mantiene una relación bastante cercana con el hardware para otorgar al desarrollador control sobre la administración de los recursos a bajo nivel de forma segura. El resultado es un lenguaje con la velocidad de C y que a su vez provee una semántica de alto nivel muy expresiva, lo que ha llamado la atención de la comunidad de desarrollo de software.

A continuación, se describen algunas de las características que hacen de Rust un lenguaje único y con bastante potencial para el desarrollo de sistemas:

- **Mutabilidad.** Por defecto todas las entidades en Rust son inmutables, el desarrollador tiene que declarar explícitamente cuando desea que una variable sea mutable. Esto ayuda a detectar errores

de mutabilidad en tiempo de compilación, así como a representar claramente las intenciones del uso de cada variable en el código.

- **Administración eficiente de memoria.** Rust elimina la gran mayoría de problemas de accesos inválidos de memoria tales como fallos de segmento, doble liberación de memoria o acceso a memoria restringida al ser detectados en tiempo de compilación. Esto es posible gracias a la introducción del innovador sistema de pertenencia (*ownership)*, que representa el núcleo de Rust.

  En Rust, cada entidad tiene un único dueño (*owner*), que puede ser otro contexto, una función, una estructura, etc. Es posible transferir el *ownership* de una entidad entre *owners*, siempre y cuando solo exista un único *owner* (*move*). El *owner* puede compartir la referencia de la entidad (*borrows*) con otras entidades, dicha referencia puede ser mutable o inmutable. El *owner* puede compartir tantas referencias inmutables como se desee, pero solo puede existir una referencia mutable a la vez.

  Gracias al sistema de *ownership*, cada entidad en Rust tiene claramente definido su tiempo de vida (*lifetime*) y es esto lo que permite a Rust detectar errores de memoria o concurrencia en tiempo de compilación.

- **No garbage collector.** Debido a que en Rust cada entidad tiene su *lifetime* definido en tiempo de compilación, el compilador es capaz de determinar en qué momento solicitar y liberar memoria sin la necesidad de un *garbage collector* (GC). Esto optimiza el manejo de memoria y hace que la liberación de recursos ocurra de forma rápida y segura.
- **Concurrencia ágil y segura**. En tiempo de compilación, Rust analiza también el *lifetime* de los recursos compartidos entre hilos (memoria, *sockets*, etc.), siendo posible detectar problemas comunes de concurrencia como lo son los *deadlocks*, condiciones de carrera, regiones críticas inseguras (*mutex*), entre otros. Normalmente los lenguajes de programación delegan al desarrollador la responsabilidad de evitar accesos inseguros a recursos compartidos, Rust garantiza que dichos accesos inseguros no existan.
- **Abstracciones *zero-cost*:** Rust cuenta con abstracciones que otorgan funcionalidad de alto nivel que al compilarse resulta en código de bajo nivel tan eficiente como si fuera escrito a mano[17]. Algunos ejemplos de abstracciones son las operaciones nativas con buffers, iteradores, mapas entre otras. Rust garantiza que estas abstracciones no agreguen *overhead* una vez compilado el código, y mencionan que no es posible generar código más eficiente que utilizar las abstracciones diréctamente.

En general, las ventajas que ofrece Rust sobrepasan por mucho los retos que supone el invertir en un nuevo lenguaje de programación, por lo que no es sorpresa que cada vez más proyectos se están migrando a este lenguaje[3], [18]. Desde su origen, Rust propuso reconsiderar la manera en que se ha empleado durante muchos años la gestión de recursos y la programación de sistemas en su totalidad. A pesar de ser un lenguaje bastante nuevo, está dando resultados en un nicho donde compite con lenguajes altamente eficientes como lo son C y C++ y que cuentan con muchos años de experiencia por detrás.

4. DESARROLLO METODOLÓGICO

En esta sección se detallan los distintos elementos que conforman el desarrollo de este trabajo, así como los puntos de referencia empleados para determinar el éxito de la librería desarrollada.

4.1. Levantamiento de requerimientos

Para determinar la eficiencia de la solución implementada se toman como referencia dos problemas bastante populares en el ámbito de machine learning cuya solución radica en el uso de redes neuronales: modelado de la función XOR y detección de dígitos escritos a mano mediante el conjunto de datos MNIST.

Ambos problemas se modelan tanto en la biblioteca desarrollada (CPU y OpenCL) como en Keras con Tensorflow, esperando obtener los mismos niveles de precisión. En ambos modelos se emplea MSE como función de costo y el algoritmo gradiente descendiente estocástico por mini-batches para ajustar los parámetros de la red.

4.1.1. Función XOR

La función XOR, también conocida como OR exclusiva, es una operación a nivel de bits la cual recibe dos entradas que pueden tener uno de dos posibles valores: 0 o 1. La salida de la función XOR será 0 si ambas entradas son iguales, o 1 si los valores de entrada son distintos, como se puede observar en la Ilustración 18.

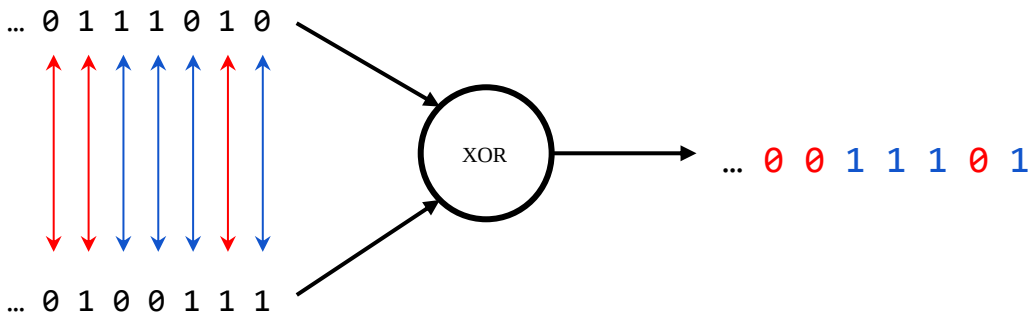
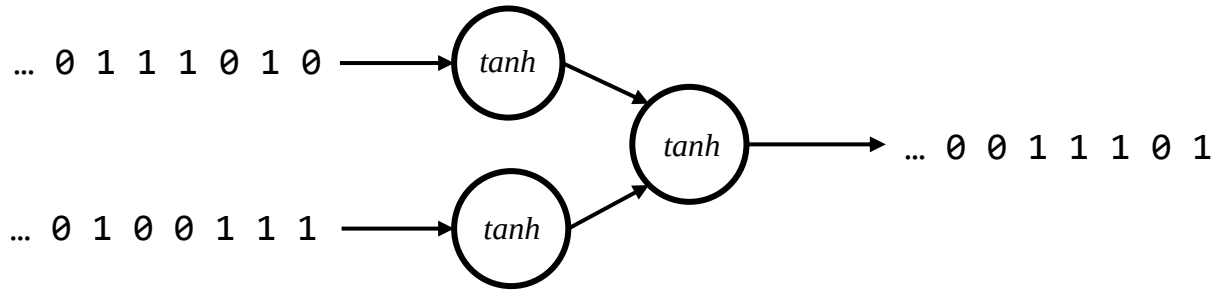


Ilustración 18. Ejemplo de función XOR.

Tabla 4. Tabla de verdad de compuerta XOR.

a	b	output
0	0	0
0	1	1
1	0	1
1	1	0

El modelado de una red neuronal para resolver este problema es bastante sencillo. La capa de entrada de la red tiene dos neuronas, que representan los dos bits de entrada de la función. De manera similar, la capa de salida tiene una única neurona, representando el resultado de la función XOR. La capa intermedia cuenta con dos neuronas para introducir complejidad a la red y mejorar así su convergencia. La activación en cada una de las neuronas es tangente hiperbólica. El modelo de red neuronal empleado para resolver este problema se puede observar en la Ilustración 19.




```
Neuro::new()
      .add_layer(Dense::new(2,Activation::Tanh))
      .add_layer(Dense::new(1,Activation::Tanh))
      .train(&x, &y, &x_test, &y, learning_rate, epochs, batch_size);
```

Ilustración 19. Grafo de red neuronal para el problema XOR y código del modelo.

El conjunto de datos de entrenamiento consistirá en las cuatro posibles entradas que tiene la función XOR (0,0), (0,1), (1,0) y (1,1). Para este problema no se cuenta con datos de validación debido a que los datos de entrenamiento representan todo el universo de entradas para la red neuronal.

4.1.2. Conjunto de datos MNIST

MNIST es una base de datos de dígitos escritos a mano que cuenta con la información de 60,000 imágenes de entrenamiento y 10,000 imágenes para pruebas. Este conjunto de datos es bastante utilizado en el

ámbito de *machine learning* ya que es un problema fácil de entender y que se puede resolver con diferentes tipos de redes neuronales.

Las imágenes de MNIST contienen los números del 0 al 9 y están representadas como matrices de 28x28 elementos en donde cada elemento representa un pixel en la imagen, como se puede observar en la Ilustración 20. Las imágenes están almacenadas en escala de grises, por lo que cada pixel cuenta con un valor entre 0 (negro) y 255 (blanco).

La capa de entrada de la red neuronal empleada para resolver este problema cuenta con 784 neuronas, donde cada neurona equivale a un pixel en la imagen (28x28 = 784 pixeles). La capa de salida tiene 10 neuronas, cada una representando un dígito del 0 al 9. La neurona de salida que tenga el valor mayor representa la predicción de la red neuronal. El modelo de red neuronal empleado para resolver este problema se puede observar en la Ilustración 21.

El número de capas ocultas en el modelo, así como el número de neuronas por capa puede variar. Para este trabajo se prueba con 1, 2 o 3 capas ocultas y 16, 32, 128 y 256 neuronas por capa oculta. El propósito de estas variaciones es el de agregar complejidad a la red esperando una convergencia más rápida. La activación en cada neurona es la función sigmoide.

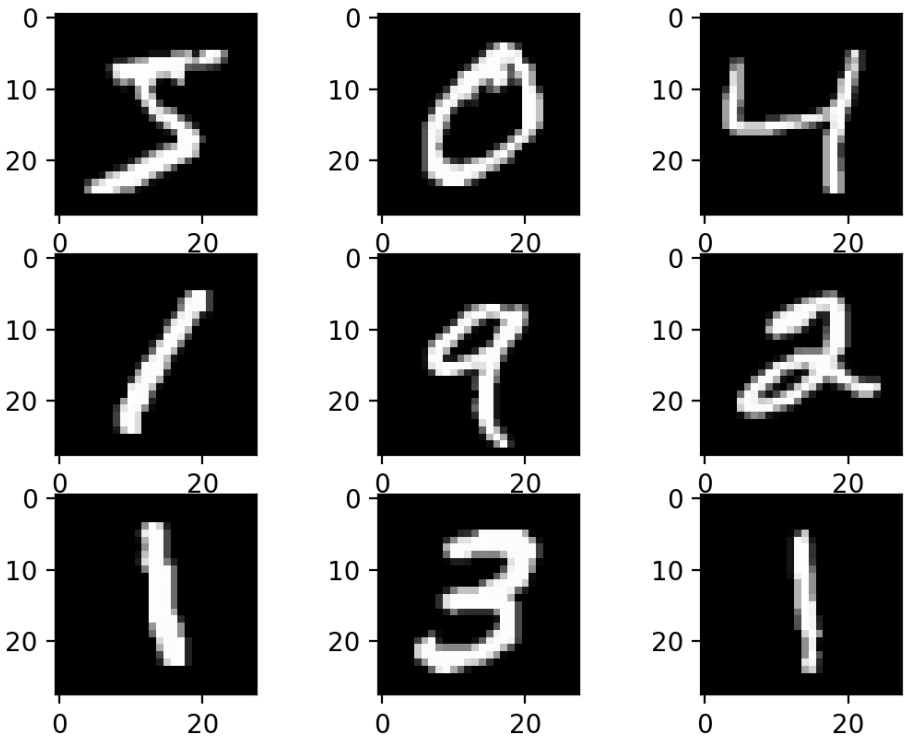
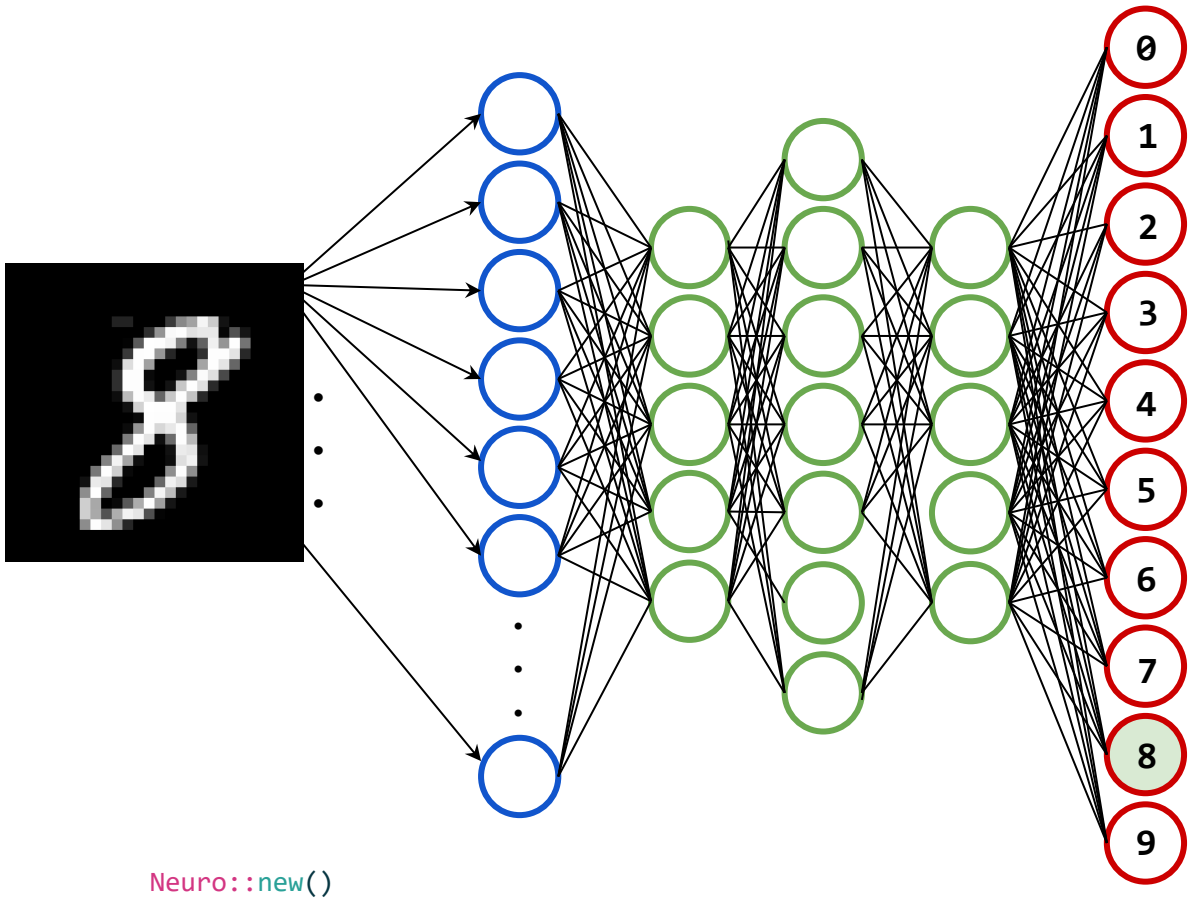


Ilustración 20. Ejemplo imágenes del conjunto de datos MNIST.




```
Neuro::new()
  .add_layer(Dense::new(layer_1_size, Activation::Sigmoid))
  .add_layer(Dense::new(layer_2_size, Activation::Sigmoid))
  .add_layer(Dense::new(layer_3_size, Activation::Sigmoid))
  .add_layer(Dense::new(10, Activation::Sigmoid))
  .train(&X, &y, &test_X, &test_y, learning_rate, epochs, batch_size);
```

Ilustración 21. Grafo de red neuronal para el problema MNIST y código del modelo. 784 neuronas de entrada (azul), número de capas ocultas variable (verde) y 10 neuronas de salida (rojo).

4.1.3. Hardware utilizado

Para el desarrollo de la biblioteca descrita en este trabajo, así como los experimentos realizados se emplean las siguientes especificaciones de hardware:

- **CPU:** Procesador con cuatro cores Intel i5-3330 con 3.00GHz y 16 GB de RAM. Soporte para OpenCL 1.2.

- **GPU:** Sapphire AMD Radeon R7 250 con 1 GB de memoria de video. Soporte para OpenCL 1.1.

4.2. Red neuronal en Rust

En esta sección se describen de cada una de las partes que conforman la biblioteca para el modelado y entrenamiento de redes neuronales implementada para este trabajo. La Ilustración 22 contiene una representación de alto nivel de las dichas secciones.

- Módulo de operaciones de álgebra lineal (`linalg`)
- Módulo para realizar cómputo en el GPU (`oclot`)
- Definición de capas (`Layer`)
- Modelado y entrenamiento de red neuronal (`Neuro`)

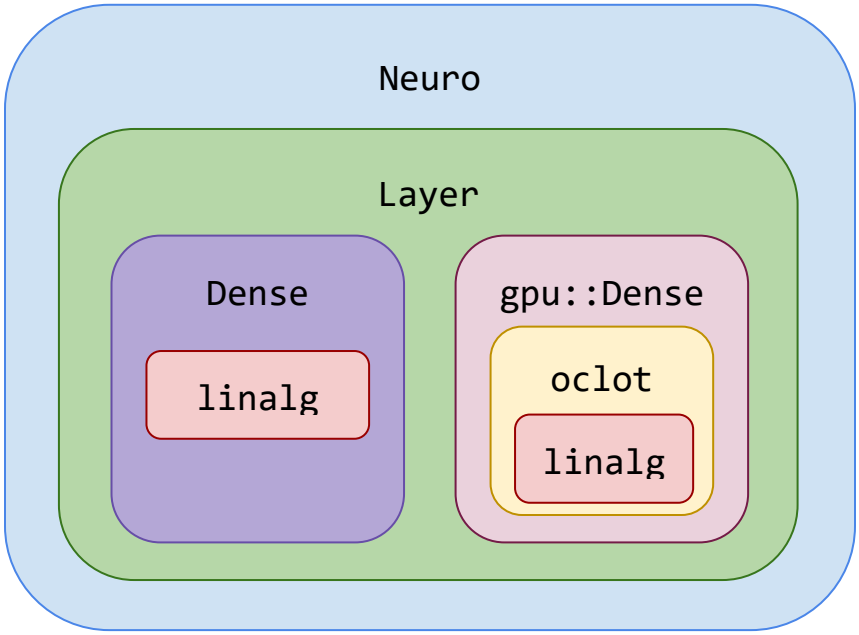


Ilustración 22. Descripción de alto nivel de las capas de la biblioteca desarrollada en este trabajo (Neuro).

4.2.1. Módulo `linalg`

Anteriormente se menciona que el entrenamiento de una red neuronal implica un gran número de operaciones que resultan exhaustivas para el procesador. Dichas operaciones son todos aquellos procedimientos de álgebra lineal que forman parte del algoritmo *feedforward-backpropagation*. Algunos ejemplos son el producto punto de matrices, obtener la transpuesta de una matriz o la suma de dos vectores. Es por este motivo que una de las partes más importantes de una biblioteca para el entrenamiento de redes neuronales es el contar con una biblioteca que se encargue de realizar de forma eficiente dichos cálculos. Es con esto en mente que se desarrolla el módulo `linalg` para este trabajo.

El módulo `linalg` implementa diversas operaciones de álgebra lineal basándose completamente en el uso de iteradores y las abstracciones *zero-cost* de Rust. El módulo define una estructura llamada `Mtx` la cual

sirve para representar matrices mediante un vector de flotantes de 32 bits (`raw`) y una tupla de dos elementos que describe la forma de la matriz (`shape`).

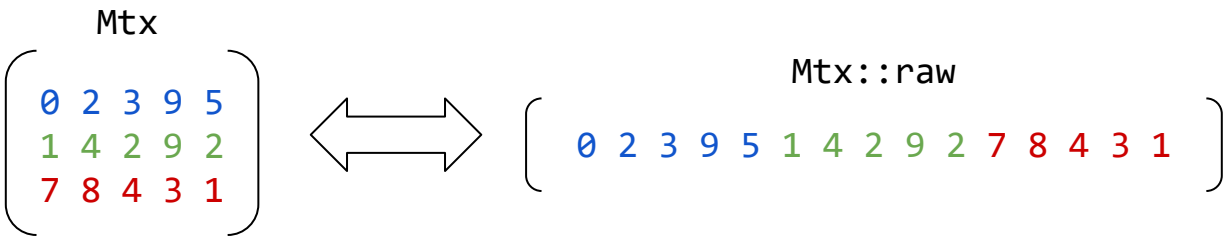


Ilustración 23. Representación de una matriz en un vector raw.

El siguiente bloque de código muestra la implementación del método `add`, el cual realiza la suma de dos matrices. El método se enfoca completamente a iterar sobre los vectores `raw` de cada matriz, realizar el cómputo deseado y así generar una tercera matriz la cual contenga el vector resultante (líneas 6-12). Si bien, cada método dentro de `linalg` realiza operaciones distintas sobre las matrices, todas emplean iteradores de Rust de forma similar para garantizar operaciones eficientes sobre los vectores `raw`.

```
1   pub fn add(&self, other: &Mtx) -> Mtx {
2       if self.shape != other.shape {
3           panic!("invalid shape");
4       }
5
6       Mtx {
7           shape: (self.shape.0, self.shape.1),
8           raw: self.raw.iter()
9               .zip(&other.raw)
10              .map(|(&a, &b)| a + b)
11              .collect()
12      }
13  }
```

El módulo `linalg` implementa las siguientes operaciones sobre la estructura `Mtx`:

- `fn new(shape: (usize, usize), raw: Vec<f32>) -> Mtx`
  - Crea una nueva instancia de Mtx a partir de la tupla de filas y columnas y el vector de datos.
- `fn trans(&self) -> Mtx`
  - Obtiene la transpuesta de la matriz.
- `fn show(&self)`
  - Muestra en pantalla la matriz formateando la salida para favorecer la lectura.

- `fn shape(&self) -> (usize, usize)`
  - Regresa la tupla (filas, columnas)
- `fn size(&self) -> usize`
  - Regresa el número total de elementos en la matriz
- `fn add(&self, other: &Mtx) -> Mtx`
  - Realiza la suma de dos matrices.
- `fn add_vector(&self, vec: &Vec<f32>) -> Mtx`
  - Suma el vector dado a cada una de las filas de la matriz.
- `fn dot(&self, other: &Mtx) -> Mtx`
  - Realiza el producto punto de dos matrices.
- `fn func<F: Fn(&f32)->f32>(&self, f: F) -> Mtx`
  - Recibe una función la cual será aplicada a cada uno de los elementos de la matriz.
- `fn prod(&self, other: &Mtx) -> Mtx`
  - Realiza la multiplicación uno a uno de dos matrices.
- `fn sum_cols(&self) -> Mtx`
  - Suma todas las columnas de una matriz.
- `fn sum_rows(&self) -> Mtx`
  - Suma todas las filas de una matriz.
- `fn get_raw(&self) -> Vec<f32>`
  - Obtiene una copia del vector interno de la matriz.
- `fn get_row(&self, index: usize) -> Mtx`
  - Obtiene una copia de una fila en la matriz.
- `fn reorder_rows(&self, index: &Vec<usize>) -> Mtx`
  - Regresa una copia de la matriz original reordenando las filas de la matriz de acuerdo con el vector de índices recibido.

El módulo `linalg` se centra completamente en la definición de matrices mediante la estructura `Mtx` así como las operaciones que se pueden realizar sobre las mismas. Todo el cómputo de `linalg` se lleva a cabo mediante operaciones nativas de Rust que se traducen a instrucciones en el CPU, de manera que no agrega ninguna dependencia externa al proyecto.

4.2.2. Módulo `oclot`

El módulo `oclot` se desarrolla con la finalidad de extender las capacidades de la estructura `Mtx` para delegar el cómputo de algunas operaciones definidas en `linalg` hacia el GPU mediante OpenCL. Las operaciones de matrices definidas en `linalg` son implementadas en *kernels* de OpenCL y expuestas mediante la estructura `oclot::Oclot`. El siguiente bloque de código muestra el *kernel* equivalente al método `add` en `linalg`.

```
1 __kernel void add(const int rows, const int cols,
2         const __global float* A, __global float* B) {
3     const int gRow = get_global_id(0);
5     const int gCol = get_global_id(1);
6
7     if (gRow < rows && gCol < cols) {
8         B[gRow*cols + gCol] += A[gRow*cols + gCol];
9     }
10 }
```

El kernel `add` recibe el número de filas, el número de columnas y dos búferes de OpenCL (líneas 1 y 2), para posteriormente proceder a calcular la suma de cada elemento de la matriz simultáneamente (línea 8). Para lograr esto, es necesario crear búferes en el GPU, definir los *work-groups*, argumentos del kernel y otras configuraciones de OpenCL. La estructura `oclot::Oclot` encapsula todas estas responsabilidades y otorga una API fácil de usar compatible con la estructura `Mtx`. `oclot::Oclot` se encarga internamente de elegir una plataforma OpenCL así como crear los contextos, búferes de memoria y cargar los *kernels* necesarios para realizar operaciones de álgebra lineal sacando provecho de los GPUs disponibles.

El módulo `oclot` implementa las siguientes operaciones sobre la estructura `Mtx`:

- `fn sum_rows(&mut self, x:&Mtx) -> Mtx`
  - Suma todas las filas de una matriz.
- `fn sum_cols(&mut self, x:&Mtx) -> Mtx`
  - Suma todas las columnas de una matriz.
- `fn trans(&mut self, x:&Mtx) -> Mtx`
  - Obtiene la transpuesta de una matriz.
- `fn dot(&mut self, a:&Mtx, b:&Mtx) -> Mtx`
  - Realiza el producto punto entre dos matrices.
- `fn forward(&mut self, x:&Mtx, w:&Mtx, b:&Vec<f32>) -> Mtx`

- Realiza la evaluación feedforward sobre una matriz de entrada (x), la matriz de pesos y el vector de bias.

- `fn create_buffer(&self, size: usize) -> ocl::Buffer<f32>`
  - Crea un búfer en memoria global en el GPU.

- `fn write_buffer(&self, src:&Vec<f32>, dst:&ocl::Buffer<f32>)`
  - Escribe los datos de un vector en el host hacia un búfer en memoria global en el GPU.

- `fn read_buffer(&self, src:&ocl::Buffer<f32>, dst:&mut Vec<f32>)`
  - Lee los datos de un búfer en memoria global del GPU hacia un vector en el host.

4.2.3. Modelo de Capas (*trait* `Layer`)

En la biblioteca desarrollada para este trabajo se cuenta con un *trait* de Rust llamado `Layer` el cual encapsula todas las operaciones que ocurren a nivel capa durante el entrenamiento de una red neuronal. Los *traits* de Rust son similares al concepto de interfaces en otros lenguajes como Java. El *trait* `Layer` define una serie de funciones que deben ser implementadas por las estructuras que se pretendan utilizar como capas en la red neuronal. Las funciones más importantes que forman parte del trait `Layer` son las siguientes:

- `fn initialize(&mut self, input_size: usize)`
  - Inicializa los componentes internos de la capa. Por ejemplo, inicializa los valores de pesos y bias aleatoriamente en una capa de tipo `Dense`.

- `fn forward(&mut self, x: &Mtx) -> (Mtx, Mtx)`
  - Realiza la evaluación de la red hacia adelante, obteniendo en la capa de salida la predicción de la red neuronal con los valores de pesos y bias hasta el momento. Regresa dos matrices, una con las cachés y otra con las activaciones en la capa.

- `fn backward(&mut self, c: &Mtx, a: &Mtx, delta:&Mtx) -> Mtx`
  - Realiza la propagación del error de la capa siguiente hacia la capa anterior. Regresa una matriz representando el error en la capa actual.

- `fn update(&mut self, rate: f32)`
  - Actualiza los pesos y bias en función al *learning rate* recibido.

- `fn error(&self, c:&Mtx, a:&Mtx, y:&Mtx) -> Mtx`
  - Calcula el error en la capa. Recibe cachés y activaciones de la capa anterior, así como el valor esperado y regresa el error obtenido en la capa. Este método solo se utiliza en la última capa de la red neuronal.

Para este trabajo se desarrollaron dos estructuras que implementan el trait `Layer`: `Dense` y `gpu::Dense`. Ambas estructuras realizan operaciones similares, con la diferencia de que `gpu::Dense` delega el cómputo al GPU. El utilizar el *trait* `Layer` para definir el comportamiento de una capa permite extender el alcance de la biblioteca desarrollada en este trabajo a capas más complejas en el futuro.

4.2.4. API para el modelado, evaluación y entrenamiento de la red (`Neuro`)

Hasta el momento se han analizado las diferentes estructuras y elementos de Rust que forman parte de los detalles de implementación de la biblioteca descrita en este trabajo. En esta sección se describe la estructura `Neuro`, la cual encapsula la API que se utiliza para modelar, evaluar y entrenar la red neuronal.

La estructura Neuro implementa los siguientes métodos:

- `fn new() -> Neuro`
  - Regresa una nueva instancia de la estructura `Neuro`.

- `fn add_layer(mut self, layer: Box<dyn Layer>) -> Neuro`
  - Agrega una capa al modelo de la red neuronal. Internamente, la estructura `Neuro` mantiene un vector de estructuras que cumplen con el *trait* `Layer`, el cual representa el modelo de la red neuronal. Esta operación agrega un nuevo elemento a dicho vector.

- ```
  fn train(mut self, x:&Mtx, y:&Mtx, test_x:&Mtx, test_y:&Mtx,
           learning_rate:f32, epochs:u64, batch_size:usize) -> Neuro
  ```
  - Inicia el entrenamiento de la red neuronal de la siguiente manera:

  1. Se inicializan todas las capas dentro del vector interno de `Layer`.
  2. Por cada época definida en `epochs` se ejecutan los siguientes pasos:
     a. Se realiza un reordenamiento aleatorio de los datos de entrenamiento y validación.
     b. Se crean los mini-batches en función del número especificado en `batch_size`. Por cada mini-batch se realizan los siguientes pasos:
        i. Se realiza la evaluación *feedforward* sobre cada una de las capas, una a la vez, utilizando los datos de salida de una capa como los datos de entrada de la siguiente. Finalmente se guarda una copia de las cachés y activaciones obtenidas de la ejecución *feedforward* de la última capa.
        ii. Se realiza *backpropagation* sobre todas las capas, una a la vez, comenzando con la última y terminando con la primera. Se toman las cachés y activaciones almacenadas en el paso anterior para calcular el error en la última capa y la salida se propaga hacia las capas siguientes.
        iii. Se realiza la actualización de pesos y bias con los datos obtenidos a partir del paso anterior y aplicando la tasa de aprendizaje establecida en `learning_rate`.
  ```

- `fn predict(&mut self, x:&Mtx) -> Result<Mtx, NeuroError>`
    - Realiza una evaluación *feedforward* sobre cada elemento del vector de `Layer` y regresa el resultado obtenido en la última capa.

- `fn on_epoch_with_loss<F:FnMut(u64, u64, f32, f32) + 'static>(mut self, func: F) -> Neuro`
    - Calcula el costo obtenido en cada época y ejecuta una función provista por el usuario.

- `fn on_epoch<F:FnMut(u64, u64) + 'static>(mut self, func: F) -> Neuro`
    - Ejecuta una función provista por el usuario durante cada época.

- `fn save(file: &str) -> Result<(), NeuroError>`
    - Guarda el modelo entrenado en formato HDF5. Si el modelo no se ha entrenado regresa un error.

- `fn load(file: &str) -> Result<(), NeuroError>`
    - Carga el modelo entrenado a partir de un archivo HDF5.

5. RESULTADOS Y DISCUSIÓN

En este capítulo se presentan los resultados obtenidos del desarrollo de este trabajo y una discusión sobre el desempeño alcanzado con la biblioteca desarrollada en comparación a los resultados de la ejecución sobre Keras y Tensorflow.

5.1. Resultados problema XOR

Para obtener un estimado del tiempo y precisión obtenida para el problema XOR, se calculó el promedio de 100 ejecuciones del entrenamiento de la red neuronal con cada uno de los modelos de ejecución: Rust nativo, OpenCL CPU y GPU, y Keras/Tensorflow. Dichos resultados se pueden observar en la Tabla 5 y en la Ilustración 24.

Debido a la simplicidad del problema y a que en este caso en particular se utilizan como datos de entrenamiento todo el universo de posibles entradas de una compuerta XOR, todos los modelos de ejecución fueron capaces de converger con un 100% de precisión.

La solución más rápida fue la ejecución en Rust nativo con un promedio de 0.063 segundos, seguida por Keras y Tensorflow con 8.233 segundos, OpenCL CPU con 9.264 segundos y finalizando con la ejecución más lenta que fue OpenCL GPU alcanzando los 28.3 segundos.

Tabla 5. Resultados problema XOR.

	Rust nativo	**OpenCL CPU**	**OpenCL GPU**	**Keras/TF**
Tiempo (s)	0.063	9.264	28.3	8.233
Precisión	1.0	1.0	1.0	1.0

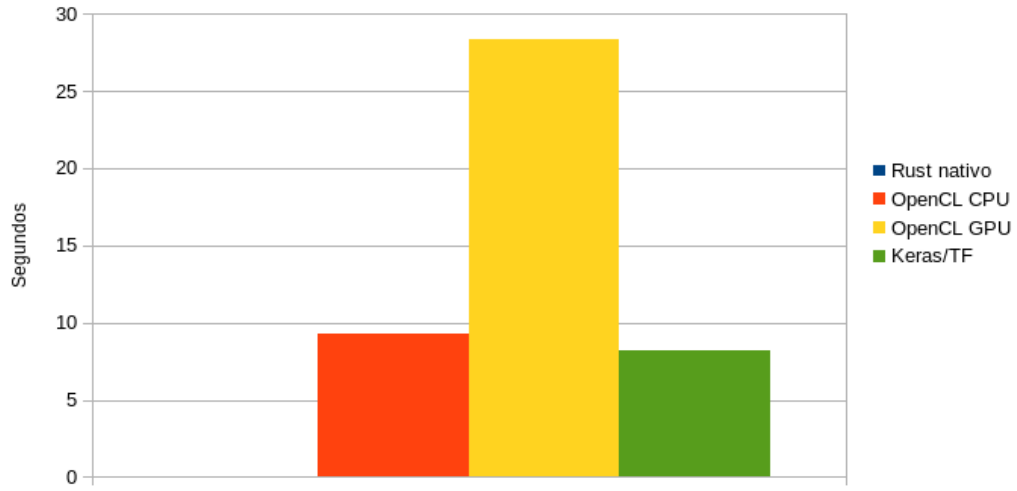


Ilustración 24. Tiempo de ejecución XOR.

5.2. Resultados Problema MNIST

Si bien el problema XOR ayudó a determinar si el algoritmo base del entrenamiento de la red neuronal funciona adecuadamente, no es lo suficientemente complejo para considerar que la biblioteca funciona correctamente en otros modelos de redes neuronales clásicas con múltiples capas ocultas. Es por este motivo que se realizó también una serie de experimentos con el problema MNIST.

En el caso del problema MNIST se realizaron ejecuciones de modelos con 1, 2 y 3 capas ocultas con 16, 32, 64, 128 y 256 neuronas cada una. No se ejecutaron todas las combinaciones posibles, sino que se tomaron las combinaciones más relevantes procurando que la complejidad de la red vaya aumentando paulatinamente. Se realizaron cinco muestras de cada ejecución y se calculó el promedio de estas. Los resultados obtenidos se pueden observar en la Tabla 6.

Tabla 6. Resultados problema MNIST.

	Rust nativo		**OpenCL (CPU)**		**OpenCL (GPU)**		**Keras/Tensorflow**	
Neuronas	**tiempo (s)**	**precisión**	**tiempo (s)**	**precisión**	**tiempo (s)**	**precisión**	**tiempo (s)**	**precisión**
16	98.958	0.9238	66.326	0.9199	105.881	0.9141	33.827	0.9367
32	178.913	0.9374	76.1577	0.9336	115.519	0.9346	36.418	0.946
64	326.822	0.9408	113.056	0.948	133.45	0.9408	44.964	0.9544
128	701.191	0.941	204.217	0.9477	179.707	0.9414	59.6	0.9555
256	1396.105	0.9326	367.595	0.9485	266.666	0.941	87.071	0.9535
16, 16	101.32	0.9274	75.969	0.9229	131.022	0.9296	35.131	0.9375
16, 32	104.823	0.931	77.888	0.9284	134.321	0.9328	35.644	0.9383
16, 64	110.7625	0.9296	82.012	0.9311	136.848	0.9279	36.717	0.9346
16, 128	123.239	0.9331	91.09	0.9372	146.7989	0.9298	39.238	0.9366
16, 256	147.774	0.9291	106.49622	0.9264	165.423	0.9307	43.0	0.9383
32, 32	186.7419	0.939	86.9596	0.9363	144.297	0.9394	40.11	0.9535

32, 64	195.7097	0.941	90.425	0.9406	148.415	0.9418	41.447	0.9544
32, 128	214.8	0.9368	99.916	0.9346	158.404	0.9242	44.267	0.9496
32, 256	252.5	0.9387	117.389	0.9376	177.6068	0.9376	49.755	0.9506
64, 64	350.341	0.947	129.386	0.9445	165.461	0.9399	50.248	0.9545
64, 128	385.411	0.9424	142.812	0.9465	175.536	0.9431	54.076	0.956
64, 256	448.216	0.9392	164.631	0.9414	200.458	0.9422	61.433	0.9573
128, 128	811.7	0.9439	246.5144	0.9496	223.087	0.937	72.177	0.9568
128, 256	921.525	0.9452	273.695	0.9454	254.581	0.9353	82.743	0.9569
256, 256	1885.176	0.9483	490.57	0.9441	379.563	0.9323	122.538	0.9538
16, 16, 16	103.382	0.9279	78.075	0.9233	156.144	0.9258	34.455	0.9294
32, 32, 32	195.39	0.9373	86.77	0.938	172.096	0.9354	40.48	0.9465
64, 64, 64	369.368	0.9414	135.78	0.9435	197.8081	0.9441	52.321	0.957
128, 128, 128	918.819	0.9474	278.778	0.9332	269.616	0.9312	81.215	0.9517
256, 256, 256	2274.149	0.9465	694.4543	0.9363	621.843	0.9356	155.881	0.9514

A diferencia de lo ocurrido durante los experimentos del problema XOR, en el caso del problema MNIST ningún modelo fue capaz de converger en una precisión del 100%. Sin embargo, se logró obtener una precisión mayor al 92% en todos los modelos de ejecución disponibles.

La solución implementada con Keras y Tensorflow fue la que obtuvo los mejores valores de precisión, oscilando entre 93% y 95% en las ejecuciones con 1, 2 y 3 capas, como se puede ver en las Ilustraciones 25, 26 y 27. Las soluciones implementadas con la biblioteca de este trabajo ya sea Rust nativo, OpenCL con CPU y OpenCL con GPU, obtuvieron precisiones bastante cercanas a lo obtenido en Keras oscilando entre 92% y 94%.

Algo interesante a notar es que las ejecuciones en Rust nativo y OpenCL con CPU tienden a tener mejor precisión que las ejecuciones en OpenCL con GPU. Esto se puede deber a que el GPU en el que se realizaron estos experimentos sólo cuenta con soporte de representación de flotantes de 32 bits, mientras que en el CPU se mantiene una representación de flotantes de 64 bits. Esta pérdida de precisión se vuelve más evidente conforme se va aumentando el número de capas, así como el número de neuronas por capa.

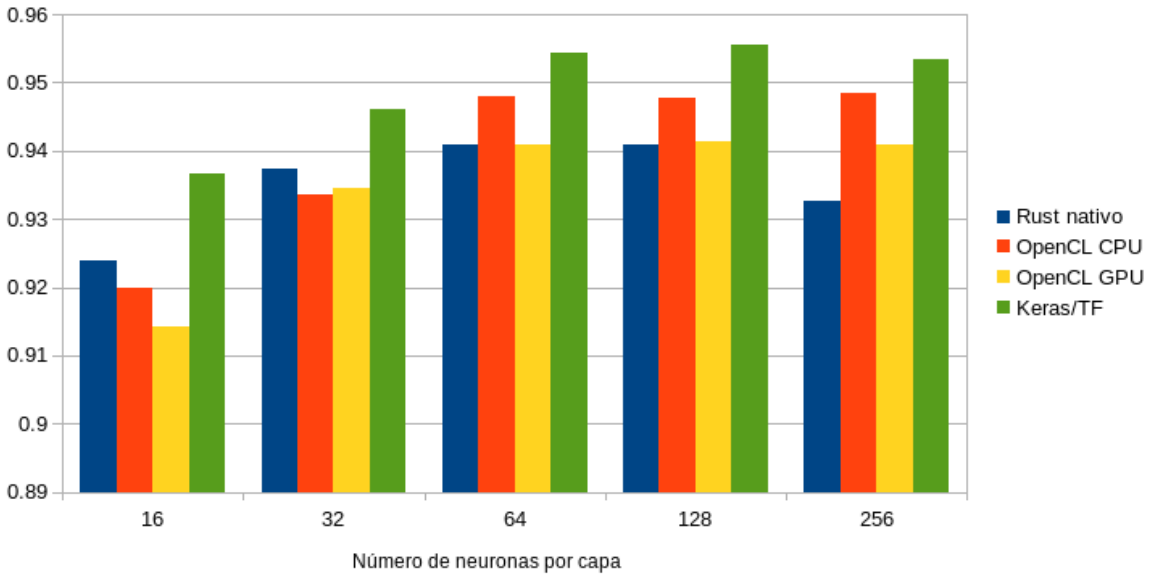


Ilustración 25. Precisión MNIST con una capa.

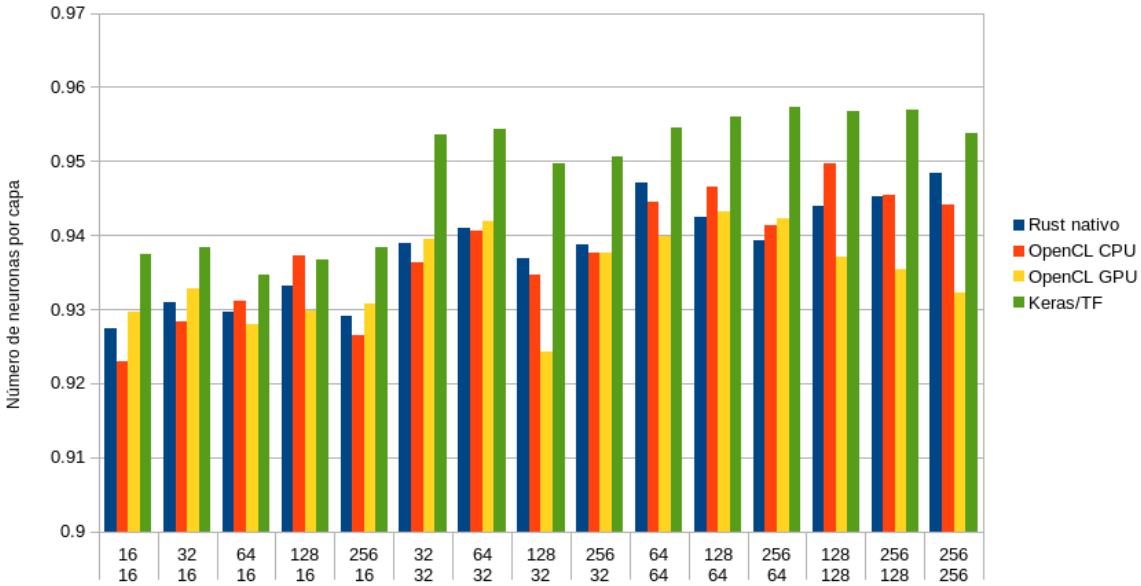


Ilustración 26. Precisión MNIST con dos capas.

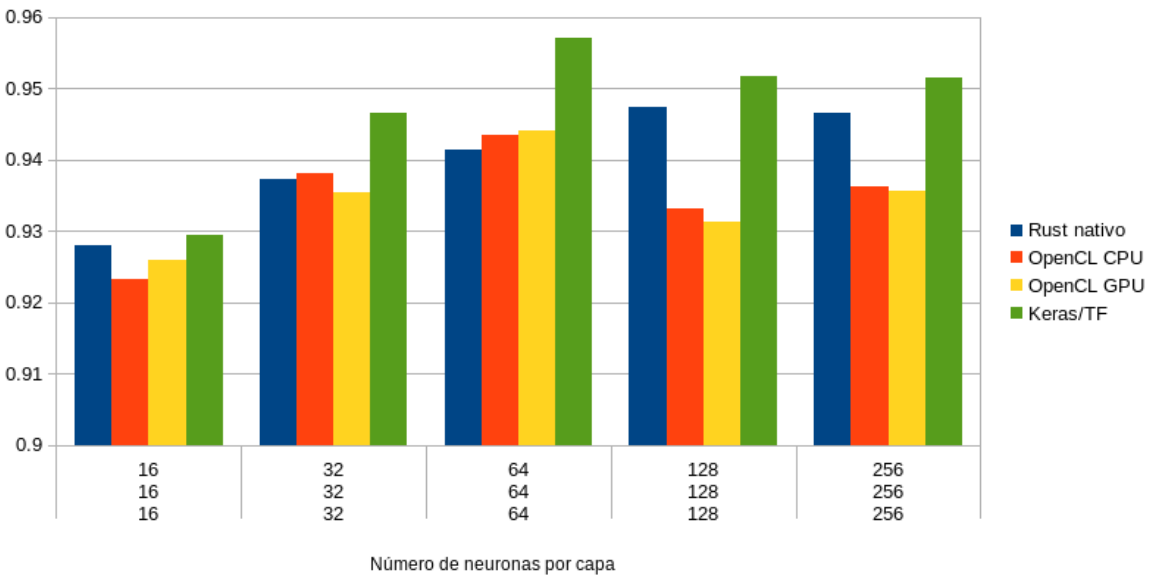


Ilustración 27. Precisión MNIST con tres capas.

En cuanto al tiempo de ejecución, la solución con Keras y Tensorflow también consigue los mejores resultados, siendo capaz de resolver el escenario más sencillo (1 capa oculta de 16 neuronas) en poco más de 30 segundos y el escenario más complejo (3 capas ocultas de 255 neuronas cada una) en aproximadamente 3 minutos, como se puede observar en las Ilustraciones 28, 29 y 30. El peor tiempo de ejecución se obtuvo con la solución en Rust Nativo el cual resolvió el escenario más sencillo en poco más de 1.5 minutos y el escenario más complejo en más de 37 minutos. En el caso de las ejecuciones mediante OpenCL ya sea CPU o GPU se obtuvieron tiempos de ejecución mucho mejores que en Rust Nativo y bastante cercanos entre sí siendo capaces de resolver el escenario más complejo en cerca de 12 minutos para CPU y 10.5 minutos en GPU.

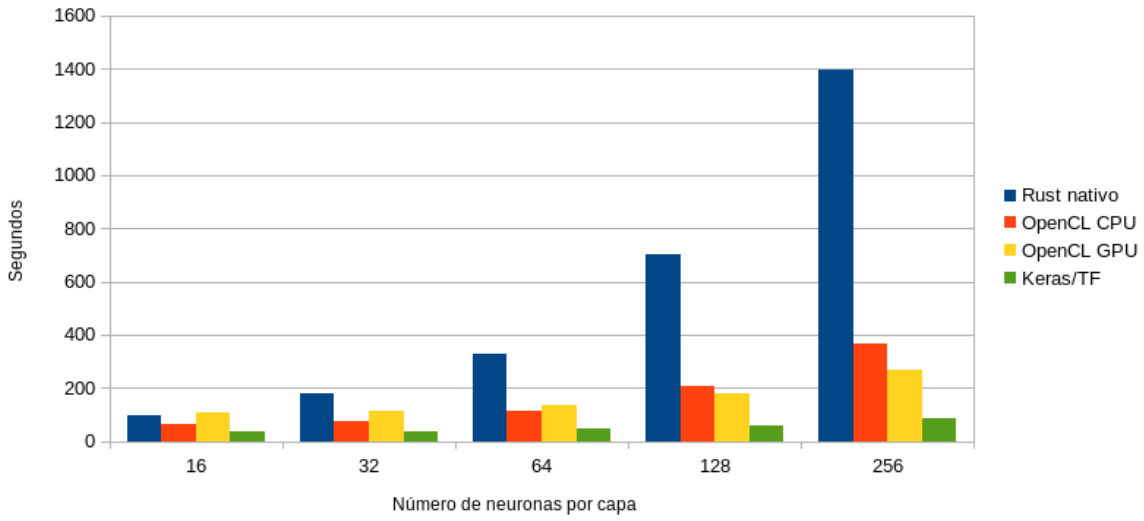


Ilustración 28. Tiempo de ejecución MNIST con una capa.

Algo interesante a notar es que conforme va aumentando el número de capas o el número de neuronas por capa, los tiempos de ejecución en OpenCL GPU van disminuyendo en comparación con OpenCL CPU. A menor número de operaciones, mejor es el desempeño de OpenCL CPU y a mayor número de operaciones mejor es el desempeño de OpenCL GPU.

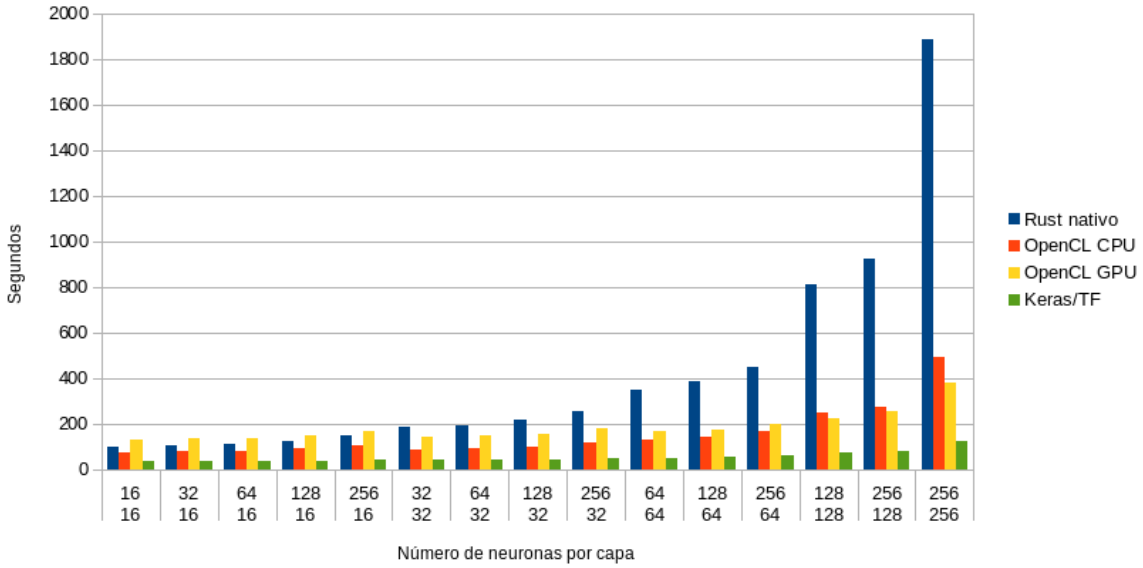


Ilustración 29. Tiempo de ejecución MNIST con dos capas.

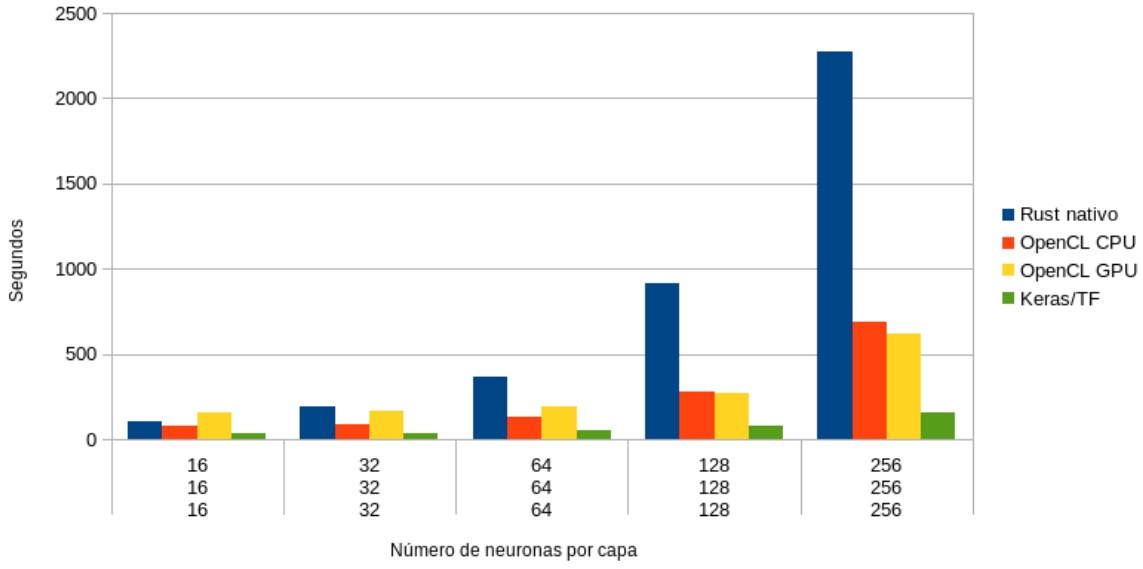


Ilustración 30. Tiempo de ejecución MNIST con tres capas.

5.3. Discusión

El problema de XOR fue de gran utilidad durante el desarrollo de esta biblioteca ya que, al ser un problema de rápida ejecución, permitió monitorear la salud del desarrollo durante cada cambio que ocurría en el mismo. A pesar de la sencillez del problema, los resultados obtenidos durante los experimentos demuestran que los modelos de ejecución basados en OpenCL necesitaron un tiempo de ejecución bastante elevado en comparación con las ejecuciones de Rust nativo y Keras con Tensorflow.

El motivo detrás de esta aparente desventaja de los modelos de ejecución en GPU es que cuando se tienen conjuntos de datos pequeños, el tiempo invertido en trasladar la información del *host* al dispositivo GPU tiende a ser más amplio que la ejecución en sí [19]. Esto se pudo comprobar posteriormente con el problema MNIST, en donde la naturaleza del problema requiere un conjunto de decenas de miles de datos de entrenamiento por lo que las ejecuciones en OpenCL GPU fueron mucho más rápidas que las ejecuciones en Rust nativo u OpenCL CPU. Existen técnicas para mitigar estas limitaciones tales como la transferencia de datos *zero-copy* [20][21]. Dichas técnicas no fueron consideradas en este trabajo, pero representan una mejora significativa sobre los tiempos de ejecución obtenidos.

Así mismo, el punto más notorio que resulta de los experimentos con el problema MNIST es el largo tiempo de ejecución que requiere Rust nativo para procesar los modelos más complejos. Esto se debe sin duda a que el algoritmo implementado en Rust nativo realiza las operaciones de álgebra lineal de forma secuencial, mientras que las implementaciones de OpenCL CPU y GPU trabajan con paralelismo de datos, por lo que son capaces de realizar muchas operaciones simultáneamente sobre distintos datos de entrenamiento, reduciendo así considerablemente el tiempo de ejecución. El ejemplo más obvio de esta disparidad en el tiempo de ejecución es el procesamiento de la multiplicación de matrices, una operación que forma parte vital en cada una de las fases del entrenamiento de una red neuronal. En Rust nativo el procedimiento para realizar una multiplicación de matrices implica 3 ciclos *for* anidados, obteniendo una complejidad de $O(n^3)$ mientras que en las soluciones de OpenCL el algoritmo requiere solo un ciclo for, resultando en una complejidad de $O(n)$. Una posible mejora para el tiempo de ejecución de Rust nativo sería el incorporar varios hilos de ejecución para operar sobre diferentes datos de entrenamiento, obteniendo otra versión de paralelismo de datos similar al de los GPUs. Sin embargo, esto abriría la puerta a otra serie de problemas ya conocidos en aplicaciones multihilo tales como condiciones de carrera, costo de cambio de contexto, deadlocks, entre otros, por lo que esta mejora tampoco se consideró para este desarrollo.

Dejando de lado la diferencia en cuanto al tiempo de ejecución, la precisión obtenida con la biblioteca realizada en este trabajo en comparación con Keras/Tensorflow durante los experimentos del problema MNIST es bastante prometedora. Si bien, Keras/Tensorflow continúa siendo la mejor opción, la precisión obtenida con Rust nativo, OpenCL GPU y OpenCL CPU no se encuentra tan lejana. Como se mencionó en secciones anteriores, en Tensorflow primero se expresa el problema como un grafo de cómputo el cual recibe una serie de transformaciones con el propósito de disminuir el tiempo de cómputo y eliminar operaciones innecesarias que puedan incrementar el error en la red. Este pre-procesamiento no existe en la biblioteca desarrollada en este trabajo ya que la implementación de dichas optimizaciones es un proceso complicado que requiere su propio trabajo de investigación, por lo que no se consideró parte de este desarrollo.

6. CONCLUSIONES

6.1. Conclusiones

Un beneficio que se obtiene al desarrollar una biblioteca de este tipo en Rust es la capacidad del lenguaje para modelar APIs claras y expresivas a pesar de ser considerado de bajo nivel, como se observa en los modelos de redes neuronales desarrollados para resolver los experimentos con XOR y MNIST. Sin embargo, donde realmente destacó Rust fue durante el desarrollo de la biblioteca en sí. El excelente control de Rust sobre el manejo de recursos de memoria e hilos de ejecución representa para el desarrollador una red de seguridad que permite avanzar rápidamente puesto que problemas como fallos de segmento, errores por NULL, desperdicio de memoria, y todo tipo de fallos de concurrencia son detectados en tiempo de compilación. Este aspecto fue de suma importancia ya que el asumir que todos esos problemas serán detectados por el compilador, permitió concentrar los esfuerzos en la lógica de la biblioteca.

Sin embargo, existen escenarios en los que este control estricto que mantiene Rust por defecto sobre los recursos no es deseado y en donde el desarrollador desea ser capaz de realizar la administración de recursos de forma manual. Un ejemplo bastante claro de esto se presentó durante el desarrollo de `oclot`, el módulo que se encarga de realizar operaciones de álgebra lineal sobre búferes de datos mediante OpenCL. Cuando se emplea una biblioteca como OpenCL, es esperado realizar operaciones a nivel punteros de memoria que normalmente el compilador de Rust identificaría como problemas potenciales (copia de búferes, transferencia de datos entre *host* y *device*, entre otras). Rust permite salirse de este control mediante el uso del tag `unsafe`, el cual indica al compilador que el desarrollador se hará responsable de la correcta manipulación de recursos y por lo tanto se ignorarán las verificaciones en dicha sección de código. Esta mezcla entre control y flexibilidad fue lo que hizo de Rust un lenguaje bastante llamativo para el desarrollo de esta biblioteca.

Al desarrollar un proyecto de este tipo es inevitable comparar soluciones de gigantes de la industria tales como Nvidia o Google. Llegar a los números de desempeño de estas frameworks de DL es una labor titánica ya que su soporte involucra equipos completos alrededor del mundo. Si bien este proyecto no pretende competir ni sustituir a estas frameworks, sí propone una alternativa a las soluciones existentes hasta el día de hoy y sobretodo, contribuye a la comunidad de Rust.

6.2. Trabajo Futuro

Este trabajo se concentró completamente en el desarrollo para redes neuronales clásicas también conocidas como *feedforward-backpropagation,* con las que se pudieron resolver los problemas diseñados en este trabajo con una precisión adecuada, pero en donde la etapa de entrenamiento tomó más tiempo que las mismas soluciones implementadas en Tensorflow/Keras. Si bien la división por capas que se realizó para estructurar esta biblioteca y su API permite extender el soporte a redes más complejas como lo son las redes convolucionales o redes recurrentes, es probable que el tiempo de entrenamiento incremente conforme aumenta la complejidad de las redes soportadas. Durante los experimentos desarrollados en este trabajo se pudo identificar que una característica que afecta de forma drástica el tiempo de cómputo durante el entrenamiento es el costo de transferencia de datos entre el *host* y el *device* de OpenCL. Existen varias técnicas para mitigar este problema, siendo una de ellas la transferencia de datos *zero-copy* [20][21]. Un desarrollo a futuro sería implementar la transferencia de datos *zero-copy* para mejorar los tiempos de respuesta y así explorar el desarrollo de otros tipos de redes neuronales más complejas.

Una característica de los proyectos actuales para redes neuronales que no se contempla en este trabajo son las diferentes herramientas que cuentan para la manipulación de grandes cantidades de datos que sirven como elementos de entrenamiento en los modelos *machine learning*. En los experimentos realizados para este desarrollo se emplearon conjuntos de datos limpios, es decir, a los que no fue necesario realizar ningún tipo de normalización o preprocesamiento para su empleo en el entrenamiento de las redes neuronales. Python cuenta con una gran variedad de herramientas que cubren aspectos de este tipo y que son ampliamente empleadas por la comunidad, por lo que otro desarrollo a futuro sería el implementar una API de Python que internamente haga uso de la biblioteca desarrollada en este trabajo.

Finalmente, y a pesar de que este desarrollo se concentra en explorar la viabilidad de emplear OpenCL como la principal interfaz para delegar cómputo hacia el GPU, una interesante mejora sería agregar soporte a CUDA/Nvidia y comparar los resultados obtenidos.

BIBLIOGRAFÍA

[1] A. Ilievski, V. Zdraveski, and M. Gusev, "How CUDA Powers the Machine Learning Revolution," *2018 26th Telecommunications Forum (TELFOR)*. 2018, doi: 10.1109/telfor.2018.8611982.

[2] "Stack Overflow Developer Survey 2019." https://insights.stackoverflow.com/survey/2019/?utm_source=social-share&utm_medium=social&utm_campaign=dev-survey-2019 (accessed Oct. 06, 2020).

[3] T. Uzlu and E. Saykol, "On utilizing rust programming language for Internet of Things," *2017 9th International Conference on Computational Intelligence and Communication Networks (CICN)*. 2017, doi: 10.1109/cicn.2017.8319363.

[4] J. Hale, "Deep Learning Framework Power Scores 2018 - Towards Data Science," *Towards Data Science*, Sep. 20, 2018. https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a (accessed Oct. 06, 2020).

[5] J. Bergstra *et al.*, "Theano: A CPU and GPU Math Compiler in Python," *Proceedings of the 9th Python in Science Conference*. 2010, doi: 10.25080/majora-92bf1922-003.

[6] chrisbasoglu, "The Microsoft Cognitive Toolkit." https://docs.microsoft.com/en-us/cognitive-toolkit/ (accessed Oct. 06, 2020).

[7] A. Paszke *et al.*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," Dec. 03, 2019.

[8] Keras Team, "Keras: the Python deep learning API." https://keras.io/ (accessed Oct. 07, 2020).

[9] "Neural Networks." http://www.arewelearningyet.com/neural-networks/ (accessed Oct. 06, 2020).

[10] tensorflow, "tensorflow/rust." https://github.com/tensorflow/rust (accessed Oct. 06, 2020).

[11] M. Hirn, "Tensorflow wins," *Medium*, May 09, 2016. https://medium.com/@mjhirn/tensorflow-wins-89b78b29aafb (accessed Oct. 06, 2020).

[12] raskr, "raskr/rust-autograd." https://github.com/raskr/rust-autograd (accessed Oct. 06, 2020).

[13] LaurentMazare, "LaurentMazare/tch-rs." https://github.com/LaurentMazare/tch-rs (accessed Oct. 06, 2020).

[14] A. Munshi, "The OpenCL specification," *2009 IEEE Hot Chips 21 Symposium (HCS)*. 2009, doi: 10.1109/hotchips.2009.7478342.

[15] M. Scarpino, *OpenCL in Action: How to Accelerate Graphics and Computation*. Manning Publications, 2012.

[16] J. Blandy and J. Orendorff, *Programming Rust: Fast, Safe Systems Development*. "O'Reilly Media, Inc.," 2017.

[17] S. Klabnik and C. Nichols, *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.

[18] A. Pinho, L. Couto, and J. Oliveira, "Towards Rust for Critical Systems," *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 2019, doi: 10.1109/issrew.2019.00036.

[19] Y. Fujii, T. Azumi, N. Nishio, S. Kato, and M. Edahiro, "Data Transfer Matters for GPU Computing," *2013 International Conference on Parallel and Distributed Systems*. 2013, doi: 10.1109/icpads.2013.47.

[20] S. Kato, J. Aumiller, and S. Brandt, "Zero-copy I/O processing for low-latency GPU computing," *Proceedings of the ACM/IEEE 4th International Conference on Cyber-Physical Systems - ICCPS '13*. 2013, doi: 10.1145/2502524.2502548.

[21] Y. Zhang, B. Zhang, and Z. Zhou, "Zero-Copy Data Transfer for an OpenCL API Remoting System," *2020 IEEE 5th International Conference on Cloud Computing and Big Data Analytics (ICCCBDA)*. 2020, doi: 10.1109/icccbda49378.2020.9095593.

[22] A. Munshi, B. R. Gaster, T. G. Mattson, J. Fung, D. Ginsburg, and P. M. Hanrahan, OpenCL programming guide. Upper Saddle River: Addison-Wesley, 2012.