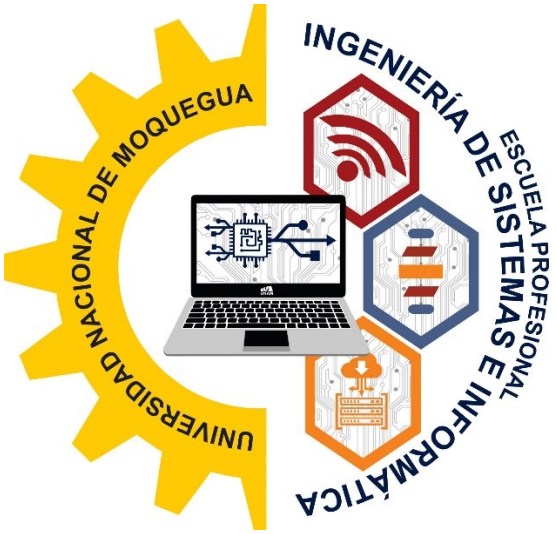
UNIVERSIDAD NACIONAL DE MOQUEGUA



ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMAS E INFORMÁTICA

"ANÁLISIS DE MODELOS PREDICTIVOS BASADOS EN VISIÓN COMPUTACIONAL APLICADOS AL PARALELISMO"

TESIS

PRESENTADO POR EL SEÑOR:
BACH. JUAN CARLOS VALERO GÓMEZ

PARA OPTAR EL TÍTULO PROFESIONAL DE:
INGENIERO DE SISTEMAS E INFORMÁTICA

MOQUEGUA – PERÚ
2019

06

UNIVERSIDAD NACIONAL DE MOQUEGUA

CARRERA PROFESIONAL DE INGENIERÍA DE SISTEMAS E INFORMÁTICA

ACTA DE SUSTENTACIÓN

El Jurado suscribiente ha calificado el trabajo de Tesis:

TITULADO Analisis de modelos predictivos basados en visión computacional aplicados al paralelismo

PRESIDENTE M5c. Victor Yana Mamani

MIEMBROS M5c. Mario Gayna Chino

Mgt. Juan Carlos Clares Perca

ASESOR M5c. Alex Peter Zuñiga Incalla

PRESENTADO POR EL BACHILLER Juan Carlos Valero Gómez

.....DE LA PROMOCIÓN: .....

CUYOS RESULTADOS HAN SIDO LOS SIGUIENTES: .....

CALIFICATIVO DEL TRABAJO 12 (doce)

CALIFICATIVO DE LA SUSTENTACIÓN 12 (doce)

CALIFICATIVO FINAL 12 (doce)

POR LO EXPUESTO, EL BACHILLER Juan Carlos Valero Gómez

HA SIDO DECLARADO EXPEDITO PARA QUE SE LE CONFIERA EL TÍTULO PROFESIONAL DE:

INGENIERO DE SISTEMAS E INFORMÁTICA

EN FE DE ELLO QUEDA ASENTADA LA PRESENTE ACTA

ILO, A LOS 18 DÍAS DE Junio DE 2019

Presidente

Miembro

Miembro

DEDICATORIA

Un agradecimiento especial a mis asesores, quienes me guiaron y trasmitieron sus conocimientos para poder culminar con éxito mi proyecto de tesis.

RESUMEN

En el siguiente informe final de tesis tiene como finalidad presentar los resultados de la investigación del comportamiento de modelos predictivos basados en Visión Computacional cuando están bajo la influencia del paralelismo, es decir cuando los procesos del cálculo computación se segmentan y procesan en paralelo, en este caso se adiciona al CPU el GPU. Si bien las nuevas arquitecturas computacionales son potentes en procesamiento de datos a comparación de décadas anteriores, esto aún no es suficiente para cubrir las necesidades de resultados inmediatos para la toma de decisiones, de aquí la necesidad de poder obtener el mejor algoritmo que dé resultados óptimos cuando esta aplicado al paralelismo. Para se tiene como objetivo comprobar si el paralelismo afecta significativamente a las dimensiones de los algoritmos que son la precisión y el tiempo de procesamiento. Para la experimentación se trabajó con el dataset MNIST que cuenta con 60000 datos para el entrenamiento y 10000 datos para la validación, utilizándose los algoritmos de machine learning como Red Neuronal Artificial (RNA) y los Nearest Neighbor (KNN); y algoritmos de deep learning como Red Neuronal Convolucional (RNC) y Red Neuronal Recurrente Bidireccional (RNCB). Se tomó datos de la experimentación cuando se aplicó con CPU y cuando se aplicó con CPU+GPU, con el objeto de poder aceptar o rechazar las hipótesis planteadas. De los resultados se puede concluir que el paralelismo en el caso de la precisión no afecta al mismo, y en el caso del tiempo de procesamiento hay mejores significativas en los algoritmos KNN, RNC y RNCB, a excepción de la RNA por no contar con mucha complejidad que paralelizar.

Palabras clave: Paralelismo, Visión computacional, Redes Neuronales, Convolucional.

ABSTRACT

The following final thesis report has the purpose of presenting the results of the investigation of the behavior of predictive models based on Computational Vision when they are under the influence of parallelism, in other words, when the computation calculation processes are segmented and processed in parallel; in this case, the GPU is added to the CPU. Even though the new computational architectures are powerful in data processing compared to previous decades, this is still not enough to cover the needs of immediate results for decision making, therefore, the need to be able to obtain the best algorithm that gives optimal results when applied to parallelism. To this end, the objective was to verify if the parallelism significantly affects the dimensions of the algorithms that are the precision and the processing time. For the experimentation, it has been worked with the MNIST dataset that has 60,000 data for training and 10,000 data for validation, using machine-learning algorithms such as Artificial Neural Network (RNA) and Nearest Neighbor (KNN); and deep learning algorithms such as Convolutional Neural Network (RNC) and Bidirectional Recurrent Neuronal Network (RNCB). Experimentation data was considered, when it was applied with CPU and when it was applied with CPU + GPU, in order to accept or reject the hypothesis. From the results, it can be concluded that the parallelism in the case of precision does not affect it, and in the case of processing time, there are significant improvements in the KNN, RNC and RNCB algorithms, excepting the RNA for not having much complexity to parallelize.

Keywords: Parallelism, Computational Vision, Neural Networks, Convolutional.

ÍNDICE

I. PLANTEAMIENTO DEL PROBLEMA ..... - 1 -
1.1. DESCRIPCIÓN DE LA REALIDAD PROBLEMÁTICA ..... - 1 -
1.2. FORMULACIÓN DEL PROBLEMA ..... - 2 -
1.2.1. PROBLEMA GENERAL: ..... - 3 -
1.2.2. PROBLEMA ESPECÍFICO: ..... - 3 -
1.3. JUSTIFICACIÓN E IMPORTANCIA DE LA INVESTIGACIÓN ..... - 3 -
1.4. OBJETIVOS DE LA INVESTIGACIÓN ..... - 4 -
1.4.1. OBJETIVO GENERAL ..... - 4 -
1.4.2. OBJETIVOS ESPECÍFICOS ..... - 4 -
1.5. HIPÓTESIS DE LA INVESTIGACIÓN ..... - 4 -
1.5.1. HIPÓTESIS GENERAL ..... - 4 -
1.5.2. HIPÓTESIS ESPECÍFICAS ..... - 5 -
II. MARCO TEÓRICO ..... - 6 -
2.1. ANTECEDENTES DEL ESTUDIO ..... - 6 -
2.2. BASES TEÓRICAS ..... - 10 -
2.2.1. PARALELISMO ..... - 10 -
2.2.2. GRANULARIDAD ..... - 12 -
2.2.3. COMPUTACIÓN PARALELA ..... - 12 -
2.2.4. VISIÓN COMPUTACIONAL ..... - 13 -
2.2.5. MACHINE LEARNING ..... - 14 -
2.2.6. DEEP LEARNING ..... - 21 -
2.2.6.1. RED NEURONAL CONVOLUCIONAL ..... - 21 -
2.2.6.2. RED NEURONAL RECURRENTE BIDIRECCIONAL ..... - 23 -

2.2.7. TECNOLOGIAS EMERGENTES ..... - 24 -
2.2.7.1. GOOGLE CLOUD PLATFORM ..... - 24 -
2.2.7.2. TENSORFLOW ..... - 26 -
III. MARCO METODOLÓGICO ..... - 28 -
3.1. TIPO Y DISEÑO ..... - 28 -
3.1.1. TIPO DE INVESTIGACIÓN ..... - 28 -
3.1.2. DISEÑO DE LA INVESTIGACIÓN ..... - 28 -
3.1.3. NIVEL DE INVESTIGACIÓN ..... - 29 -
3.2. OPERACIÓN DE VARIABLES ..... - 29 -
3.2.1. VARIABLE INDEPENDIENTE ..... - 29 -
3.2.2. VARIABLE DEPENDIENTE ..... - 30 -
3.3. POBLACIÓN Y MUESTRA ..... - 30 -
3.3.1. LA POBLACIÓN ..... - 30 -
3.3.2. MUESTRA ..... - 30 -
3.4. TÉCNICAS E INSTRUMENTOS ..... - 30 -
3.4.1. SELECCIÓN DE INSTRUMENTOS ..... - 30 -
3.4.2. VALIDACIÓN Y CONFIABILIDAD DEL INSTRUMENTO ..... - 31 -
3.5. DISEÑO EXPERIMENTAL O MÉTODOS Y TÉCNICAS ..... - 31 -
IV. PRESENTACIÓN DE RESULTADOS ..... - 32 -
4.1. EXPLORACIÓN DEL DATASET MNIST ..... - 32 -
4.2. PREPARACIÓN DEL ENTORNO VIRTUAL ..... - 46 -
4.2.1. HABILITANDO EN GPU NVIDIA GEFORCE GTX 980M ..... - 46 -
4.2.2. IMPLEMENTANDO UN ENTORNO CONDA ..... - 48 -
4.2.3. INSTALANDO TENSORFLOW EN ENTORNO CONDA ..... - 48 -
4.3. RESULTADOS ..... - 50 -

4.3.1. RED NEURONAL ARTIFICIAL ..... - 50 -
4.3.2. NEAREST NEIGHBOR ..... - 54 -
4.3.3. RED NEURONAL RECURRENTE BIDIRECCIONAL ..... - 57 -
4.3.4. RED NEURONAL CONVOLUCIONAL ..... - 62 -
4.4. COMPROBACIÓN DE LA HIPOTESIS ..... - 77 -
V. CONCLUSIONES ..... - 82 -
VI. RECOMENDACIONES ..... - 83 -
VII. REFERENCIAS BIBLIOGRÁFICAS ..... - 84 -
VIII. ANEXOS: MATRIZ DE CONSISTENCIA ..... - 88 -

LISTA DE FIGURAS

Figura 1: Aplicaciones de la Visión Computacional ..... - 14 -
Figura 2: Clasificación de los Algoritmos de Machine Learning ..... - 15 -
Figura 3: Modelo Lineal ..... - 16 -
Figura 4: Modelo de Árbol ..... - 17 -
Figura 5: Modelo Red Neuronal ..... - 18 -
Figura 6: Red neuronal totalmente conectada ..... - 19 -
Figura 7: Distancia del vecino más cercano ..... - 21 -
Figura 8: Red Neuronal Artificial ..... - 22 -
Figura 9: Diagrama de una CNN ..... - 23 -
Figura 10: Red neuronal recurrente bidireccional ..... - 24 -
Figura 11: Productos de Google Cloud Platform ..... - 25 -
Figura 12: Diagrama de Tensor RGB ..... - 27 -
Figura 13: Presentación de digito aleatorio ..... - 33 -
Figura 14: Presentación aleatoria de dígitos ..... - 34 -
Figura 15: Valor Total de Píxeles vs Pixel Index ..... - 36 -
Figura 16: Valor Medio de Píxeles vs Pixel Index ..... - 37 -
Figura 17: El dígito promedio de MNIST ..... - 38 -
Figura 18: Frecuencia de Dígito MNIST vs. Dígito ..... - 39 -
Figura 19: Muestreo de dígitos MNIST ..... - 42 -
Figura 20: Promedio del dígito 0 ..... - 43 -
Figura 21: Trazo del promedio del dígito 0 ..... - 44 -
Figura 22: Trazo y valor promedio de cada clase ..... - 45 -
Figura 23: Fuente de descarga de CUDA 9.0 ..... - 47 -
Figura 24: Configurando la ruta PATH de Windows 10 ..... - 47 -
Figura 25: Comprobación de la virtualización CONDA ..... - 49 -

Figura 26: RN con CPU Vs. RN con CPU+GPU ..... - 54 -
Figura 27: KNN con CPU Vs. KNN con CPU+GPU ..... - 57 -
Figura 28: RNRB con CPU Vs. RNRB con CPU+GPU ..... - 61 -
Figura 29: RNC con CPU Vs. RNC con CPU+GPU ..... - 67 -
Figura 30: Distribución el cálculo en paralelo ..... - 68 -
Figura 31: Procesamiento del cálculo en GPU ..... - 69 -
Figura 32: Diagrama Proceso de una RNC ..... - 70 -
Figura 33: Diagrama Proceso de una RNC ..... - 71 -
Figura 34: Proceso detalla de una RNC ..... - 72 -
Figura 35: Filtrando la capa input con una capa 5x5 ..... - 73 -
Figura 36: Función ReLU ..... - 73 -
Figura 37: Max Polling 4x4 ..... - 74 -
Figura 38: Medias comparativas entre los modelos predictivos ..... - 81 -

LISTA DE TABLAS

Tabla 1: Esquema de la experimentación ..... - 28 -
Tabla 2: Definir Dimensiones e Indicadores de VI ..... - 29 -
Tabla 3: Definir Dimensiones e Indicadores de VD ..... - 30 -
Tabla 4: Resultados de la RN ..... - 54 -
Tabla 5: Resultados de KNN ..... - 57 -
Tabla 6: Resultados de RNRB ..... - 62 -
Tabla 7: Resultados de RNC ..... - 67 -
Tabla 8: Resultados globales para cada modelo predictivo ..... - 77 -
Tabla 9: Resultados globales expresados logarítmicamente ..... - 78 -
Tabla 10: Resultados de la precisión para cada modelo predictivo ... - 78 -
Tabla 11: Resultados del tiempo para cada modelo predictivo ..... - 79 -
Tabla 12: Resultados del tiempo expresado logarítmicamente ..... - 80 -

I. PLANTEAMIENTO DEL PROBLEMA

1.1. DESCRIPCIÓN DE LA REALIDAD PROBLEMÁTICA

La visión Computación entre sus ramas tiene a Machine Learning y el Deep Learning los cuales en las últimas décadas han tenido una relevancia importante en el desarrollo de nuevas tecnologías, desde los algoritmos Naïve Bayes, Random Forest, Multilayer Perceptron desarrollados dentro de Machine Learning para el aprendizaje automático (Bansal, Chhikara, Khanna, & Gupta, 2018); hasta las redes neuronales convolucionales del Deep Learning que buscan el entendimiento de los datos y que cada cierto tiempo evolucionan mejorando su precisión (Gu et al., 2017), la búsqueda de la precisión perfecta ha sido una tarea desde los inicios y se ha conseguido desarrollar estos algoritmos que cubren esa necesidad, no obstante, esto ha conllevado a que los costos computacionales se eleven exponencialmente.

En la actualidad la necesidad de poder procesar ingentes cantidades de datos con resultados aceptables en el menor tiempo posible para el reconocimiento de patrones, imágenes, etc., es vital (Qiao, Wang, Li, & Chen, 2018); para ello se han desarrollado varios métodos con diferentes algoritmos para lograr una precisión optima (Romero Ugalde, Carmona, Reyes-Reyes, Alvarado, & Mantilla, 2015), pero el costo computacional es demasiado alto en relación directa con la precisión del algoritmo (Kokkinos & Margaritis, 2018).

Uno de los problemas emblemáticos de la visión computacional es el reconocimiento de manuscritos a mano alzada, el cual ha sido tratado desde varios puntos, desarrollando en las últimas décadas diferentes algoritmos que con el tiempo han sido optimizados y perfeccionados en su eficiencia, pero poder develar cuál de ellos es el adecuado para el reconocimiento de manuscritos aún no es claro, realizar un análisis comparativo entre ellos tanto en su precisión y tiempo de ejecución es importante para marcar el camino adecuado del reconocimiento de imágenes (Impedovo, Mangini, & Barbuzzi, 2014).

Empresas de vanguardia en el uso de tecnologías emergentes tales como Facebook, Google, Amazon, IBM, Microsoft Azure entre otras, están en la búsqueda de nuevos algoritmos o la optimización de los ya existentes para el reconocimiento de imágenes; vemos que Facebook lo utiliza para el reconocimiento de rostros en la infinidad de imágenes que almacena de sus usuarios, aunque en la gran mayoría acierta, eso no es suficiente ya que la empresa busca la perfección (Kim, Kim, & Wise, 2014); otro caso es de Google Maps que busca reconocer direcciones y señales de tránsito del mapeo que realizo hace unos años de las ciudades del mundo, con el objetivo de mejorar su sistema de geolocalización y recomendaciones de rutas de Google Maps (Branson et al., 2018). Ante esta problemática es imperante la necesidad de profundizar en el reconocimiento de imágenes y determinar con cuál de los métodos es el más idóneo para el mismo.

1.2. FORMULACIÓN DEL PROBLEMA

El problema de investigación es planteado mediante un problema general y dos específicos:

1.2.1. PROBLEMA GENERAL:

¿Los modelos predictivos basados en Visión Computacional mejoraran al ser aplicados en la computación paralela?

1.2.2. PROBLEMA ESPECÍFICO:

- ¿Los modelos predictivos basados en Visión Computacional mejoraran la precisión al ser aplicados en la computación paralela?
- ¿Los modelos predictivos basados en Visión Computacional mejoraran el tiempo de procesamiento al ser aplicados en la computación paralela?

1.3. JUSTIFICACIÓN E IMPORTANCIA DE LA INVESTIGACIÓN

La visión computacional cambiará la forma en la que vivimos y trabajamos. De hecho, los algoritmos de reconocimiento y aprendizaje, ya están presente en nuestros días: en los teléfonos, en los vehículos, en la robótica, y en las aplicaciones de las redes sociales (Giancarlo Zaccone, 2017).

El aprendizaje automático está transformando la forma en el que las personas percibimos el mundo. Esta tecnología progresa a la par con los datos y la aparición de las arquitecturas computacionales “la nube” que almacenan y procesan ingentes cantidades de datos (big data) (Haykin, 2008).

El aprendizaje automático tiene un increíble potencial, sobre todo para la industria tecnológica como Google y Facebook, son quienes necesitan, invierten y le dan uso a machine learning en escalas inmensas (Zinchenko, 2017).

1.4. OBJETIVOS DE LA INVESTIGACIÓN

1.4.1. OBJETIVO GENERAL

Evaluar los modelos predictivos basados en Visión Computacional al ser aplicados en la computación paralela.

1.4.2. OBJETIVOS ESPECÍFICOS

Objetivo específico 1:

Evaluar la precisión de los modelos predictivos basados en Visión Computacional al ser aplicados en la computación paralela.

Objetivo específico 2:

Evaluar el tiempo de procesamiento de los modelos predictivos basados en Visión Computacional al ser aplicados en la computación paralela.

1.5. HIPÓTESIS DE LA INVESTIGACIÓN

El sistema de hipótesis está constituido por una hipótesis principal y dos específicas, que se plantea en la siguiente forma:

1.5.1. HIPÓTESIS GENERAL

Los modelos predictivos basados en Visión Computacional mejoran al ser aplicados en la computación paralela.

1.5.2. HIPÓTESIS ESPECÍFICAS

Hipótesis de investigación 1:

Los modelos predictivos basados en Visión Computacional mejoran la precisión al ser aplicados en la computación paralela.

Hipótesis de investigación 2:

Los modelos predictivos basados en Visión Computacional mejoran el tiempo de procesamiento al ser aplicados en la computación paralela.

II. MARCO TEÓRICO

2.1. ANTECEDENTES DEL ESTUDIO

Se han examinado diversas investigaciones similares o afines en las diferentes universidades nacionales y del extranjero que en sus recomendaciones proponen posibles investigaciones a tener en cuenta.

Se mostrarán a continuación investigaciones referidas a la aplicación de la Visión Computacional en las diferentes áreas del saber:

Investigaciones Internacionales

- Las unidades de procesamiento gráfico (GPU) pueden acelerar significativamente las simulaciones de redes neuronales al explotar el paralelismo para cálculos independientes. Tanto los cambios en el potencial de membrana en cada paso de tiempo, y la comprobación de los cruces de umbral de adición para cada neurona, se pueden calcular de forma independiente. Sin embargo, debido a que la transmisión sináptica requiere comunicación entre muchas neuronas diferentes, el procesamiento paralelo eficiente puede verse obstaculizado, ya sea por transferencias de datos entre GPU y CPU en cada paso de tiempo o, alternativamente, ejecutando muchos cálculos paralelos para neuronas que no provocan picos. Esto, a su vez, reduciría el rendimiento efectivo de las simulaciones (Kasap & van Opstal, 2018).

- Las plataformas de computación en clúster han cobrado impulso en los últimos años, debido a su capacidad para analizar grandes cantidades de datos en paralelo. Estas plataformas son entornos complejos y difíciles de administrar. Además, faltan herramientas para comprender mejor y optimizar dichas plataformas que, en consecuencia, forman la columna vertebral de la infraestructura y las tecnologías de big data. Esto lleva directamente a la subutilización de los recursos disponibles y las fallas de las aplicaciones en dicho entorno. Uno de los aspectos clave que pueden abordar este problema es la optimización del paralelismo de la aplicación en dichos entornos (Hernández, Perez, Gupta, & Muntés-Mulero, 2017).

- En los últimos años, el aprendizaje profundo ha llevado a un rendimiento muy alto en una variedad de problemas, como reconocimiento visual, reconocimiento de voz y procesamiento del lenguaje natural. Entre los diferentes tipos de redes neuronales profundas, las redes neuronales convolucionales se han estudiado más ampliamente. Aprovechando el rápido crecimiento en la cantidad de datos y las grandes mejoras en las fortalezas de las unidades de procesadores gráficos, la investigación sobre redes neuronales convolucionales ha surgido rápidamente y ha logrado resultados de vanguardia en diversas tareas (Gu et al., 2017).

- Hoy en día, más y más tareas de reconocimiento de objetos se están resolviendo con Convolutional Neural Networks (CNN). Debido a su alta tasa de reconocimiento y rápida ejecución, las redes neuronales convolucionales han mejorado la mayoría de las tareas de visión por computadora, tanto las existentes como las nuevas. En este artículo, proponemos una implementación del algoritmo de reconocimiento de señales de tráfico utilizando una red neuronal de convolución. El documento también muestra varias arquitecturas CNN, que se comparan entre sí. El entrenamiento de la red neuronal se implementa

utilizando la biblioteca TensorFlow y la arquitectura masivamente paralela para la programación multiproceso CUDA. Todo el procedimiento para la detección y el reconocimiento de señales de tráfico se ejecuta en tiempo real en una GPU móvil. Los resultados experimentales confirmaron la alta eficiencia del sistema de visión por computadora desarrollado (Shustanov & Yakimov, 2017).

- Con el desarrollo continuo del aprendizaje profundo, la red neuronal de convolución con su excelente rendimiento de reconocimiento obtiene una serie de importantes resultados en detección de objetivos, reconocimiento de imágenes y otros campos. Se propone una función de activación de corrección de la segmentación de ReLu mejorada, mejorando la red neuronal de convolución tradicional, agregando la capa de normalización de respuesta local y utilizando el apilamiento máximo, y así sucesivamente. Basado en la plataforma de aprendizaje profundo de Google TensorFlow, la función de activación se utiliza para construir el modelo de estructura de red neuronal de convolución modificada, utilizando el conjunto de datos CIFAR-10 como la entrada de red neuronal para la capacitación y evaluación del modelo (Lin & Shen, 2018).

- El reconocimiento de los dígitos escritos a mano es un problema desafiante en los últimos años. Aunque se estudian muchos algoritmos de clasificación basados en el aprendizaje profundo para el reconocimiento de los dígitos escritos a mano, la precisión del reconocimiento y el tiempo de ejecución aún deben mejorarse. En este documento, se propone una estrategia adaptativa de Q-learning profunda para mejorar la precisión y acortar el tiempo de ejecución para el reconocimiento de dígitos. La estrategia de Q-learning combina la capacidad de extracción de características del aprendizaje profundo y

la toma de decisiones del aprendizaje de refuerzo para formar una red de creencias profundas adaptativa Q-learning (Qiao et al., 2018).

Investigaciones Nacionales

- Para esta investigación se ha usado una base de datos de señales de tránsito en Alemania; sin embargo, por más de que algunas de estas señales sean universales, para implementar el sistema en el Perú se necesita recolectar más data para ser entrenada, sobre todo por diferencias regionales como la señal “PARE” que en Alemania es “STOP”. Existen aproximadamente 100 tipos de señales de tránsito en el Perú, se recomienda por lo menos 5000 muestras por cada señal de tránsito para obtener un buen resultado y 10 millones de muestras de data etiquetada para alcanzar o exceder el rendimiento humano (ARENAS, 2016).

- Para los datasets tabular se observó una mejora respecto a la disminución de dimensiones y aumento de performance de aprendizaje. Ya que, de los 15 data sets, 8 (8/15) 53.33% obtuvieron una mejora en los datos de test usando PCA en comparación con el (5/15) 33.33% usando el método clásico. Asimismo, ambos modelos obtuvieron resultados iguales en un total de (2/15) 13.33%. Este hecho también se refleja al observar el promedio de los resultados para los datos de test, donde PCA obtiene un total de 73.56% en comparación con el método clásico: 71.27%. No obstante, también se observaron problemas de overffiting para la gran mayoría de modelos (método clásico y PCA). La razón reside en la arquitectura usada, ya que, al usar una misma arquitectura, esta podría resultar muy compleja (overffiting) para el tipo de data set (Aguilar, 2019).

- En la presente tesis se propone una red neuronal convolucional profunda para segmentar imágenes de documentos manuscritos históricos, a

diferencia de métodos tradicionales de segmentación de páginas que usan clasificadores entrenados con vectores de características construidos manualmente, el método propuesto extrae características directamente de los parches extraídos de las imágenes, la extracción de características y la clasificación se combinan en un solo paso, es un modelo end-to-end. Los experimentos en el dataset publico Parzival muestran que la calidad de la segmentación mejora cuando se segmenta a las imágenes del conjunto de entrenamiento en un mayor número de súper pixeles. En todos los experimentos se encontró que los modelos entrenados para clasificar los parches, confunden más los parches de tipo página y texto. Los parches de tipo “decoración” y “comentario” son los que han sido clasificados con menor precisión por el modelo, si observamos los resultados podemos observar que los parches de tipo comentario no han podido ser detectados correctamente por el modelo y han sido confundidos en su mayoría por parches de tipo “pagina” y que existe un desbalance entre el número de parches de tipo “comentario” y “decoración” con respecto a la cantidad de parches de tipo “texto” y “pagina”. La métrica de calidad de segmentación que mejora considerablemente cuando aumentamos el número de súper pixeles por imagen son las de precisión de pixeles promedio (Mean pixel accuracy) e intersección sobre la unión promedio (mean IU) (Ascarza Mendoza, 2018).

2.2. BASES TEÓRICAS

2.2.1. PARALELISMO

Todos los principales proveedores de computadoras están introduciendo agresivamente una nueva generación de hardware que incorpora múltiples núcleos en un chip, a veces con capacidades de multihilo simultáneas adicionales y potencialmente junto con otros tipos de hardware, como las

placas aceleradoras. Se espera que el número de núcleos configurados crezca rápidamente, de modo que incluso en el plazo relativamente corto podemos esperar que un solo chip soporte la ejecución de unos pocos cientos de subprocesos. Así, casi todas las plataformas futuras soportarán directamente aplicaciones paralelas. Existe una gran variedad en el diseño de chips, especialmente en la cantidad y naturaleza del intercambio de recursos entre subprocesos en un sistema determinado. Con más memoria en chip y rutas de datos y caché potencialmente compartidas, el uso compartido de recursos puede beneficiarse, y la contención de recursos entre hilos puede afectar de manera adversa el rendimiento. Con su alta densidad de cómputo, el rendimiento del programa y la utilización de la energía deben equilibrarse en el hardware de multiprocesamiento del chip. Las implicaciones para el diseño, desarrollo, despliegue y soporte de software son enormes (John von Neumann, 2002).

Una gran cantidad de aplicaciones secuenciales existentes también deben adaptarse para ejecutarse en estas nuevas plataformas informáticas. Se necesitan con urgencia uno o más modelos de programación de alto nivel si las aplicaciones existentes en negocios, tecnología y ciencia deben migrarse razonablemente sin problemas a ellos. Además de las necesidades típicas bastante conocidas de la comunidad HPC, que tienen experiencia en el desarrollo y migración de aplicaciones de programas paralelos, se deben considerar requisitos como el tiempo de comercialización, el costo de desarrollo y mantenimiento de aplicaciones y la confiabilidad del software. En algunas situaciones, pueden ser suficientes niveles más modestos de paralelismo, incluida la paralelización parcial del programa. Esta generación de plataformas también requiere inherentemente que se tenga en cuenta el consumo de energía y esto puede influir en la estrategia de migración de la aplicación. El lanzamiento de sistemas heterogéneos en el mercado introduce niveles sin precedentes de complejidad arquitectónica en el momento en que se desean con mayor

urgencia enfoques simplificados para la paralelización de programas. Aún no está claro si un solo modelo de programación puede satisfacer todas las necesidades o si, por ejemplo, es posible que un modelo de programación deba combinarse con bibliotecas preempaquetadas o personalizadas para la explotación de ciertos tipos de hardware (John von Neumann, 2002).

2.2.2. GRANULARIDAD

En la computación en paralelo , la granularidad (o tamaño de grano) de una tarea es una medida de la cantidad de trabajo (o cómputo) que realiza esa tarea. Otra definición de granularidad toma en cuenta la sobrecarga de comunicación entre múltiples procesadores o elementos de procesamiento. Define la granularidad como la relación entre el tiempo de cálculo y el tiempo de comunicación, en donde, el tiempo de cálculo es el tiempo requerido para realizar el cálculo de una tarea y el tiempo de comunicación es el tiempo requerido para intercambiar datos entre procesadores.

Si Tcomp es el tiempo de cálculo y Tcomm denota el tiempo de comunicación, entonces la Granularidad G de una tarea se puede calcular. La granularidad generalmente se mide en términos del número de instrucciones ejecutadas en una tarea en particular. Alternativamente, la granularidad también se puede especificar en términos del tiempo de ejecución de un programa, combinando el tiempo de cálculo y el tiempo de comunicación.

2.2.3. COMPUTACIÓN PARALELA

la computación paralela está estrechamente relacionada con el procesamiento paralelo (o computación concurrente). Es la forma de cálculo en la que el uso concomitante ("en paralelo") de varias CPU se lleva a cabo simultáneamente con los sistemas de memoria compartida para resolver un problema computacional de supercomputación. El paralelismo

es el proceso de grandes cálculos, que se pueden dividir en múltiples procesadores que pueden procesarse de forma independiente y cuyos resultados se combinan al finalizar. El paralelismo ha sido empleado durante mucho tiempo en la supercomputación de alto rendimiento.

El procesamiento paralelo generalmente se implementa en el amplio espectro de aplicaciones que requieren grandes cantidades de cálculos. El objetivo principal de la computación paralela es aumentar la potencia de cómputo disponible para sus aplicaciones esenciales. Normalmente, esta infraestructura es donde el conjunto de procesadores está presente en un servidor, o se conectan servidores separados entre sí para resolver un problema informático.

En el primer software de computadora , que ejecuta una sola instrucción (que tiene una sola Unidad Central de Procesamiento (CPU)) a la vez que se ha escrito para el cómputo en serie. Un problema se divide en varias series de instrucciones, y esas instrucciones se ejecutan una tras otra. Sólo una instrucción computacional completa a la vez.

2.2.4. VISIÓN COMPUTACIONAL

Como seres humanos, percibimos la estructura tridimensional del mundo que nos rodea con aparente facilidad. Piensa en lo vívida que es la percepción tridimensional cuando miras un jarrón de flores sentado en la mesa junto a ti. Puede distinguir la forma y la translucidez de cada pétalo a través de los sutiles patrones de luz y sombreado que juegan en su superficie y segmentar sin esfuerzo cada flor desde el fondo de la escena. Al mirar un retrato de grupo enmarcado, puede contar fácilmente (y nombrar) a todas las personas en la imagen e incluso adivinar sus emociones por su apariencia facial. Los psicólogos de la percepción han pasado décadas tratando de comprender cómo funciona el sistema visual y, aunque pueden idear ilusiones ópticas1 para desentrañar algunos de sus

principios, una solución completa a este rompecabezas sigue siendo difícil de alcanzar (Szeliski, 2010).



Figura 1: Aplicaciones de la Visión Computacional
Fuente: (Szeliski, 2010)

2.2.5. MACHINE LEARNING

Machine Learning puede ser ampliamente definida como métodos computacionales que usan la experiencia para mejorar el desempeño de las predicciones, logrando ser estas más precisas. Cuando nos referimos a experiencias hablamos específicamente de la información histórica

recolectada que se utiliza para los procesos de entrenamiento (Mohri, 2012).

Los algoritmos de machine learning, se pueden clasificar en tres grupos como se detalla en la Figura 2:

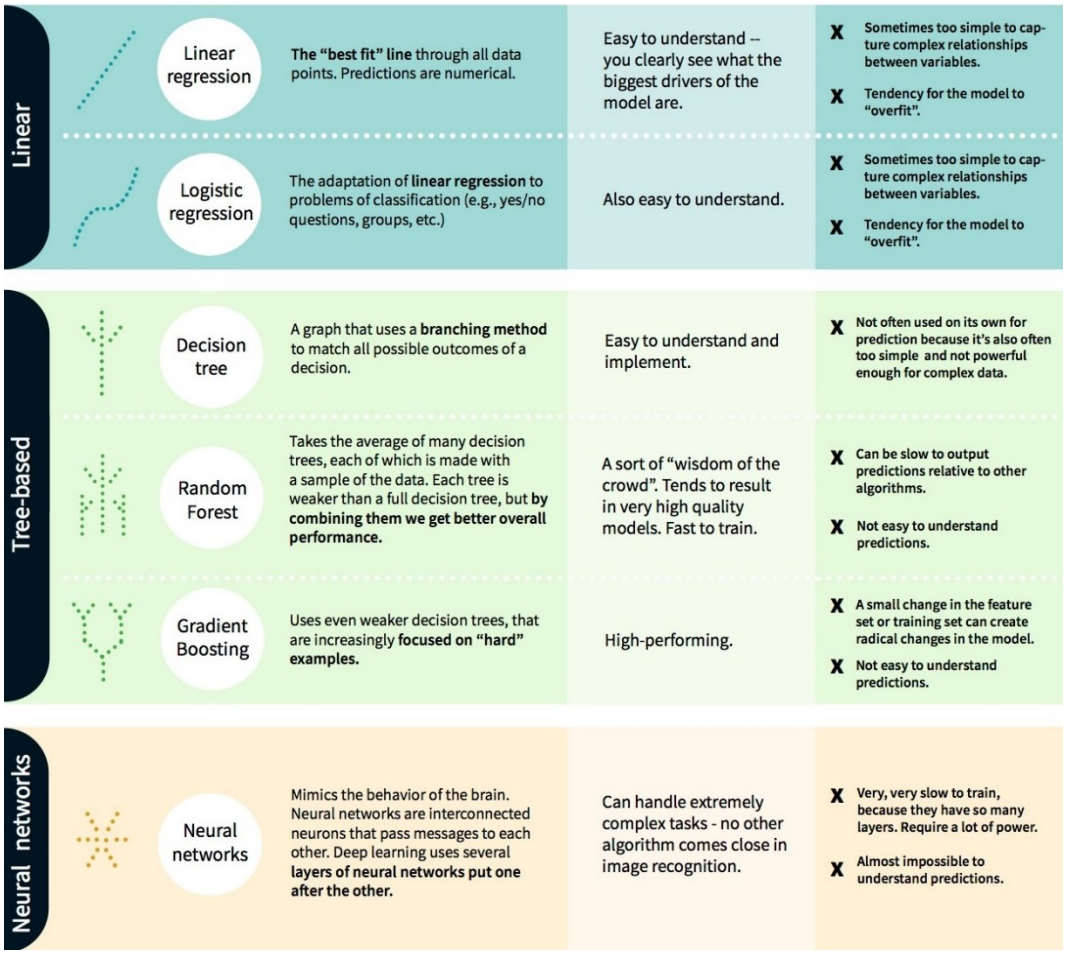


Figura 2: Clasificación de los Algoritmos de Machine Learning
Fuente: www.dataiku.com

- **Modelos lineales**: trata de encontrar una línea que se "ajuste" bien a la nube de puntos que se disponen. Aquí destacan desde modelos muy conocidos y usados como la regresión lineal (también conocida como la regresión de mínimos cuadrados), la logística (adaptación de la lineal a problemas de clasificación -cuando son variables discretas o

categóricas-). Estos dos modelos tienen el problema del “overfit”: esto es, que se ajustan “demasiado” a los datos disponibles, con el riesgo que esto tiene para nuevos datos que pudieran llegar. Al ser modelos relativamente simples, no ofrecen resultados muy buenos para comportamientos más complicados.

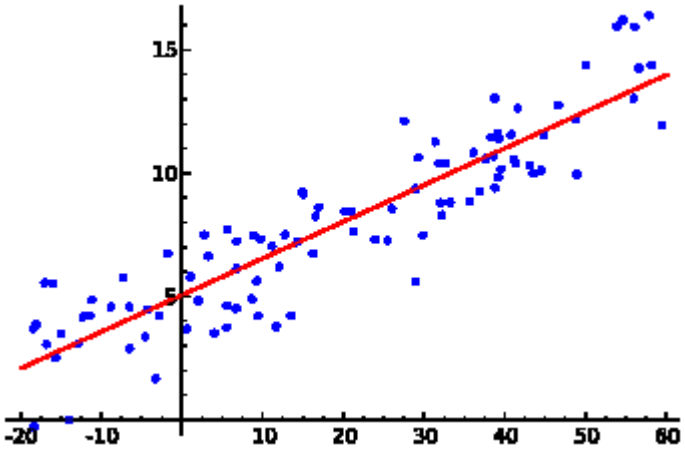


Figura 3: Modelo Lineal
Fuente: es.wikipedia.org

- **Modelos de árbol**: modelos precisos, estables y más sencillos de interpretar básicamente porque construyes unas reglas de decisión que se pueden representar como un árbol. A diferencia de los modelos lineales, pueden representar relaciones no lineales para resolver problemas. En estos modelos, destacan los árboles de decisión y los random forest (una media de árboles de decisión). Al ser más precisos y elaborados, obviamente ganamos en capacidad predictiva, pero perdemos en rendimiento. Nada es gratis.

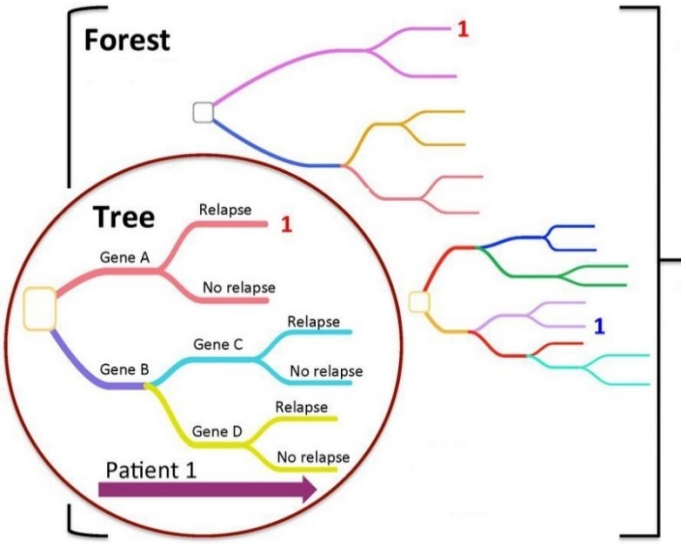


Figura 4: Modelo de Árbol
Fuente: www.biostars.org

- **Redes neuronales**: las redes artificiales de neuronas tratan, en cierto modo, de replicar el comportamiento del cerebro, donde tenemos millones de neuronas que se interconectan en red para enviarse mensajes unas a otras. Esta réplica del funcionamiento del cerebro humano es uno de los "modelos de moda" por las habilidades cognitivas de razonamiento que adquieren. El reconocimiento de imágenes o vídeos, por ejemplo, es un mecanismo complejo que nada mejor que una red neuronal para hacer. El problema, como el cerebro humano, es que son/somos lentos de entrenar, y necesitan mucha capacidad de cómputo. Quizás sea de los modelos que más ha ganado con la "revolución de los datos"; tanto los datos como materia prima, como procesadores de entrenamiento, le vienen como anillo al dedo para las necesidades que tienen (Haykin, 2008).

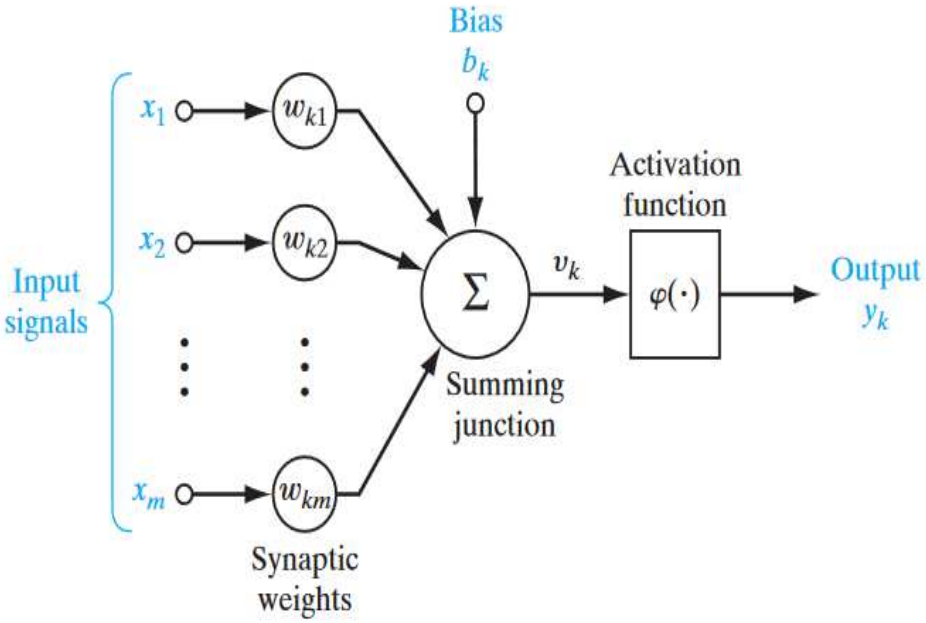


Figura 5: Modelo Red Neuronal
Fuente: Neural Networks and Learning Machines (Haykin, 2008)

2.2.3.1 RED NEURONAL ARTIFICIAL

Existen numerosas formas de definir a las redes neuronales; desde las definiciones cortas y genéricas hasta las que intentan explicar más detalladamente qué son las redes neuronales. Por ejemplo: 1) Una nueva forma de computación, inspirada en modelos biológicos. 2) Un modelo matemático compuesto por un gran número de elementos procesales organizados en niveles. 3) Un sistema de computación compuesto por un gran número de elementos simples, elementos de procesos muy interconectados, los cuales procesan información por medio de su estado dinámico como respuesta a entradas externas. 4) Redes neuronales artificiales son redes interconectadas masivamente en paralelo de elementos simples (usualmente adaptativos) y con organización jerárquica, las cuales intentan interactuar con los objetos del mundo real del mismo modo que lo hace el sistema nervioso biológico (Ruiz & Matich, 2001).

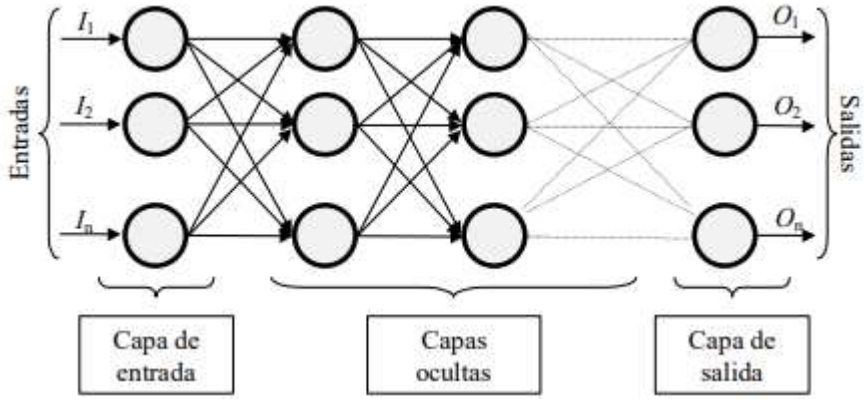


Figura 6: Red neuronal totalmente conectada
Fuente: (Ruiz & Matich, 2001)

Para entender el potencial de la computación neuronal, sería necesario hacer una breve distinción entre sistemas de computación neuronales y digitales: los sistemas neurológicos no aplican principios de circuitos lógicos o digitales. Un sistema de computación digital debe ser síncrono o asíncrono. Si fuera asíncrono, la duración de los impulsos neuronales debería ser variable para mantener uno de los valores binarios por periodos de tiempo indefinido, lo cual no es el caso. Si el 11 principio fuera síncrono, se necesitaría un reloj global o maestro con el cual los pulsos estén sincronizados. Éste tampoco es el caso. Las neuronas no pueden ser circuitos de umbral lógico, porque hay miles de entradas variables en la mayoría de las neuronas y el umbral es variable con el tiempo, siendo afectado por la estimulación, atenuación, etc. La precisión y estabilidad de tales circuitos no es suficiente para definir ninguna función booleana. Los procesos colectivos que son importantes en computación neuronal no pueden implementarse por computación digital. Por todo ello, el cerebro debe ser un computador analógico. Ni las neuronas ni las sinapsis son elementos de memoria biestable. Todos los hechos fisiológicos hablan a favor de las acciones de las neuronas como integradores analógicos, y la eficiencia de la sinapsis cambia de forma gradual, lo cual no es característico de sistemas biestables. Los circuitos del cerebro no

implementan computación recursiva y por lo tanto no son algorítmicos. Debido a los problemas de estabilidad, los circuitos neuronales no son suficientemente estables para definiciones recursivas de funciones como en computación digital. Un algoritmo, por definición, define una función recursiva (Ruiz & Matich, 2001).

2.2.3.2 NEAREST NEIGHBOR (K-NN)

En el reconocimiento de patrones, el algoritmo KNN es un método para clasificar objetos según los ejemplos de entrenamiento más cercanos en el espacio de características. KNN es un tipo de aprendizaje basado en instancias, o aprendizaje perezoso, donde la función solo se aproxima localmente y todos los cálculos se aplazan hasta la clasificación. El KNN es la técnica de clasificación fundamental y más simple cuando hay poco o ningún conocimiento previo sobre la distribución de los datos. Esta regla simplemente retiene todo el conjunto de entrenamiento durante el aprendizaje y asigna a cada consulta una clase representada por la etiqueta de la mayoría de sus vecinos más cercanos a k en el conjunto de entrenamiento. La regla del vecino más cercano (NN) es la forma más simple de KNN cuando K = 1. En este método, cada muestra debe clasificarse de manera similar a sus muestras circundantes. Por lo tanto, si la clasificación de una muestra es desconocida, entonces podría predecirse considerando la clasificación de las muestras vecinas más cercanas. Dada una muestra desconocida y un conjunto de entrenamiento, se pueden calcular todas las distancias entre la muestra desconocida y todas las muestras en el conjunto de entrenamiento. La distancia con el valor más pequeño corresponde a la muestra en el conjunto de entrenamiento más cercano a la muestra desconocida. Por lo tanto, la muestra desconocida puede clasificarse según la clasificación de este vecino más cercano (Imandoust & Bolandraftar, 2013).

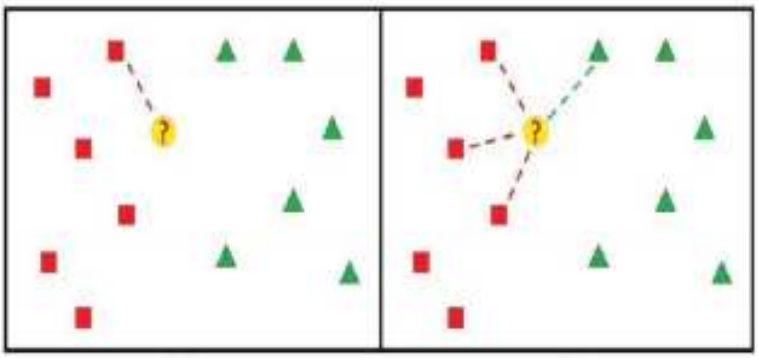


Figura 7: Distancia del vecino más cercano
Fuente: (Imandoust & Bolandraftar, 2013)

2.2.6. DEEP LEARNING

El Deep Learning lleva a cabo el proceso de Machine Learning usando una red neuronal artificial que se compone de un número de niveles jerárquicos. En el nivel inicial de la jerarquía la red aprende algo simple y luego envía esta información al siguiente nivel. El siguiente nivel toma esta información sencilla, la combina, compone una información algo un poco más compleja, y se lo pasa al tercer nivel, y así sucesivamente. Continuando con el ejemplo del gato, el nivel inicial de una red de Deep Learning podría utilizar las diferencias entre las zonas claras y oscuras de una imagen para saber dónde están los bordes de la imagen. El nivel inicial pasa esta información al segundo nivel, que combina los bordes construyendo formas simples, como una línea diagonal o un ángulo recto. El tercer nivel combina las formas simples y obtiene objetos más complejos cómo óvalos o rectángulos. El siguiente nivel podría combinar los óvalos y rectángulos, formando barbas, patas o colas rudimentarias. El proceso continúa hasta que se alcanza el nivel superior en la jerarquía, en el cual la red aprende a identificar gatos (Deshpande, 2018).

2.2.6.1. RED NEURONAL CONVOLUCIONAL

Una red neuronal es un sistema de "neuronas" artificiales interconectadas que intercambian mensajes entre sí. Las conexiones tienen pesos numéricos que se ajustan durante el proceso de

entrenamiento, de modo que una red entrenada correctamente responderá correctamente cuando se le presente una imagen o patrón para reconocer. La red consta de múltiples capas de "neuronas" detectoras de características. Cada capa tiene muchas neuronas que responden a diferentes combinaciones de entradas de las capas anteriores. Como se muestra en la Figura 8, las capas se construyen de modo que la primera capa detecta un conjunto de patrones primitivos en la entrada, la segunda capa detecta patrones de patrones, la tercera capa detecta patrones de esos patrones, y así sucesivamente. Los CNN típicos usan de 5 a 25 capas distintas de reconocimiento de patrones (Samer, Rishi, & Rowen, 2015).

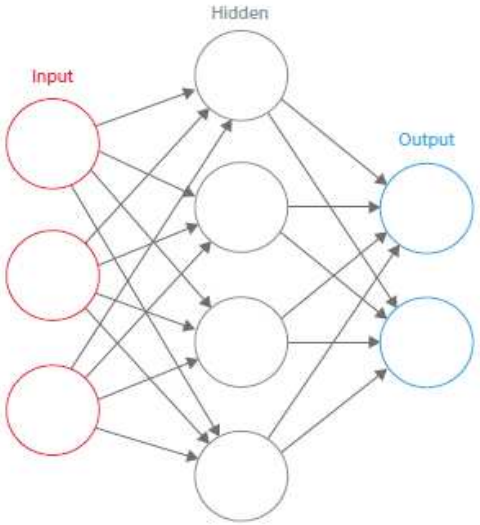


Figura 8: Red Neuronal Artificial
Fuente: (Samer et al., 2015)

En una CNN, las capas de convolución desempeñan el papel de extractor de funciones. Pero no están diseñados a mano. Los pesos del núcleo del filtro de convolución se deciden como parte del proceso de capacitación. Las capas convolucionales pueden extraer las características locales porque restringen los campos receptivos de las capas ocultas para que sean locales.

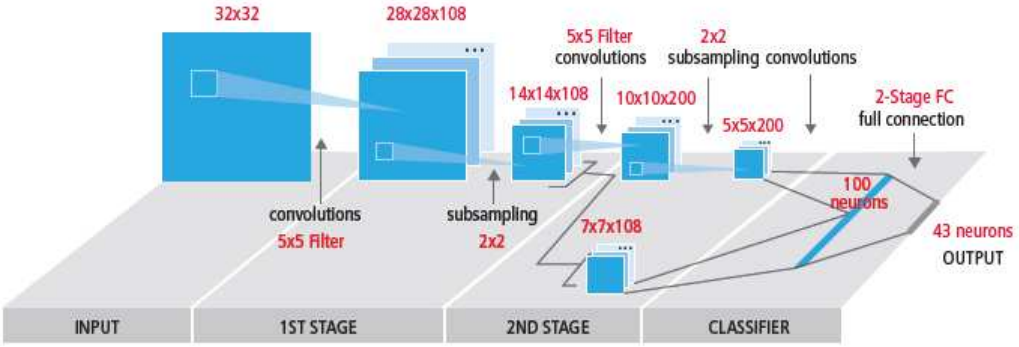


Figura 9: Diagrama de una CNN
Fuente: (Samer et al., 2015)

Los CNN se utilizan en diversas áreas, incluido el reconocimiento de imágenes y patrones, reconocimiento de voz, procesamiento de lenguaje natural y análisis de video. Hay una serie de razones por las cuales las redes neuronales convolucionales se están volviendo importantes. En los modelos tradicionales para el reconocimiento de patrones, los extractores de funciones están diseñados a mano. En CNN, los pesos de la capa convolucional que se utiliza para la extracción de características, así como la capa totalmente conectada que se usa para la clasificación, se determinan durante el proceso de entrenamiento. Las estructuras de red mejoradas de CNN conducen a ahorros en requisitos de memoria y requisitos de complejidad de cómputo y, al mismo tiempo, brindan un mejor rendimiento para aplicaciones donde la entrada tiene correlación local (Samer et al., 2015).

2.2.6.2. RED NEURONAL RECURRENTE BIDIRECCIONAL

Los modelos conexionistas para aprendizaje en los dominios secuenciales son, usualmente, sistemas dinámicos que necesitan estados ocultos para almacenar información contextual. Estos modelos pueden adaptarse a tiempos variables, en dependencia de las entradas del mismo, lo cual puede permitir el trabajo con secuencias de tamaños diferentes. Los problemas de aprendizaje de secuencias, todavía hoy en día se exploran,

a pesar de las numerosas aplicaciones que se han desarrollado. En particular la traslación secuencial es generalmente Una tarea dura y los modelos actuales parecen tener varias limitaciones. Una de estas limitaciones es la suposición de que el sistema es causal. Un sistema dinámico se denomina causal si la salida en el tiempo "t" no depende de entradas futuras. Y se denomina no-causal si en la salida del tiempo "t" se tiene en cuenta la información de los tiempos pasados y futuros. Para ciertas categorías de secuencias finitas, la información del tiempo pasado y futuro es muy importante (Universidad de las Ciencias Informáticas Havana, Martínez, Abed, Ábalo, & Lorenzo, 2006).

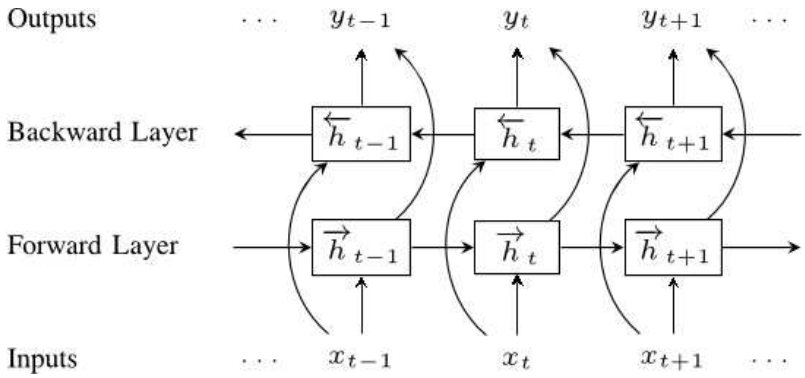


Figura 10: Red neuronal recurrente bidireccional
Fuente: (Havana et al., 2006)

2.2.7. TECNOLOGIAS EMERGENTES PARA LA VISIÓN COMPUTACIONAL

2.2.7.1. GOOGLE CLOUD PLATFORM

Google es una de las compañías que tiene su propia infraestructura y herramientas de software, pero el enfoque de Google es hacer innovaciones en cada capa, sin importar si se trata de hardware, software o red. Su infraestructura no es solo centros de datos en todo el mundo, sino también redes avanzadas con miles de kilómetros de cables de fibra óptica

e incluso su propio cable en el fondo del Océano Pacífico (Zinchenko, 2017).

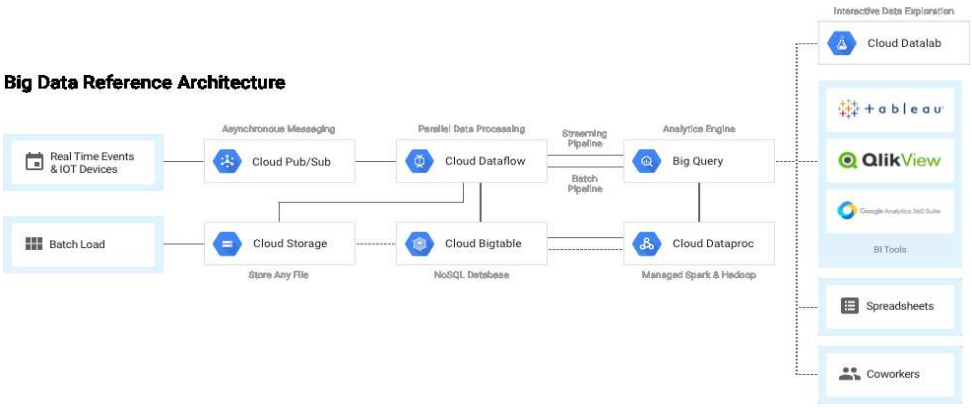


Figura 11: Productos de Google Cloud Platform
Fuente: (Zinchenko, 2017)

Google ha diseñado servidores con ahorro de energía de forma que eliminan piezas innecesarias, como tarjetas de video, y usan fuentes de alimentación que son más eficientes en el uso de la energía. Además, no solo los servidores son eficientes desde el punto de vista energético, sino también toda la información, ya que está compuesta de módulos y cada módulo incluye servidores y equipos de refrigeración. Además, los centros de datos son de carbono neutral, lo que reduce aún más el suministro de energía. Todos estos factores reducen el precio para el usuario final, lo que representa una gran ventaja. Google Cloud Platform pone a disposición de las empresas todos esos productos de Big Data. Es posible usarlos de Google Compute Engine, Google Container Engine y App Engine. Google Compute Engine es un IaaS, el usuario puede cargar cualquier imagen de Linux y ejecutar su código en la máquina virtual en Google Cloud Platform. Google Container Engine es también un IaaS, la gran diferencia para IaaS es que se ejecuta en un contenedor Docker. Con el uso de esas herramientas, es posible usar todos los productos necesarios sin limitaciones a través de la Interfaz de programación de aplicaciones (API) externa. Por el contrario, App Engine es un PaaS y con él no hay necesidad

de configurar el sistema operativo o hacer algunas configuraciones de red, pero tiene limitaciones de plataforma (Zinchenko, 2017).

2.2.7.2. TENSORFLOW

TensorFlow es una biblioteca de software de código abierto que se utiliza para implementar sistemas de aprendizaje automático y aprendizaje en profundidad. Detrás de estos dos nombres se esconden una serie de potentes algoritmos que comparten un desafío común: permitir que una computadora aprenda cómo reconocer automáticamente patrones complejos y tomar las decisiones más inteligentes posibles. Los algoritmos de aprendizaje automático son supervisados o no supervisados; Simplificando tanto como sea posible, podemos decir que la mayor diferencia es que en el aprendizaje supervisado el programador instruye a la computadora sobre cómo hacer algo, mientras que en el aprendizaje no supervisado la computadora aprenderá todo por sí misma.

El aprendizaje profundo es, en cambio, una nueva área de investigación de aprendizaje automático que se ha introducido con el objetivo de acercar el aprendizaje automático a objetivos de inteligencia artificial. Esto significa que los algoritmos de aprendizaje profundo intentan operar como el cerebro humano. Con el objetivo de realizar investigaciones en estas fascinantes áreas, el equipo de Google desarrolló TensorFlow (Giancarlo Zaccone, 2017).

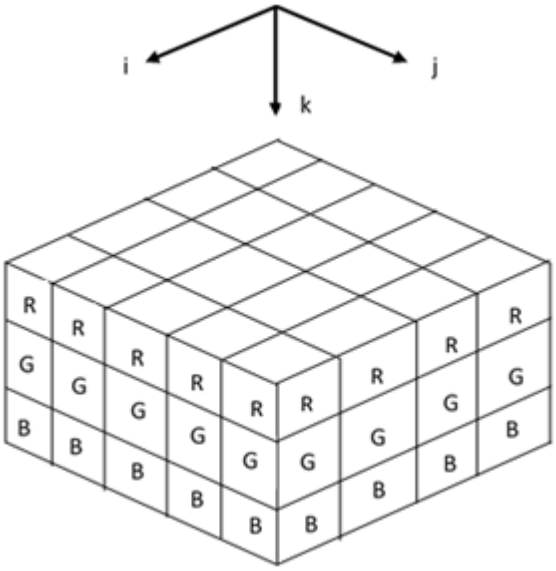


Figura 12: Diagrama de Tensor RGB
Fuente: (Giancarlo Zaccone, 2017)

III. MARCO METODOLÓGICO

3.1. TIPO Y DISEÑO

3.1.1. TIPO DE INVESTIGACIÓN

La presente tesis es una investigación cuantitativa de tipo comparativa-experimental, el estudio tendrá dos grupos: uno experimental y otra de control; testeados antes y después, sobre los modelos predictivos basados en machine learning y Deep learning planteados para la presente investigación.

3.1.2. DISEÑO DE LA INVESTIGACIÓN

El diseño de la presente investigación es Cuasi-Experimental al poner en tratamiento a un grupo experimental (GE) y otro grupo de control (GC) a los cuales se les someterá a un pretest (O1 y O3) y postest (O2 y O4).

Tabla 1: Esquema de la experimentación

GE:	01	X	02
GC:	03	--	04

Dónde:

GE : Grupo experimental

GC : Grupo de control

O1, O3 : Pretest

O2, O4 : Postest

X : Computación Paralela

-- : Computación convencional

Según (Hernandez Sampieri, Fernandez Collado, & Baptista Lucio, 2010) el diseño cuasi experimental se define como la investigación que se realiza para analizar situaciones de causa-efecto pero sin el control riguroso de las variables que maneja el investigador en una situación experimental. El diseño cuasi experimental se divide tomando en cuenta el tiempo durante el cual se recolectan los datos, estos son diseño Transversal, donde se recolectan datos en un solo momento, en un tiempo único, su propósito es describir variables y su incidencia de interrelación en un momento dado, y el diseño longitudinal, donde se recolectan datos a través del tiempo en puntos o periodos, para hacer inferencias respecto al cambio, sus determinantes y sus consecuencias.

3.1.3. NIVEL DE INVESTIGACIÓN

La investigación es de nivel aplicativa, está relacionada con la investigación básica, por depender de los descubrimientos y avances de la investigación básica y se fortalece de ellas. La investigación aplicada busca el conocer para hacer, para actuar, para construir, para modificar.

Al ser investigación aplicada se evaluará a los modelos predictivos basados en machine/deep learning sobre su precisión y el tiempo de procesamiento.

3.2. OPERACIÓN DE VARIABLES

3.2.1. Variable Independiente

X : Computación paralela

Tabla 2: Definir Dimensiones e Indicadores de VI

DIMENSIONES	INDICADORES
Granularidad	- Número de instrucciones ejecutadas. - Número de palabras de memorias usadas. - Duración de su tiempo de ejecución
Grado de Paralelismo	- Número de operaciones que se pueden ejecutar simultáneamente - Número de procesadores a utilizar en paralelo

3.2.2. Variable Dependiente

Y : Modelos predictivos

Tabla 3: Definir Dimensiones e Indicadores de VD

DIMENSIONES	INDICADORES GLOBALES
Precisión	El grado de precisión del modelo planteado
Tiempo de procesamiento	Estimación del tiempo en ejecución del modelo

.

3.3. POBLACIÓN Y MUESTRA

3.3.1. La Población

La población está representada por la base de datos MNIST, que está compuesta por 70 000 muestras.

3.3.2. Muestra

La muestra está constituida por 60 000 muestras de la población de estudio y 10 000 muestras para la validación del modelo, para determinar la proporción de las muestras se ha tomado como punto de referencia investigaciones realizadas con el DataSet MNIST (Kussul & Baidyk, 2004).

3.4. TÉCNICAS E INSTRUMENTOS PARA RECOLECCIÓN DE DATOS VALIDACIÓN Y CONFIABILIDAD DE LOS INSTRUMENTOS

3.4.1. SELECCIÓN DE INSTRUMENTOS

Los instrumentos seleccionados son Pretest y Postest de la precisión y el tiempo de ejecución de los modelos predictivos.

3.4.2. VALIDACIÓN Y CONFIABILIDAD DEL INSTRUMENTO DE INVESTIGACIÓN.

Los instrumentos de medición para la precisión y el tiempo de ejecución de los modelos predictivos planteados, fueron validados en el artículo científico “Using Machine Learning To Optimize Parallelism In Big Data Applications” (Hernández et al., 2017).

3.5. DISEÑO EXPERIMENTAL O MÉTODOS Y TÉCNICAS PARA LA PRESENTACIÓN Y ANÁLISIS DE DATOS

Los datos serán procesados y analizadas cuantitativamente mediante comparación de medias para dos grupos: uno de control y experimental, referidos a los grupos independientes y dependientes y tomando cada uno de los resultados obtenidos se realiza la prueba de hipótesis correspondiente. Además, para validar los modelos predictivos utilizaremos el método Holdout que consiste en separar de manera aleatoria, una parte de los datos de los que se dispone, aplicando una Validación Cruzada y siendo sus métricas la precisión y el tiempo (Kokkinos & Margaritis, 2018).

IV. PRESENTACIÓN DE RESULTADOS

4.1. EXPLORACIÓN DEL DATASET MNIST

Antes de proceder a implementar los modelos predictivos planteados en la presente investigación, primero debemos explorar el dataset MNIST para tener una idea más clara con que datos se va a trabajar, de qué forma debemos abordarlo; y así poder aplicar los modelos predictivos planteados con los datos ya analizados.

Primeramente, cargaremos las librerías necesarias, además que cargamos el dataset MNIST y le asignamos una carpeta donde serán guardados los datos.

```
import matplotlib.pyplot as plt
import numpy as np
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("MNIST_data/", one_hot=False)
```

Ya habiendo cargado el dataset, podemos hacer una primera exploración del mismo, visualizaremos el primer dato del dataset.

```
digito1 = mnist.train.images[0].reshape((28, 28))
plt.imshow(digito1, cmap = 'gray')
plt.show()
```

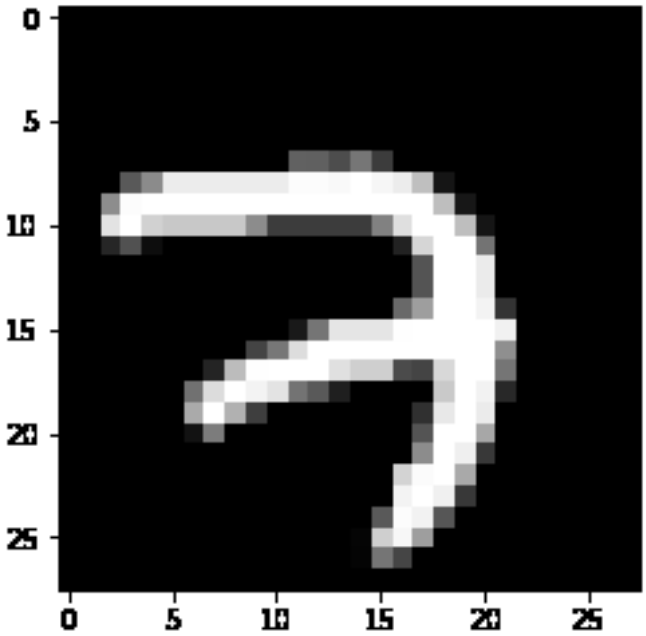


Figura 13: Presentación de digito aleatorio
Fuente: Propia

A través de una función haremos la visualización de un determinado número de imágenes para ver una pequeña muestra de los datos a analizar.

```
def visualizar_imagenes(dataset, cant_img):
    img_linea = 5
    lineas = int(cant_img / img_linea)
    imagenes = []
    for i in range(lineas):
        datos = []
        for img in dataset[img_linea* i:img_linea* (i+1)]:
            datos.append(img.reshape((28,28)))
            imgs = np.hstack(datos)
        imagenes.append(imgs)
    data = np.vstack(imagenes)
    plt.imshow(data, cmap = 'gray' )
```

```
    plt.show()
```

Con esta función podremos visualizar algunos datos, ahora pasaremos a configurar algunos parámetros del mismo.

```
plt.figure(figsize=(8, 8))
visualizar_imagenes(mnist.train.images, 30)
```

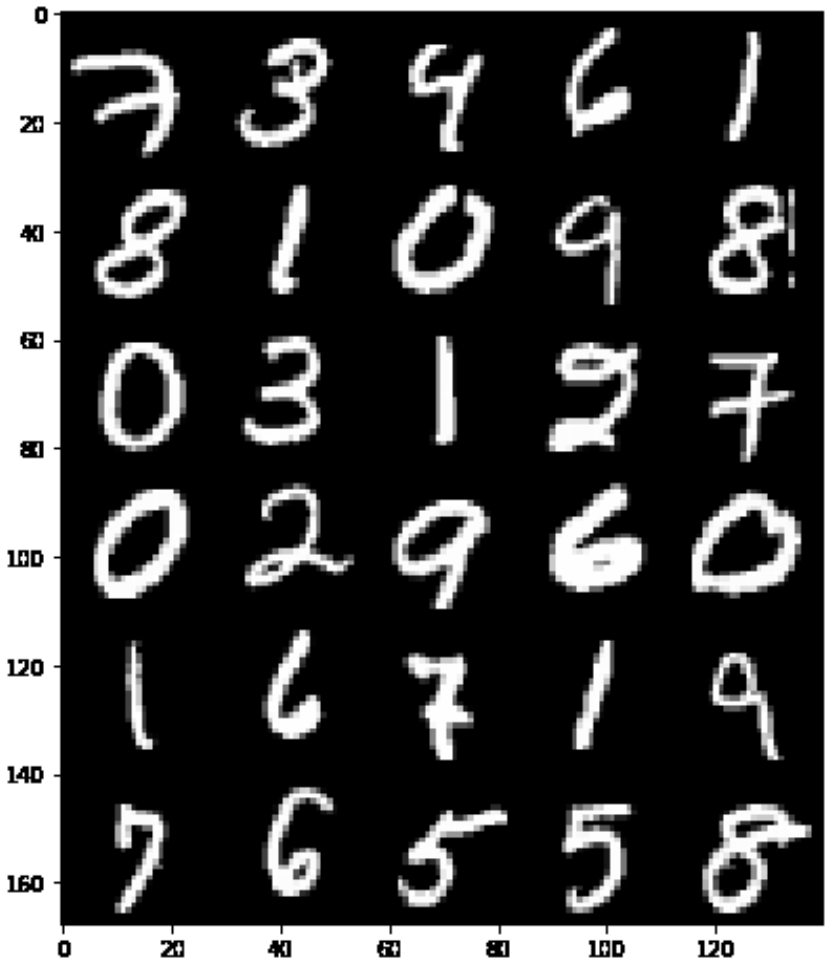


Figura 14: Presentación aleatoria de dígitos
Fuente: Propia

Como se visualiza, se ha reducido cada imagen de 28x28 a 8x8 para poder visualizar una mayor cantidad de imágenes y se ha delimitado a solo

mostrar 30 de las mismas. Para tener una idea más clara de las dimensiones exactas de la data imprimiremos dicha información.

```
print("Forma de los datos de la matrix de imagen:
      {0}".format(mnist.train.images.shape))
print("Forma de los datos de label: {0}".format(mnist.train.labels.shape))
```

```
Forma de los datos de la matrix de imagen: (55000, 784)
Forma de los datos de label: (55000,)
```

Al observar cada una de las imágenes de la Figura N° 14, podemos inferir que los bordes de los mismos no cuentan con algún tipo de trazo, para comprobar esta suposición comprobaremos que el promedio de la suma de los pixeles en la esquina superior izquierda sea igual a 0.

```
sum(mnist.train.images[:,0])/mnist.train.images[:,0].shape[0]
```

```
0.0
```

Verificamos que nuestra suposición es cierta. Ahora se hará lo mismo para cada píxel. Podríamos iterar en todas las 784 filas y usar la fórmula anterior, pero hay una mejor manera de hacerlo con el álgebra lineal. Primero calculamos las sumas de las filas para obtener un vector de 784 x 1 de sumas de filas.

```
SumarFilas = np.dot(mnist.train.images.T, np.ones((55000,1)))
plt.plot(SumarFilas)
plt.title("Valor Total de Píxeles vs. Pixel Index")
plt.ylabel("Valor Total de Píxeles")
plt.xlabel("Pixel Index (28 X 28)")
plt.show()
```

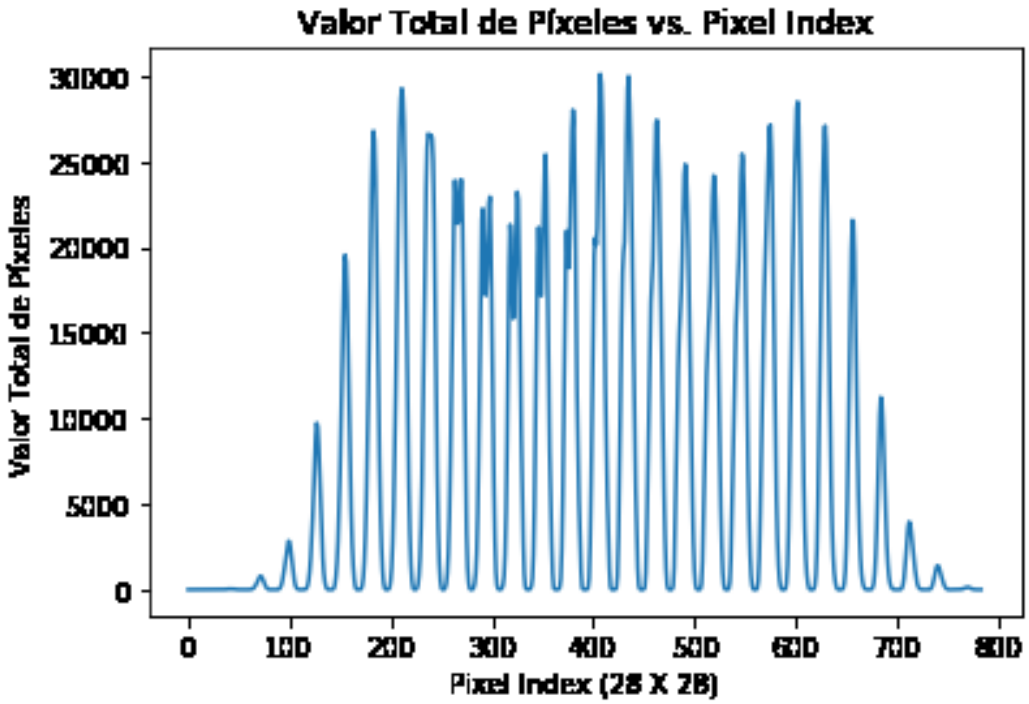


Figura 15: Valor Total de Píxeles vs Pixel Index
Fuente: Elaboración Propia

Este es un resultado bastante interesante. Podemos ver claramente lo que parecen tres curvas de campana. A partir de esto, parece que tres regiones del espacio están más activas en todas las imágenes. Se espera que la gráfica sea una consecuencia del teorema del límite central. Ahora calculemos el valor promedio para cada píxel.

```
valorMedioPixeles = SumarFilas / 55000
plt.plot(valorMedioPixeles)
plt.title("Valor Medio de Píxeles Vs. Pixel Index")
plt.ylabel("Valor Medio de Píxeles ")
plt.xlabel("Pixel Index (28 X 28)")
plt.show()
```

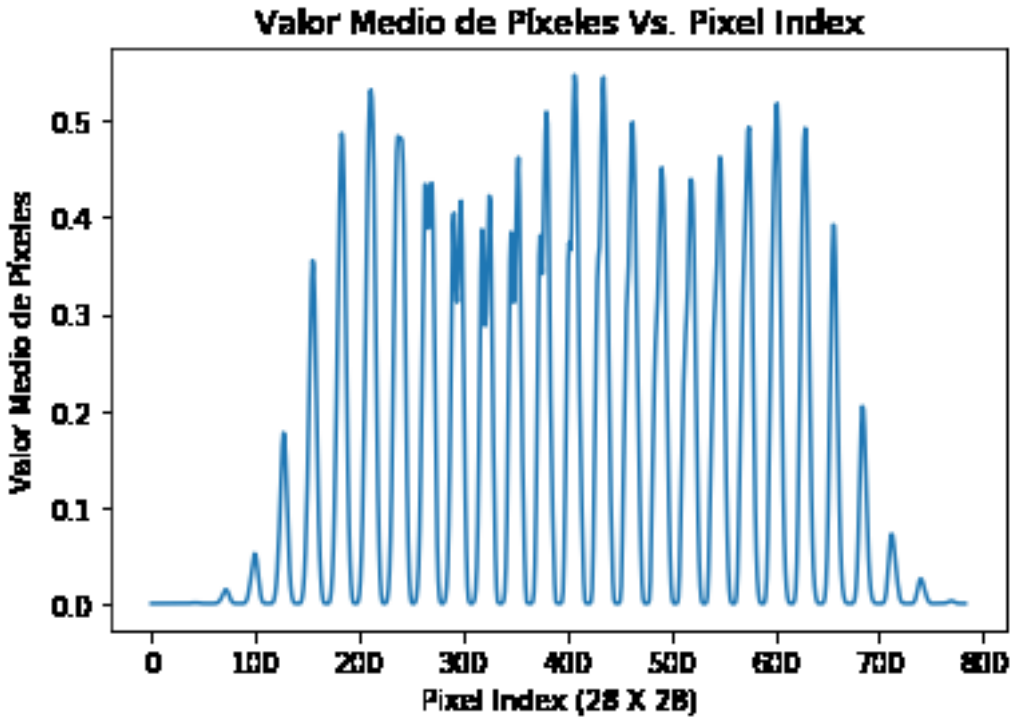


Figura 16: Valor Medio de Píxeles vs Pixel Index
Fuente: Elaboración Propia

Como se visualiza, la normalización de las sumas de las filas, el cálculo de los valores promedio de los píxeles, no cambia la forma de la gráfica, pero nos da valores de píxeles dentro del rango válido de la gráfica [0,1]. Ahora cambiemos la forma de este vector de valor de píxel promedio, de manera que sea 28 x 28 y lo grafiquemos.

```
plt.imshow(valorMedioPixeles.reshape(28,28), cmap='gray')
plt.title("El dígito promedio de MNIST")
plt.show()
```

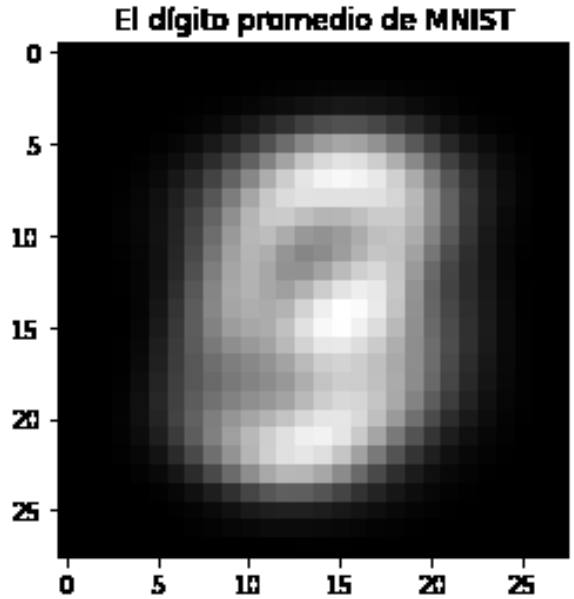


Figura 17: El dígito promedio de MNIST
Fuente: Elaboración Propia

De la Figura 17, podemos tener dos observaciones muy relevantes:

1. El número tres está sobresaliendo.
2. También se visualiza tres regiones principales de grupos blancos dispuestos en una línea vertical. Estas tres áreas corresponden a los tres picos de la curva de campana que observamos anteriormente. Estas regiones representan los píxeles más activos, o las regiones más intersecantes en todos los dígitos. Con esta observación, afirmo que estas son las regiones de menor información cuando se trata de dígitos discriminantes porque estas tres áreas representan la mayor dependencia entre los dígitos (aquí es donde los dígitos tienen la mayoría en común). En el problema de alinear las imágenes para el entrenamiento, podríamos usar esta propiedad, donde movemos las imágenes hasta que maximicemos algunos picos conjuntos.

   Primero investigaremos porque aparece un 3. Podemos suponer lo siguiente:

   1. Tenemos más del digito 3 en nuestro dataset de entrenamiento.

2. Hay un sesgo en las imágenes sobre el digito 3, donde se dibujan de manera más consistente, con menos varianza o, en el mismo sentido, hay duplicados del mismo.

```
unique, counts = np.unique(mnist.train.labels, return_counts=True)
sortedCount = sorted(dict(zip(unique, counts)).items(), key=lambda x: x[1],
                     reverse=True)
sortedCountLabels = [i[0] for i in sortedCount]
sortedCountFreq = [i[1] for i in sortedCount]

plt.bar(range(0,10),sortedCountFreq)
plt.xticks(range(0,10),sortedCountLabels)
plt.title("Frecuencia de Dígitos MNIST Vs. Dígito")
plt.ylabel("Frecuencia de Dígitos")
plt.xlabel("Dígito")
plt.show()
sortedCount
```

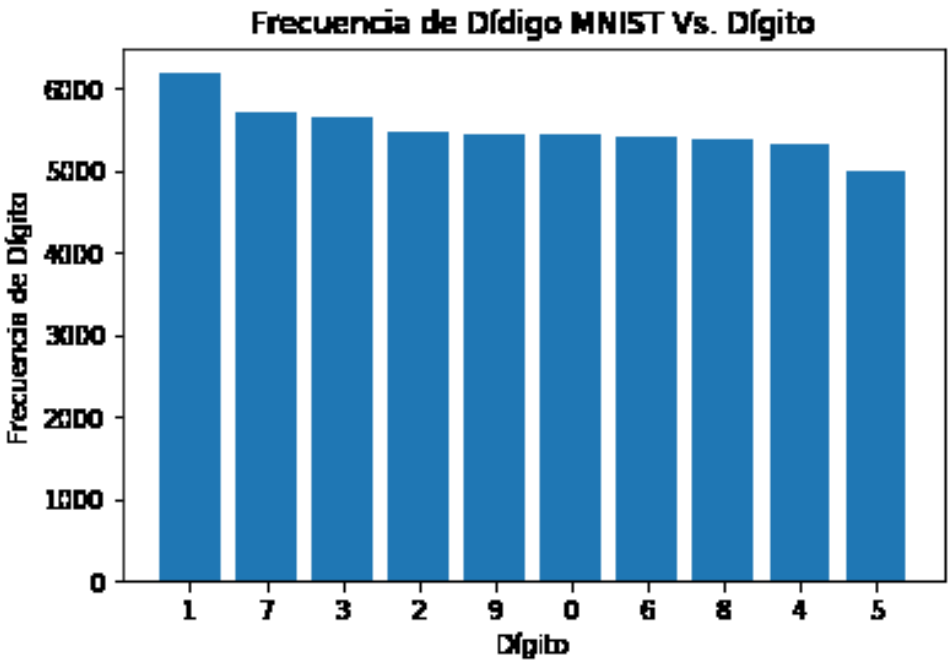


Figura 18: Frecuencia de Dígito MNIST vs. Dígito
Fuente: Elaboración Propia

```
[(1, 6179),
 (7, 5715),
 (3, 5638),
 (2, 5470),
 (9, 5454),
 (0, 5444),
 (6, 5417),
 (8, 5389),
 (4, 5307),
 (5, 4987)]
```

El conjunto de datos está sesgado, pero no predomina el digito de tres. Para entender por qué un tres es tan prominentes en la imagen promedio, hagamos algunas exploraciones más que muestren más propiedades de los datos.

Al comenzar vimos el valor promedio de píxeles en todas las clases de dígitos y vimos algunos resultados interesantes. Ahora veamos el valor de píxel promedio dentro de cada clase de dígitos. Primero vamos a zipear las etiquetas de entrenamiento con las imágenes de entrenamiento.

```
dataWithLabels = zip(mnist.train.labels, mnist.train.images)
digitDict = {}
for i in range(0,10):
    digitDict[i] = []
```

Convertimos esto en un diccionario donde los subconjuntos de las imágenes con respecto a la clase de dígitos se almacenan a través de la clase correspondiente. Se inicializa el dataDict con claves [0,9] y lista vacía.

Ahora se asigna una lista de vectores de imagen a cada índice de clase de dígito correspondiente, y posteriormente convertir las listas en matrices numpy.

```
for i in dataWithLabels:
    digitDict[i[0]].append(i[1])
```

```
for i in range(0,10):
    digitDict[i] = np.matrix(digitDict[i])
    print("Digito {0} - forma de la matriz: {1}".format(i,digitDict[i].shape))
```

```
Digito 0 - forma de la matriz: (5444, 784)
Digito 1 - forma de la matriz: (6179, 784)
Digito 2 - forma de la matriz: (5470, 784)
Digito 3 - forma de la matriz: (5638, 784)
Digito 4 - forma de la matriz: (5307, 784)
Digito 5 - forma de la matriz: (4987, 784)
Digito 6 - forma de la matriz: (5417, 784)
Digito 7 - forma de la matriz: (5715, 784)
Digito 8 - forma de la matriz: (5389, 784)
Digito 9 - forma de la matriz: (5454, 784)
```

En este punto, se ha recortado la matriz de imagen original de todos los dígitos de entrenamiento a matrices más pequeñas de cada dígito. Vamos a visualizar algunos ejemplos de cada clase. Para lograrlo crearemos algunas funciones de ayuda para el trazado.

```
def makeSubplots(nGridRow,nGridCol,figsize=(20,20)):
    sps = []
    fig = plt.figure(figsize=figsize)
    for i in range(1,(nGridRow*nGridCol)+1):
        sps.append(fig.add_subplot(nGridRow,nGridCol,1))
    return (fig,sps)

def turnOffAxes(sps):
    for ax in sps:
        ax.set_xticks([])
        ax.set_yticks([])
```

```
# Vamos a trazar algunos dígitos.
figRows,figCols = (10,4)
fig,sps = makeSubplots(figRows,figCols)
turnOffAxes(sps)

# Traza los dígitos como subconjuntos, 4 por fila de 10 filas.
subplotCounter = 0
for digitClass in range(0,10):
    for spi in range(0,4):
        # Toma una sola columna de la matriz digitClass.
        sps[subplotCounter].imshow(digitDict[digitClass][spi,:]
                                   .reshape(28,28),cmap='gray')
        subplotCounter += 1
plt.suptitle('Muestreo de dígitos MNIST', fontsize=60)
plt.show()
```

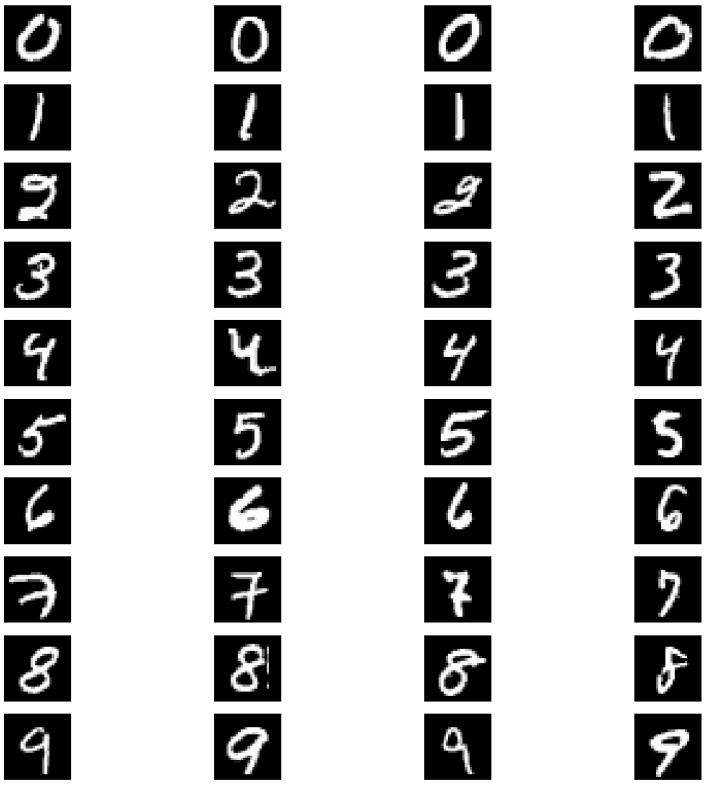


Figura 19: Muestreo de dígitos MNIST
Fuente: Elaboración Propia

Ahora mantengámoslo simple y calculemos el promedio 0. Después de eso, generalizaremos el algoritmo para que podamos trazar todos los dígitos promedio. Comenzaremos calculando el promedio del dígito 0.

```
avgZero = np.dot(digitDict[0].
avgZero = np.dot(digitDict[0].T, np.ones((nZeroImgs,1)))/nZeroImgs
plt.imshow(avgZero.reshape(28,28),cmap="gray")
plt.show()
plt.plot(avgZero)
plt.show()
```

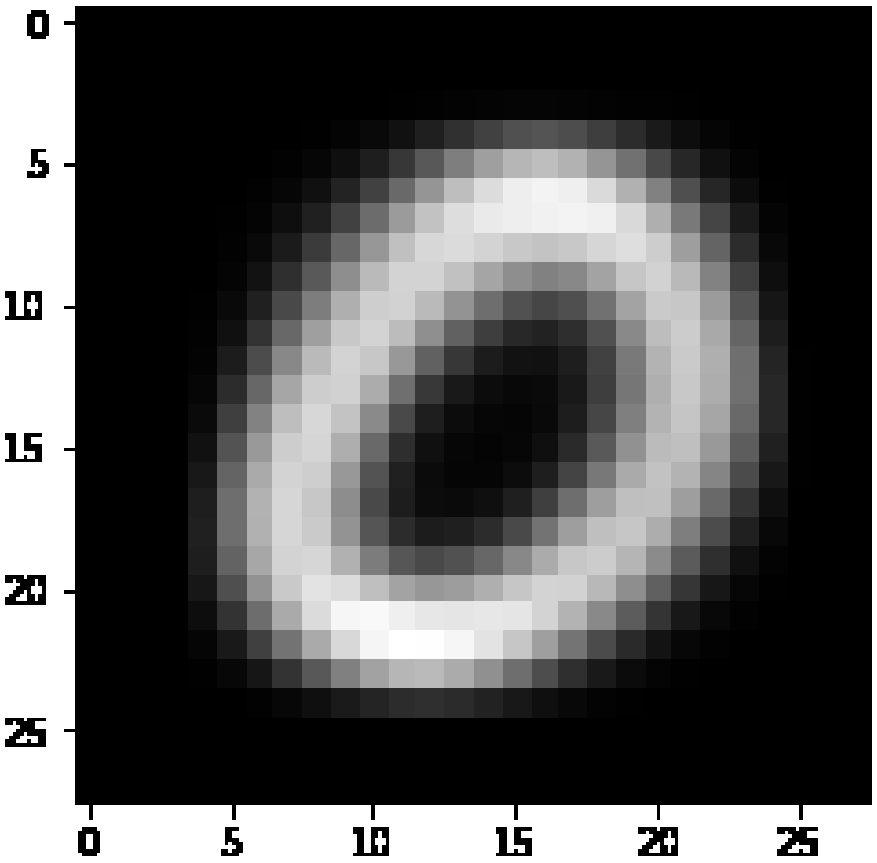


Figura 20: Promedio del dígito 0
Fuente: Elaboración Propia

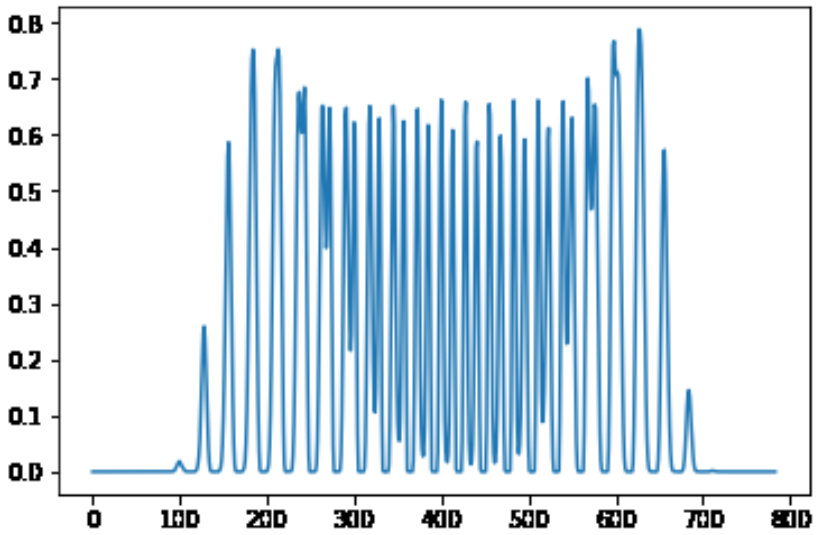


Figura 21: Trazo del promedio del dígito 0
Fuente: Elaboración Propia

Como podemos ver, hay dos máximos en las imágenes cero (las ubicaciones más activas). Ahora que sabemos cómo trazar el promedio de un solo dígito, vamos a trazar los dígitos promedio de todas las clases, junto con sus diagramas.

```
figRows,figCols = (5,4)
fig,sps = makeSubplots(figRows,figCols)
turnOffAxes(sps)

# Traza los dígitos como subconjuntos, 4 columnas y 5 filas.
subplotCounter = 0
for digitClass in range(0,10):
    nImgs = digitDict[digitClass].shape[0]
    avgImg = np.dot(digitDict[digitClass].T, np.ones((nImgs,1)))/nImgs
    # Agarra una sola columna de la matriz digitClass.
    sps[subplotCounter].imshow(avgImg.reshape(28,28),cmap='gray')
    subplotCounter += 1
    sps[subplotCounter].plot(avgImg)
```

```
    subplotCounter += 1

plt.suptitle('Dígitos promedio de MNIST', fontsize=60)
plt.show()
```

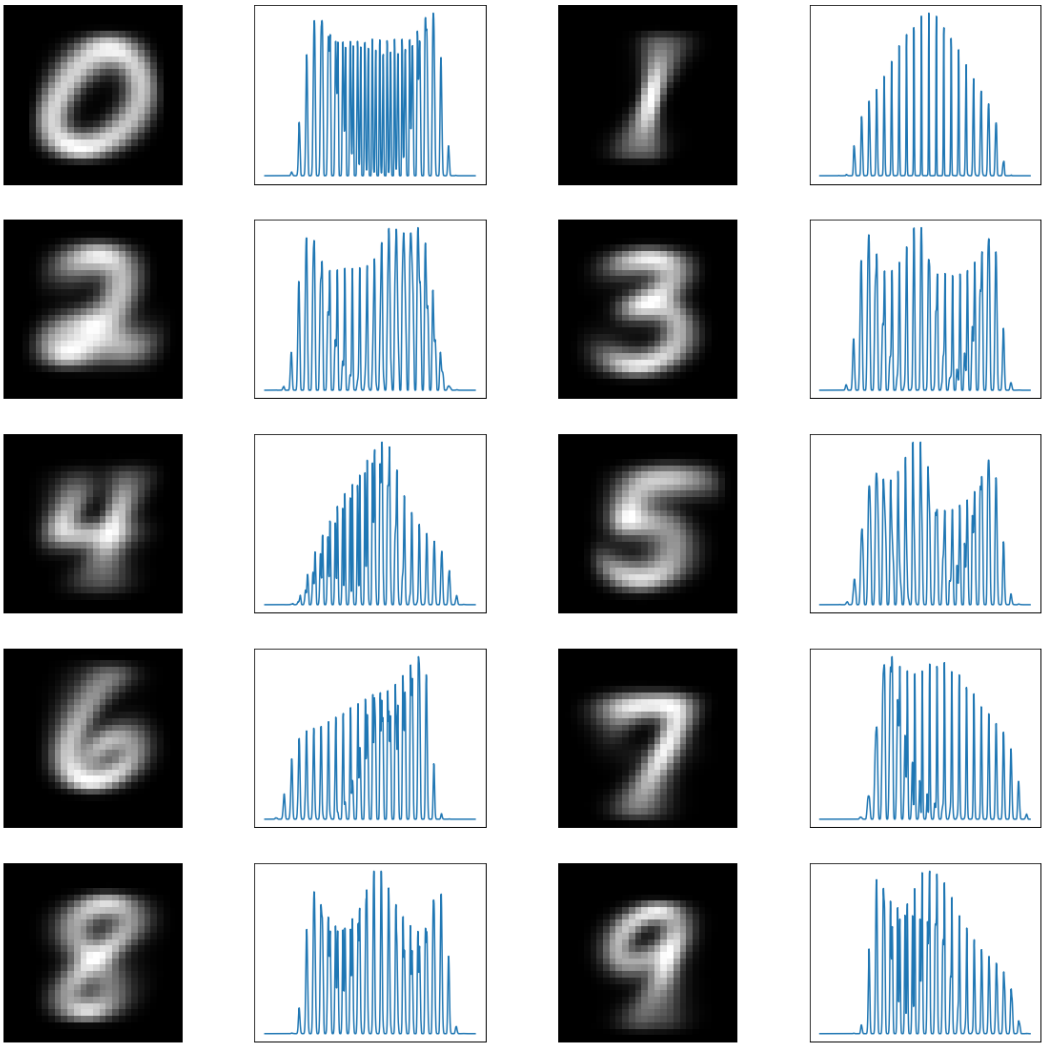


Figura 22: Trazo y valor promedio de cada clase
Fuente: Elaboración Propia

Las imágenes son el valor promedio para esos píxeles dentro de esa clase de dígitos, y los gráficos son una aproximación de la distribución de probabilidad conjunta de todos los píxeles en un dígito. Cada columna representa el valor promedio que toma el píxel. Cada píxel tiene una distribución de probabilidad propia.

4.2. PREPARACIÓN DEL ENTORNO VIRTUAL CONDA CON TENSORFLOW Y CUDA

Antes de comenzar con la experimentación con los modelos predictivos aplicados al dataset MNIST, tendremos que hacer unas configuraciones adicionales para poder utilizar aplicar el paralelismo a través del uso del GPU. Debemos instalar CUDA 9 de NVIDIA justamente para poder usar toda la capacidad de la tarjeta gráfica, también creará un entorno CONDA que en si es una virtualización donde podemos hacer instalaciones configuradas, las cuales no se afectan de una a otra. Por último, instalaremos TENSORFLOW en el entorno CONDA donde trabajaremos con el GPU.

4.2.1. HABILITANDO EN GPU NVIDIA GEFORCE GTX 980M

Para poder habilitar todo el potencial de la tarjeta gráfica, es necesario para la instalación del codo 9.0 y cudnn v7.5 en windows 10. La primera tarea a ejecutar es la instalación de estos componentes para compilar los modelos predictivos en la gpu.

En primer lugar, verificamos que la tarjeta de vídeo de NVIDIA es compatible y actualice los controladores. Visitamos la web oficial CUDA Toolkit 9.0 y descargamos la versión de acuerdo a nuestro sistema operativo y modelo de la tarjeta NVIDIA GEFORCE GTX 980M.

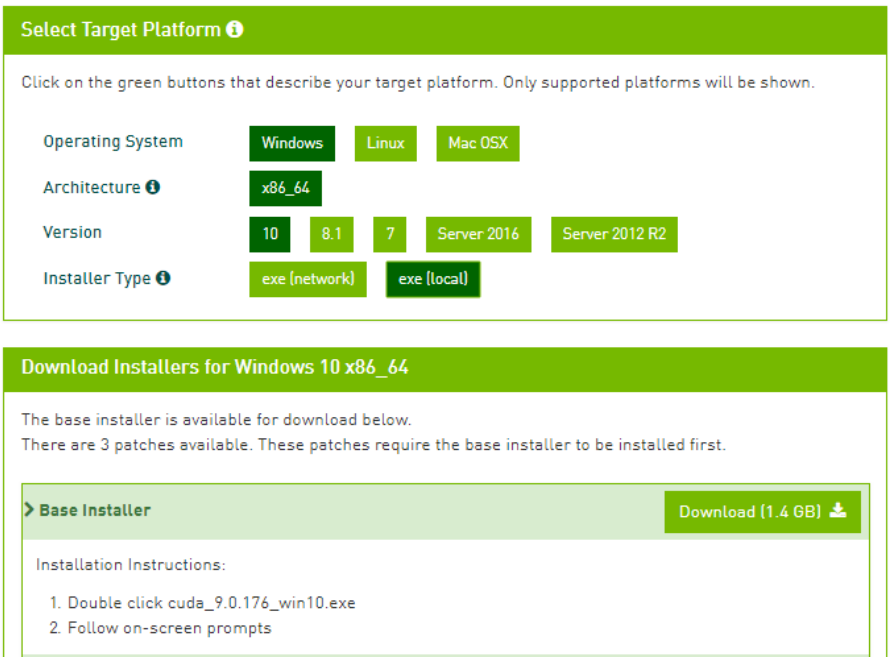


Figura 23: Fuente de descarga de CUDA 9.0
Fuente: Elaboración Propia

El proceso de instalación es simple, solo seguimos dando NEXT. Después debemos configurar los **Variables de Entorno**, donde debemos agregar la ruta de la ubicación de CUDA.

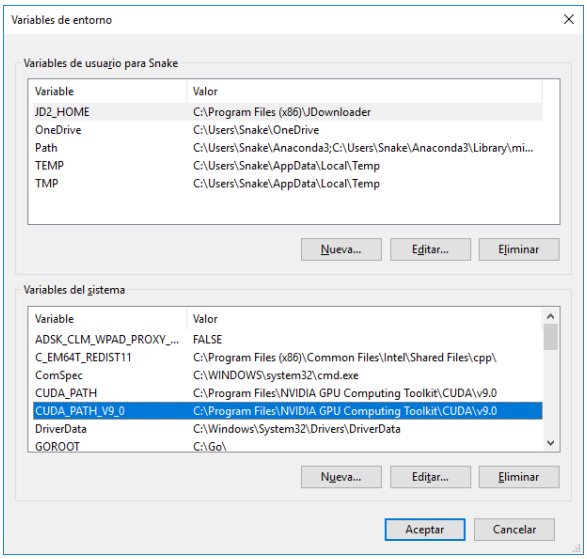


Figura 24: Configurando la ruta PATH de Windows 10
Fuente: Elaboración Propia

Con esta configuración ya podemos utilizar nuestro GPU para poder realizar cálculos en paralelo.

4.2.2. IMPLEMENTANDO UN ENTORNO CONDA

Para poder utilizar los Entornos CONDA se debe instalar **Anaconda** requiere aproximadamente 3 GB de espacio en disco, e instala más de 150 paquetes científicos (por ejemplo, paquetes para estadísticas y aprendizaje automático). También configura Anaconda Navigator, una herramienta de GUI que le ayuda a administrar los entornos y paquetes de Conda.

Una vez culminado las instalaciones, procedemos a abrir una consola donde ejecutaremos los siguientes comandos:

```
conda create --name tensorflow-gpu python=3.6
proceed ([y]/n)?  Y
actívate tensorflow-gpu
```

Así ya tenemos creado nuestro entorno CONDA llamado tensorflow-gpu que tendrá instalado Python 3.6, y procedemos a activarla para seguir con las instalaciones necesarias, así como el tensorflow.

4.2.3. INSTALANDO TENSORFLOW EN ENTORNO CONDA

Para iniciar a trabajar con Tensorflow en un entorno virtualizado, debemos estar dentro del entorno conda que hemos creado anteriormente, y procedemos a ejecutar la siguiente línea de comandos:

```
pip install --ignore-installed --upgrade tensorflow-gpupip install --ignore-installed --
upgrade tensorflow-gpu
```

Este proceso demorara en terminar, ya que tiene que descargarse varias librerías necesarias para poder trabajar con tensorflow en su versión para GPU.

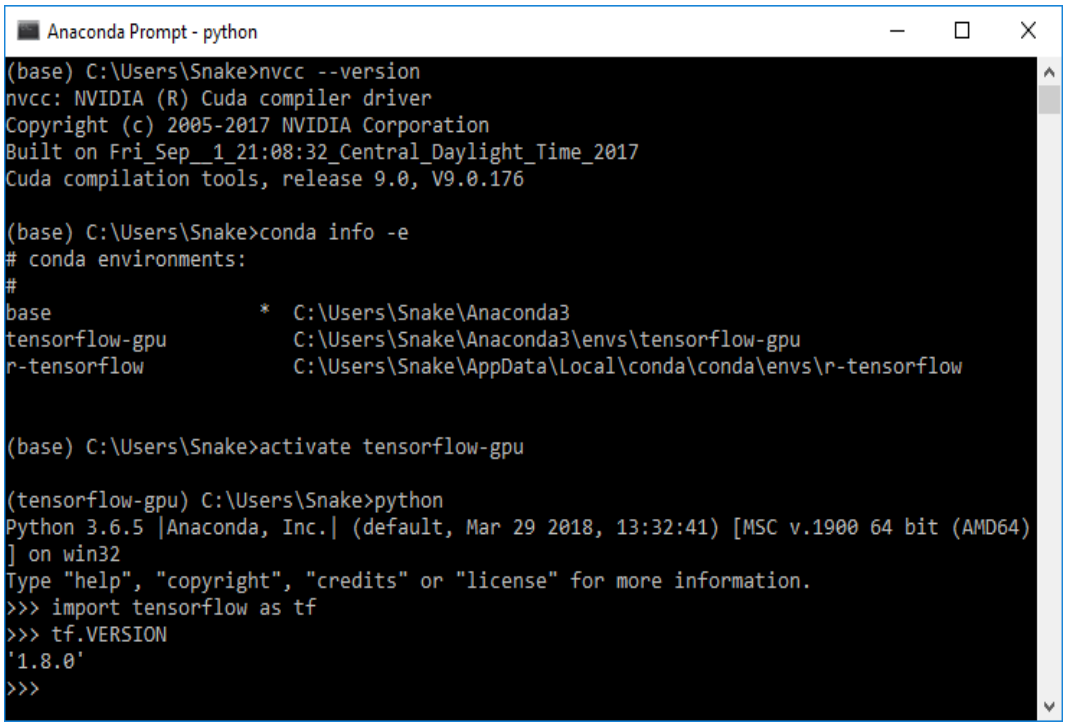


Figura 25: Comprobación de la virtualización CONDA
Fuente: Elaboración Propia

Como se observa en la Figura 25, se cuenta con CUDA 9.0 instalado para paralelizar los cálculos que se tengan que realizar con los modelos planteados. Además, ingresamos el entorno CONDA llamado **tensorflow-gpu**, donde instalaremos Tensorflow 1.8 para GPU, el entorno CONDA denominado **base** será donde se realizará la experimentación solo con CPU, pero tendrá las mismas librería y dependencias instaladas como el entorno CONDA **tensorflow-gpu**, la diferencia que la versión de tensorflow será solo para CPU.

4.3. RESULTADOS

En esta sección pasaremos a experimentar con los modelos predictivos planteados para evaluar su precisión y el tiempo de ejecución cuando se trabaja solo con el CPU y cuando se trabaja con el CPU + GPU. Se detallará como sido codificado cada modelo predictivo teniendo como valor de iteración de 20000 en cada modelo, para garantizar que estén en las mismas condiciones de experimentación.

4.3.1. Red Neuronal Artificial

Las Redes Neuronales Artificiales están dentro de la rama de Machine Learning, también es conocida como Red Neuronal superficial, a través de ella procederemos a la experimentación en los entornos CONDA configurados con CPU y GPU, para posteriormente analizar el comportamiento del algoritmo ante el paralelismo.

```
import numpy as np
from time import time
import tensorflow as tf
import matplotlib.pyplot as plt

from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("data", one_hot=True)
```

Establecemos los parámetros necesarios para homogenizar las mismas condiciones de la experimentación en todos los modelos predictivos planteados.

```
learning_rate = 0.1
num_steps = 20000
batch_size = 128
display_step = 100
```

```
# Parametros de la red neuronal
n_hidden_1 = 256 # Primera capa de neuronas
n_hidden_2 = 256 # Segunda capa de neuronas
num_input = 784 # Longitud de cada clase (cada imagen es de 28x28)
num_classes = 10 # Total de clases del MNIST (0-9 dígitos)

# Entradas de tf
X = tf.placeholder("float", [None, num_input])
Y = tf.placeholder("float", [None, num_classes])

# Definimos el peso y sesgo
weights = {
    'h1': tf.Variable(tf.random_normal([num_input, n_hidden_1])),
    'h2': tf.Variable(tf.random_normal([n_hidden_1, n_hidden_2])),
    'out': tf.Variable(tf.random_normal([n_hidden_2, num_classes]))
}
biases = {
    'b1': tf.Variable(tf.random_normal([n_hidden_1])),
    'b2': tf.Variable(tf.random_normal([n_hidden_2])),
    'out': tf.Variable(tf.random_normal([num_classes]))
}

# Creamos el Modelo
def neural_net(x):
    # Capa totalmente conectada oculta con 256 neuronas.
    layer_1 = tf.add(tf.matmul(x, weights['h1']), biases['b1'])
    # Capa totalmente conectada oculta con 256 neuronas.
    layer_2 = tf.add(tf.matmul(layer_1, weights['h2']), biases['b2'])
    # Salida de capa totalmente conectada con una neurona para cada clase.
    out_layer = tf.matmul(layer_2, weights['out']) + biases['out']
    return out_layer

# Construimos el Modelo
```

```
logits = neural_net(X)

loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
                logits=logits, labels=Y))
optimizer = tf.train.AdamOptimizer(learning_rate=learning_rate)
train_op = optimizer.minimize(loss_op)

# Evaluar el modelo
correct_pred = tf.equal(tf.argmax(logits, 1), tf.argmax(Y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))

# Inicializamos las variables
init = tf.global_variables_initializer()
```

Inicializamos el entrenamiento del algoritmo, para ello aperturaremos sesión en Tensorflow. Además, inicializaremos varias variables y listas, donde se guardarán los datos de cada iteración tanto de la precisión como del tiempo, al finalizar el entrenamiento con estos datos podremos hacer las gráficas necesarias para analizar el modelo.

```
with tf.Session() as sess:
    tiempo_inicial = time()
    tiempo_med = []
    prec_med = []
    # Ejecutamos la sesión tf.
    sess.run(init)

    for step in range(1, num_steps+1):
        batch_x, batch_y = mnist.train.next_batch(batch_size)
        # Ejecutamos el optimizador
        sess.run(train_op, feed_dict={X: batch_x, Y: batch_y})
        if step % display_step == 0 or step == 1:
            # Calcular la pérdida de lotes y la precisión
```

```
            loss, acc = sess.run([loss_op, accuracy],
                                 feed_dict={X: batch_x, Y: batch_y})
            tiempo_paso = time()
            tiempo_parcial = tiempo_paso - tiempo_inicial
            print("Step " + str(step) + ", Tiempo Parcial= " +
                "{:.4f}".format(tiempo_parcial) + ", Precisión del Entren.= " +
                "{:.4f}".format(acc))
            tiempo_med.append(tiempo_parcial)
            prec_med.append(acc)
```

```
# Calculamos la precisión final y el tiempo.
tiempo_final = time()
tiempo_ejecucion = tiempo_final - tiempo_inicial
print("Precisión:", \
    sess.run(accuracy, feed_dict={X: mnist.test.images,
                                  Y: mnist.test.labels}))
print("Tiempo: " + "{:.4f}".format(tiempo_ejecucion))
```

Al finalizar el entrenamiento procedemos a realizar la gráfica de los resultados guardados en las listas establecidas inicialmente antes del entrenamiento.

```
plt.plot(tiempo_med, prec_med)
plt.xlabel("Tiempo (s)")
plt.ylabel("Precisión")
plt.title("RN con CPU+GPU")
plt.grid()
plt.show()
```

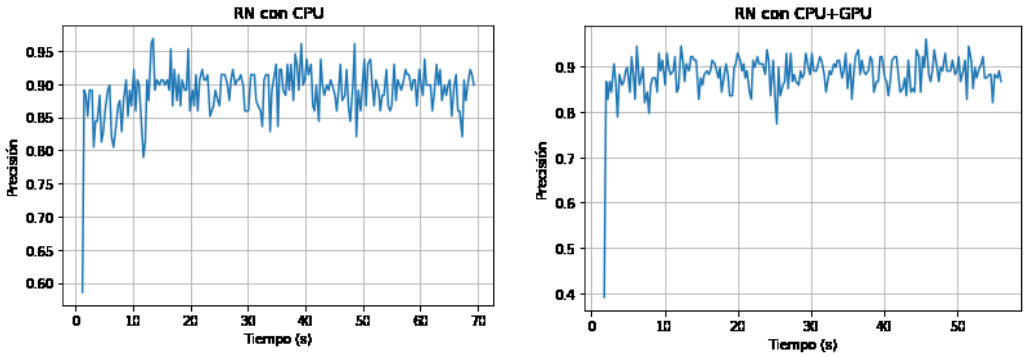


Figura 26: RN con CPU Vs. RN con CPU+GPU
Fuente: Elaboración Propia

Ahora podemos visualizar los resultados en las Figuras 26, donde resalta claramente la diferencia en el tiempo.

Tabla 4: Resultados de la RN

RN	Precisión (%)	Tiempo (seg)
CPU	84.20	69
CPU+GPU	86.66	56

En la Tabla 4, se visualiza los resultados obtenidos en la experimentación con el algoritmo de Red Neuronal, como se aprecia en la precisión se ha mantenido en un promedio ~85.43%, es decir; al aplicarle el paralelismo al algoritmo la precisión no varía, caso contrario es en el tiempo, al aplicarse el CPU+GPU al algoritmo ha reducido en un 18.84% en el tiempo.

4.3.2. Nearest Neighbor

Nearest Neighbor (Vecinos más cercanos) es un algoritmo que está dentro de la rama de Machine Learning, con el cual iniciaremos la experimentación, procederemos a detallar la codificación para la experimentación del mismo, tanto en los entornos CONDA configurados con CPU y GPU.

Comenzamos con las importaciones necesarias, además de descargar el dataset MNIST.

```
import numpy as np
from time import time
import tensorflow as tf
import matplotlib.pyplot as plt

from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("data", one_hot=True)
```

Pasaremos a limpiar los datos, definiremos los parámetros y las entradas para los tensores.

```
Xtr, Ytr = mnist.train.next_batch(55000)
Xte, Yte = mnist.test.next_batch(20000)

xtr = tf.placeholder("float", [None, 784])
xte = tf.placeholder("float", [784])
```

Calculamos la distancia del vecino más cercano, de cada lote de iteración puede ser impreciso al inicio, pero mejorar mientras transcurra más iteraciones y se tome los lotes para calcular la distancia del vecino más cercano.

```
distance = tf.reduce_sum(tf.abs(tf.add(xtr, tf.negative(xte))),
                                                    reduction_indices=1)
pred = tf.argmin(distance, 0)
accuracy = 0.
init = tf.global_variables_initializer()
```

Inicializamos el entrenamiento del algoritmo, para ello aperturaremos sesión en Tensorflow. Además, inicializaremos varias variables y listas, donde se guardarán los datos de cada iteración tanto de la precisión como del tiempo, al finalizar el entrenamiento con estos datos podremos hacer las gráficas necesarias para analizar el modelo.

```
with tf.Session() as sess:
    sess.run(init)
    tiempo_inicial = time()
    prec_med = []
    tiempo_med = []
    tiempo_parcial = 0.

    for i in range(len(Xte)):
        nn_index = sess.run(pred, feed_dict={xtr: Xtr, xte: Xte[i, :]})
        if np.argmax(Ytr[nn_index]) == np.argmax(Yte[i]):
            accuracy += 1./len(Xte)
            prec_med.append(accuracy)
            tiempo_med.append(tiempo_parcial)

        tiempo_paso = time()
        tiempo_parcial = tiempo_paso - tiempo_inicial

        if i%100 == 0 or i == 0:
            print("Iteración: " + str(i) + ", Tiempo Parcial = " +
                  "{:.4f}".format(tiempo_parcial) + ", Precisión de
                   Entrenamiento = " + "{:.4f}".format(accuracy))

    tiempo_final = time()
    tiempo_ejecucion = tiempo_final - tiempo_inicial
    print("Precisión = " + "{:.4f}".format(accuracy))
    print("Tiempo = " + "{:.4f}".format(tiempo_ejecucion))
```

Al finalizar el entrenamiento procedemos a realizar la gráfica de los resultados guardados en las listas establecidas inicialmente antes del entrenamiento.

```
plt.plot(tiempo_med, prec_med)
plt.xlabel("Tiempo (s)")
plt.ylabel("Precisión")
```

```
plt.grid()
plt.show()
```

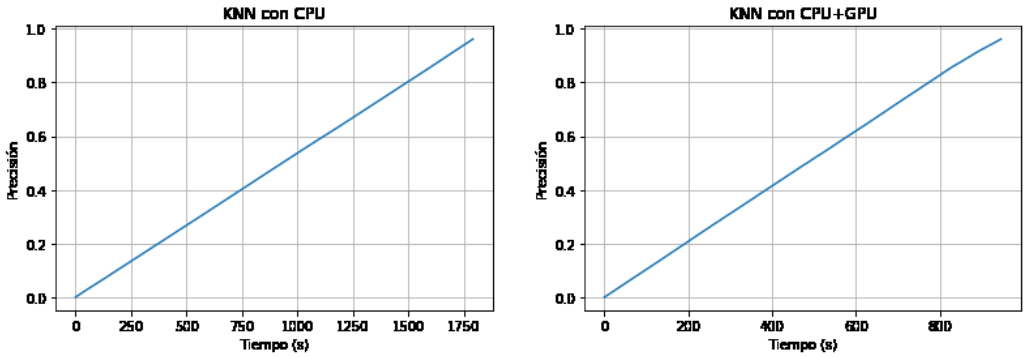


Figura 27: KNN con CPU Vs. KNN con CPU+GPU
Fuente: Elaboración Propia

Ahora podemos visualizar los resultados en las Figuras 27, donde resalta claramente la diferencia en el tiempo.

Tabla 5: Resultados de KNN

KNN	Precisión (%)	Tiempo (seg)
CPU	96.16	1792
CPU+GPU	96.16	947

En la Tabla 5, se visualiza los resultados obtenidos en la experimentación con el algoritmo Nearest Neighbor, como se aprecia en la precisión se ha mantenido en un 96.16%, es decir; al aplicarle el paralelismo al algoritmo la precisión no varía, caso contrario es en el tiempo, al aplicarse el CPU+GPU al algoritmo ha reducido significativamente a un 47.15% en el tiempo.

4.3.3. Red Neuronal Recurrente Bidireccional

Las Redes Neuronales Recurrentes Bidireccionales están dentro de la rama de Deep Learning, a través de ella procederemos a la

experimentación en los entornos CONDA configurados con CPU y GPU, para posteriormente analizar el comportamiento del algoritmo ante el paralelismo.

```
import numpy as np
from time import time
import tensorflow as tf
import matplotlib.pyplot as plt
from tensorflow.contrib import rnn
from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets("data", one_hot=True)
```

Establecemos los parámetros necesarios para homogenizar las mismas condiciones de la experimentación en todos los modelos predictivos planteados.

```
learning_rate = 0.001
training_steps = 20000

batch_size = 128
display_step = 100

# Definimos los parametros de la red neuronal
num_input = 28
timesteps = 28
num_hidden = 128
num_classes = 10

# Entradas para tf
X = tf.placeholder("float", [None, timesteps, num_input])
Y = tf.placeholder("float", [None, num_classes])
```

```
# Definimos el peso y sesgo
weights = {
    'out': tf.Variable(tf.random_normal([2*num_hidden, num_classes]))
```

```
}
biases = {
    'out': tf.Variable(tf.random_normal([num_classes]))
}

# Creamos el Modelo
def BiRNN(x, weights, biases):
    x = tf.unstack(x, timesteps, 1)
    lstm_fw_cell = rnn.BasicLSTMCell(num_hidden, forget_bias=1.0)
    lstm_bw_cell = rnn.BasicLSTMCell(num_hidden, forget_bias=1.0)

    try:
        outputs, _, _ = rnn.static_bidirectional_rnn(lstm_fw_cell,
                                                     lstm_bw_cell, x, dtype=tf.float32)
    except Exception:
        outputs = rnn.static_bidirectional_rnn(lstm_fw_cell,
                                               lstm_bw_cell, x, dtype=tf.float32)
    return tf.matmul(outputs[-1], weights['out']) + biases['out']

# Construimos el Modelo
logits = BiRNN(X, weights, biases)
prediction = tf.nn.softmax(logits)

# Defimos la perdida y la optimización.
loss_op = tf.reduce_mean(tf.nn.softmax_cross_entropy_with_logits(
                                                    logits=logits, labels=Y))
optimizer = tf.train.GradientDescentOptimizer(learning_rate=learning_rate)
train_op = optimizer.minimize(loss_op)

# Evaluamos el modelo.
correct_pred = tf.equal(tf.argmax(prediction, 1), tf.argmax(Y, 1))
accuracy = tf.reduce_mean(tf.cast(correct_pred, tf.float32))
```

```
# Inicializamos las variables
init = tf.global_variables_initializer()
```

Inicializamos el entrenamiento del algoritmo, para ello aperturaremos sesión en Tensorflow. Además, inicializaremos varias variables y listas, donde se guardarán los datos de cada iteración tanto de la precisión como del tiempo, al finalizar el entrenamiento con estos datos podremos hacer las gráficas necesarias para analizar el modelo.

```
with tf.Session() as sess:
    tiempo_inicial = time()
    tiempo_med = []
    prec_med = []
    # Ejecutamos la sesión tf.
    sess.run(init)

    for step in range(1, num_steps+1):
        batch_x, batch_y = mnist.train.next_batch(batch_size)
        batch_x = batch_x.reshape((batch_size, timesteps, num_input))
        # Ejecutamos la optimización
        sess.run(train_op, feed_dict={X: batch_x, Y: batch_y})
        if step % display_step == 0 or step == 1:
            # Calculamos los lotes de perdida y la precisión.
            loss, acc = sess.run([loss_op, accuracy],
                                 feed_dict={X: batch_x, Y: batch_y})
            tiempo_paso = time()
            tiempo_parcial = tiempo_paso - tiempo_inicial
            print("Interación " + str(step) + ", Tiempo Parcial= " +
                "{:.4f}".format(tiempo_parcial) + ", Precisión Entren.= " +
                "{:.3f}".format(acc))
```

```
# Calculamos la precisión final y el tiempo.
```

```
tiempo_final = time()
tiempo_ejecucion = tiempo_final - tiempo_inicial

test_len = 128
test_data = mnist.test.images[:test_len].reshape((-1, timesteps, num_input))
test_label = mnist.test.labels[:test_len]
print("Precisión:",
                    sess.run(accuracy, feed_dict={X: test_data, Y: test_label}))
print("Tiempo: " + "{:.4f}".format(tiempo_ejecucion))
```

Al finalizar el entrenamiento procedemos a realizar la gráfica de los resultados guardados en las listas establecidas inicialmente antes del entrenamiento.

```
plt.plot(tiempo_med, prec_med)
plt.xlabel("Tiempo (s)")
plt.ylabel("Precisión")
plt.title("RNRB")
plt.grid()
plt.show()
```

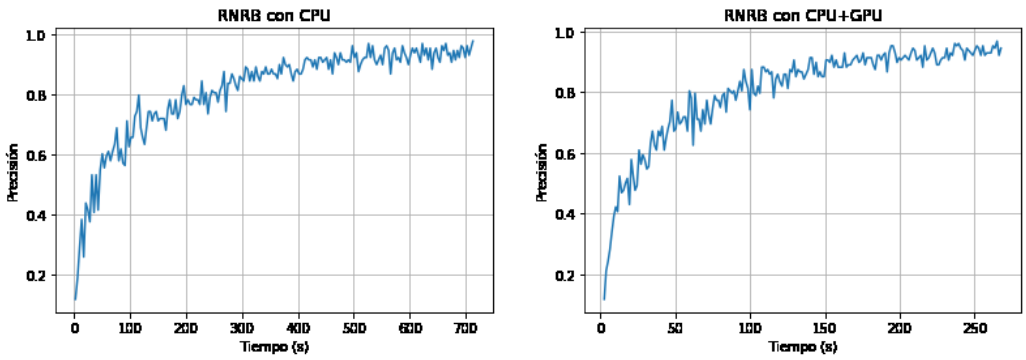


Figura 28: RNRB con CPU Vs. RNRB con CPU+GPU
Fuente: Elaboración Propia

Ahora podemos visualizar los resultados en las Figuras 28, donde resalta claramente la diferencia en el tiempo.

Tabla 6: Resultados de RNRB

RNRB	Precisión (%)	Tiempo (seg)
CPU	96.86	713
CPU+GPU	97.66	268

En la Tabla 6, se visualiza los resultados obtenidos en la experimentación con el algoritmo de una Red Neuronal Recurrente Bidireccional, como se aprecia en la precisión se ha mantenido en un promedio ~97.26%, es decir; al aplicarle el paralelismo al algoritmo la precisión no varía, caso contrario es en el tiempo, al aplicarse el CPU+GPU al algoritmo, el tiempo ha reducido significativamente en un 62.41%.

4.3.4. Red Neuronal Convolucional

Las Redes Neuronales Convolucionales están dentro de la rama de Deep Learning, a través de ella procederemos a la experimentación en los entornos CONDA configurados con CPU y GPU, para posteriormente analizar el comportamiento del algoritmo ante el paralelismo.

```
import tensorflow as tf
import matplotlib.pyplot as plt
from time import time

from tensorflow.examples.tutorials.mnist import input_data
mnist = input_data.read_data_sets('data', one_hot=True)
sess = tf.InteractiveSession()
```

Establecemos los parámetros necesarios para homogenizar las mismas condiciones de la experimentación en todos los modelos predictivos planteados.

```
# Definiendo los parametros de entrenamiento
x = tf.placeholder(tf.float32, shape=[None, 784])
y_ = tf.placeholder(tf.float32, shape=[None, 10])

W = tf.Variable(tf.zeros([784,10]))
b = tf.Variable(tf.zeros([10]))

sess.run(tf.global_variables_initializer())

y = tf.matmul(x,W) + b

cross_entropy = tf.reduce_mean(
                    tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y))
```

Antes de ingresar a la etapa de convolución del algoritmo, vamos hacer una prueba de la precisión en este punto, ya que nos encontramos ante una red neuronal simple.

```
# realizando un pretest de la neurona
train_step = tf.train.GradientDescentOptimizer(0.5).minimize(cross_entropy)

for _ in range(1000):
    batch = mnist.train.next_batch(100)
    train_step.run(feed_dict={x: batch[0], y_: batch[1]})
correct_prediction = tf.equal(tf.argmax(y,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))
print(accuracy.eval(feed_dict={x: mnist.test.images, y_: mnist.test.labels}))
0.9185
```

Como se visualiza el resultado es muy parecido al resultado de la red neuronal simple que experimentamos en primer lugar, esto nos indica que vamos por buen camino. A continuación, definiremos dos funciones para trabajar con el peso y el sesgo.

```
# Definimos las funciones del peso y el sesgo
def weight_variable(shape):
    initial = tf.truncated_normal(shape, stddev=0.1)
    return tf.Variable(initial)

def bias_variable(shape):
    initial = tf.constant(0.1, shape=shape)
    return tf.Variable(initial)
```

Crearemos funciones necesarias para la etapa de convolución.

```
def conv2d(x, W):
    return tf.nn.conv2d(x, W, strides=[1, 1, 1, 1], padding='SAME')

def max_pool_2x2(x):
    return tf.nn.max_pool(x, ksize=[1, 2, 2, 1],
                          strides=[1, 2, 2, 1], padding='SAME')
```

Configuraremos la primera y segunda capa de la red neuronal convolucionada.

```
# Configuramos la primera y segunda capa de la RNC
W_conv1 = weight_variable([5, 5, 1, 32])
b_conv1 = bias_variable([32])

x_image = tf.reshape(x, [-1,28,28,1])

h_conv1 = tf.nn.relu(conv2d(x_image, W_conv1) + b_conv1)
h_pool1 = max_pool_2x2(h_conv1)

W_conv2 = weight_variable([5, 5, 32, 64])
b_conv2 = bias_variable([64])
```

```
h_conv2 = tf.nn.relu(conv2d(h_pool1, W_conv2) + b_conv2)
h_pool2 = max_pool_2x2(h_conv2)

W_fc1 = weight_variable([7 * 7 * 64, 1024])
b_fc1 = bias_variable([1024])

h_pool2_flat = tf.reshape(h_pool2, [-1, 7*7*64])
h_fc1 = tf.nn.relu(tf.matmul(h_pool2_flat, W_fc1) + b_fc1)

keep_prob = tf.placeholder(tf.float32)
h_fc1_drop = tf.nn.dropout(h_fc1, keep_prob)

W_fc2 = weight_variable([1024, 10])
b_fc2 = bias_variable([10])

y_conv = tf.matmul(h_fc1_drop, W_fc2) + b_fc2
```

Inicializamos el entrenamiento del algoritmo, para ello aperturaremos sesión en Tensorflow. Además, inicializaremos varias variables y listas, donde se guardarán los datos de cada iteración tanto de la precisión como del tiempo, al finalizar el entrenamiento con estos datos podremos hacer las gráficas necesarias para analizar el modelo.

```
# Inicializamos el entrenamiento.
cross_entropy = tf.reduce_mean(
    tf.nn.softmax_cross_entropy_with_logits(labels=y_, logits=y_conv))

train_step = tf.train.AdamOptimizer(1e-4).minimize(cross_entropy)
correct_prediction = tf.equal(tf.argmax(y_conv,1), tf.argmax(y_,1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

tiempo_inicial = time()
```

```
tiempo_med = []
prec_med = []
sess.run(tf.global_variables_initializer())

for i in range(20000):
    batch = mnist.train.next_batch(50)
    if i%100 == 0:
        train_accuracy = accuracy.eval(feed_dict={
                                    x:batch[0], y_: batch[1], keep_prob: 1.0})
        tiempo_paso = time()
        tiempo_parcial = tiempo_paso - tiempo_inicial
        print("Interación " + str(i) + ", Tiempo Parcial= " + \
            "{:.4f}".format(tiempo_parcial) + ", Precisión Entren.= " + \
            "{:.3f}".format(train_accuracy))

        tiempo_med.append(tiempo_parcial)
        prec_med.append(train_accuracy)
    train_step.run(feed_dict={x: batch[0], y_: batch[1], keep_prob: 0.5})
```

```
# Calculamos la precisión final y el tiempo.
tiempo_final = time()
tiempo_ejecucion = tiempo_final - tiempo_inicial

print("Precision = %g"%accuracy.eval(feed_dict={
        x: mnist.test.images, y_: mnist.test.labels, keep_prob: 1.0}))
print("Tiempo: " + "{:.4f}".format(tiempo_ejecucion))
```

Al finalizar el entrenamiento procedemos a realizar la gráfica de los resultados guardados en las listas establecidas inicialmente antes del entrenamiento.

```
plt.plot(tiempo_med, prec_med)
plt.xlabel("Tiempo (s)")
```

```
plt.ylabel("Precisión")
plt.title("RNC")
plt.grid()
plt.show()
```

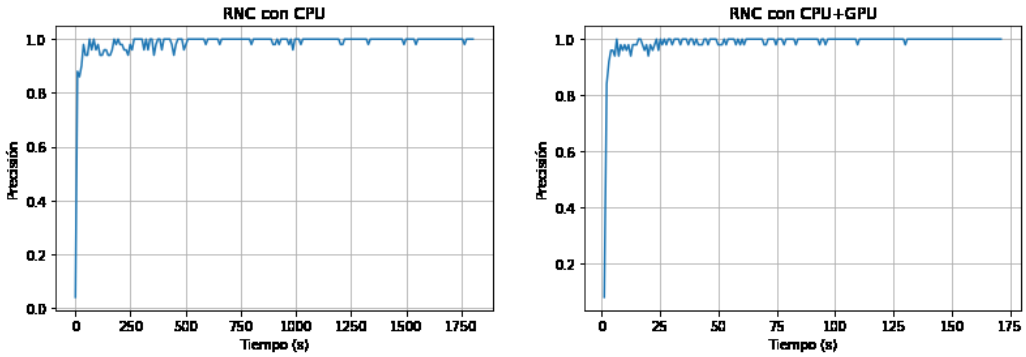


Figura 29: RNC con CPU Vs. RNC con CPU+GPU
Fuente: Elaboración Propia

Ahora podemos visualizar los resultados en las Figuras 29, donde resalta claramente la diferencia en el tiempo.

Tabla 7: Resultados de RNC

RNC	Precisión (%)	Tiempo (seg)
CPU	99.27	1819
CPU+GPU	99.13	180

En la Tabla 7, se visualiza los resultados obtenidos en la experimentación con el algoritmo de una Red Neuronal Convolucional, como se aprecia en la precisión se ha mantenido en un promedio ~99.20%, es decir; al aplicarle el paralelismo al algoritmo la precisión no varía, caso contrario es en el tiempo, al aplicarse el CPU+GPU al algoritmo, el tiempo ha reducido significativamente en un 90.10%.

Análisis de la RNC en un sistema en paralelo

La mejora en el tiempo de ejecución de la RNC es debido que los cálculos matemáticos, es decir las operaciones matriciales, hay sido ejecutados de forma paralela en el GPU, teniendo una significativa reducción del 90.10% a comparación de solo ejecutarlo con el CPU, para poder entender estos resultados analizaremos las operaciones en paralelo y el algoritmo en si para entender los resultados obtenidos.

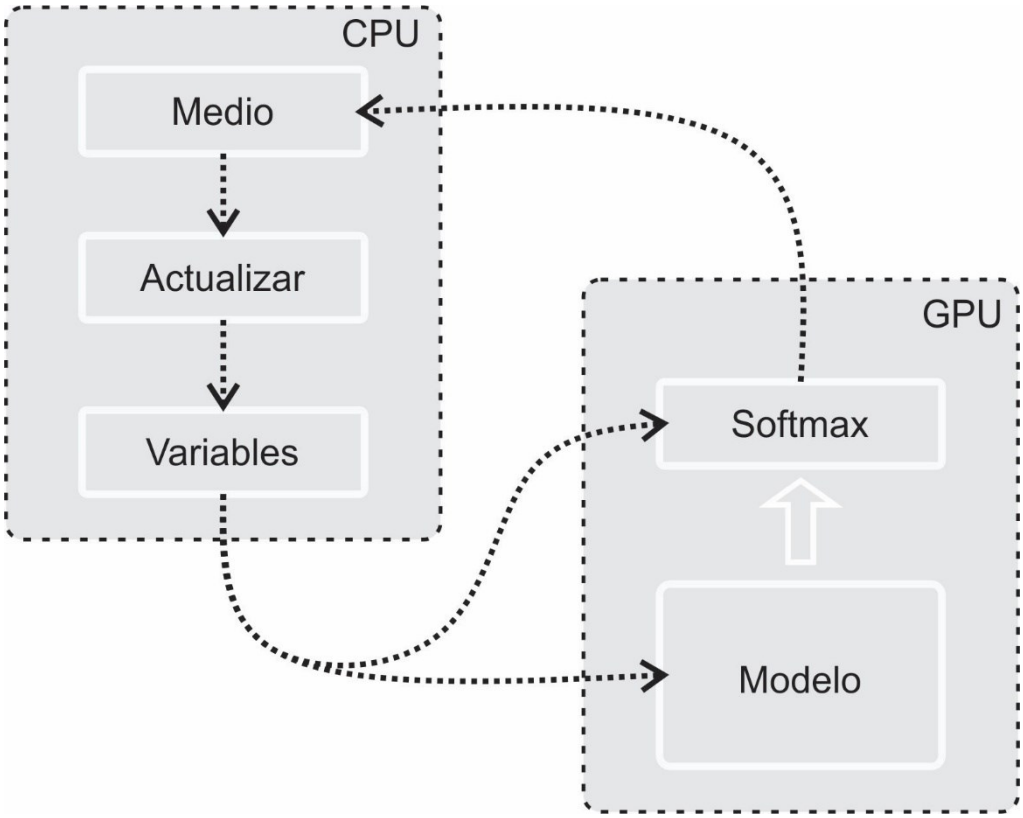


Figura 30: Distribución el cálculo en paralelo
Fuente: Elaboración Propia

Como vemos en la Figura N° 30, la repartición del trabajo entre la CPU y el GPU es dinámica, en un primer momento se definirán las variables, placeholder, el dataset para el entrenamiento, parámetros y el modelo del algoritmo, todo esto ocurrirán en el lado del CPU, para entrenar el modelo se inicia una sesión de tensorflow, el cual ya está configurado para trabajar en el lado del GPU, llevando el entrenamiento para ese lado, es decir; que

los cálculos matemáticos se realizarán en el GPU, posteriormente se distribuye los pesos probabilísticos en cada clase que son 10 a través de la función SOFTMAX, a la clase con mayor peso probabilístico será la clase predicha por el modelo.

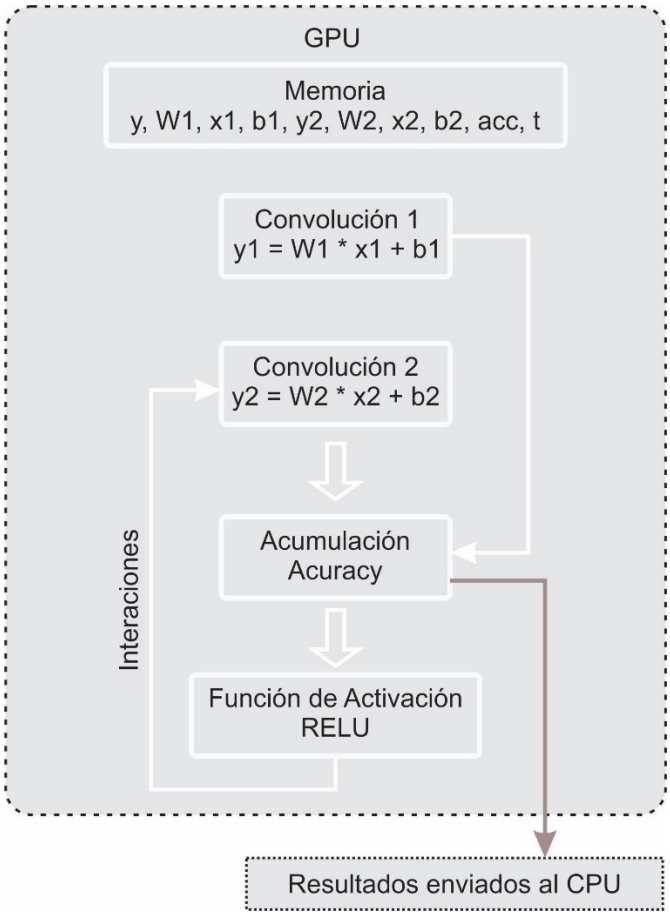


Figura 31: Procesamiento del cálculo en GPU
Fuente: Elaboración Propia

Un proceso más detallado de lo que ocurre en la GPU, podemos ver en la Figura N° 31, donde en memoria se encuentran cargada las variables inicializadas, posteriormente se inicia la primera convolución donde los resultados de ella son guardados en el acumulador ***acuracy***, pasando posteriormente a la función de activación ReLU, en este punto pasamos a la segunda convolución, los cálculos son acumulados en el acuracy y por último los resultados son enviados al entorno del CPU para su tratamiento.

Análisis del algoritmo de la RNC:

En la Figura N° 32, se visualiza un resumen de los procesos establecidos para una RNC, donde analizaremos cada uno de los procesos.

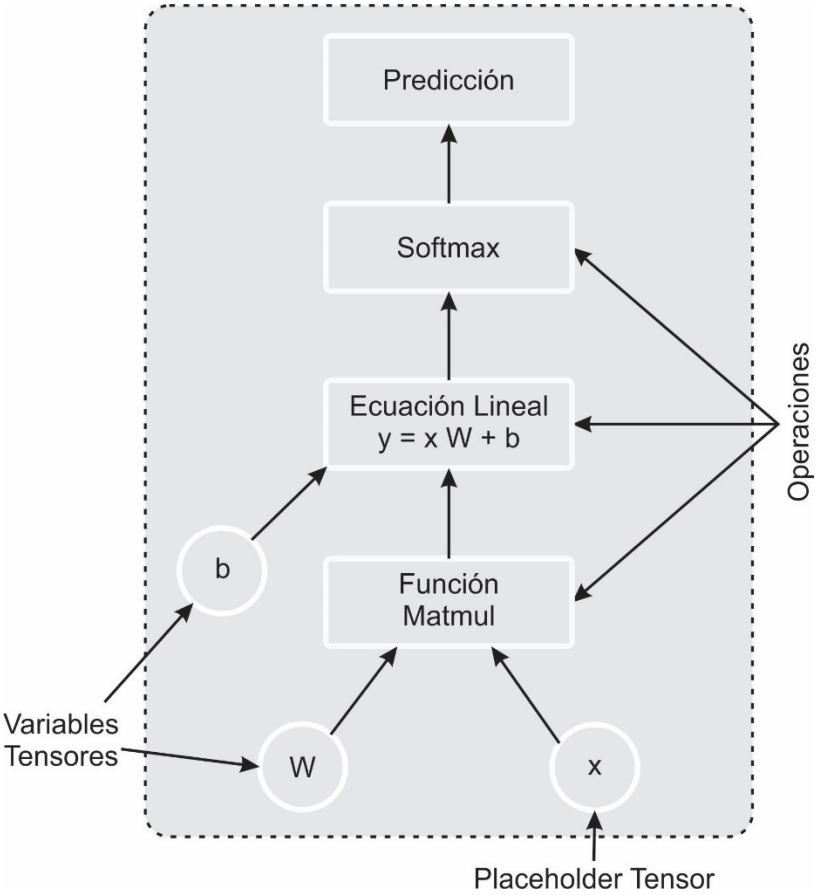


Figura 32: Diagrama Proceso de una RNC
Fuente: Elaboración Propia

Los valores de las variables tensores y el placeholder tensor serán parte de una expresión lineal para recudir su operacionalidad al momento de realizar los cálculos matemáticos y para su mejor entendimiento, es así que:

$$y = x * W + b \quad (1)$$

Como se ve en la ecuación 1, es una expresión lineal de los valores de entrada los cuales son matrices de diferentes dimensiones como ***x[99,783]***, el peso ***W[783,9]*** y el ***bias b[9]***, una idea más clara podemos verla en la Figura N° 33.

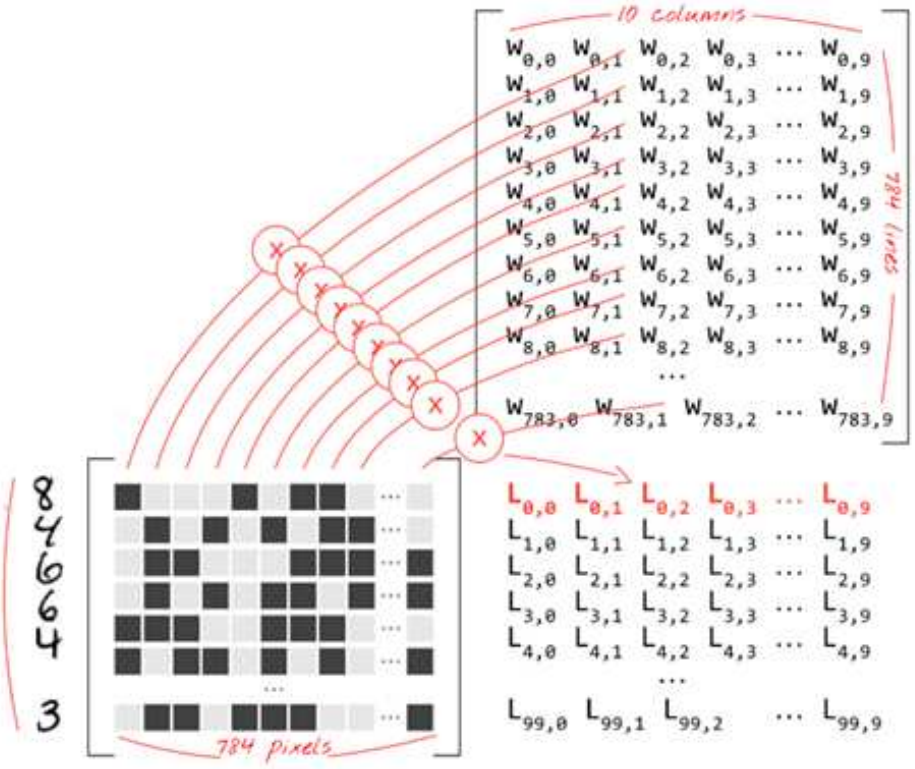


Figura 33: Diagrama Proceso de una RNC
Fuente: Elaboración Propia

La Figura N°33, denota claramente se realiza la multiplicación de las matrices $x * W$ dando como resultado ***L[99,9]***, sumándole a este el ***bias*** b, quedando la siguiente ecuación.

$$y = \sum \left[\begin{pmatrix} x_{0,0} & \cdots & & & \cdots & & \\ \vdots & \ddots & \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{99,0} & \cdots & & & \cdots & & \end{pmatrix} \right] \quad (2)$$

Al haber definido claramente la ecuación lineal (2) con la que se trabajara, donde sus elementos están compuestos por matrices establecidas, procederemos a definir las convoluciones necesarias para la presente investigación.

Método de Convolucional de 2 Capas

La capa convolucional es la primera capa donde extraemos características de las imágenes en nuestros conjuntos de datos. Debido a que los píxeles solo están relacionados con los píxeles adyacentes y cercanos, la convolución nos permite preservar la relación entre las diferentes partes de una imagen. La convolución consiste básicamente en filtrar la imagen con un filtro de píxeles más pequeño para disminuir el tamaño de la imagen sin perder la relación entre los píxeles. Cuando aplicamos la convolución a una imagen de 28x28 usando un filtro de 5x5 con paso de 24x24, como se visualiza en la Figura N° 34.

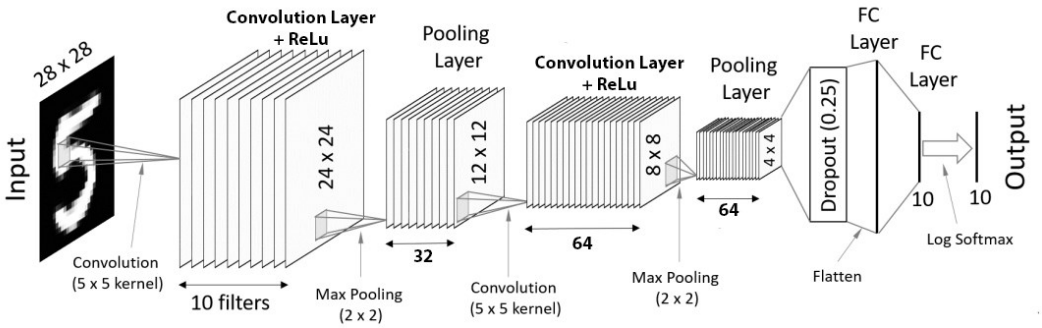


Figura 34: Proceso detalla de una RNC
Fuente: Elaboración Propia

Capa de Convolución

La capa convolucional es la primera capa en la que se extrae características de las imágenes en el conjunto de datos. Debido a que los píxeles solo están relacionados con los píxeles adyacentes y cercanos, la convolución nos permite preservar la relación entre las diferentes partes de una imagen. La convolución consiste básicamente en filtrar la imagen con un filtro de píxeles más pequeño para disminuir el tamaño de la imagen sin perder la relación entre los píxeles. Cuando aplicamos la convolución a una imagen de 28x28 usando un filtro de 5x5 con paso de 1x1 (desplazamiento de 1 píxel en cada paso) como se visualiza en la Figura N°

35. Terminaremos teniendo una salida de 24x24, de este modo se ha disminuido la complejidad.

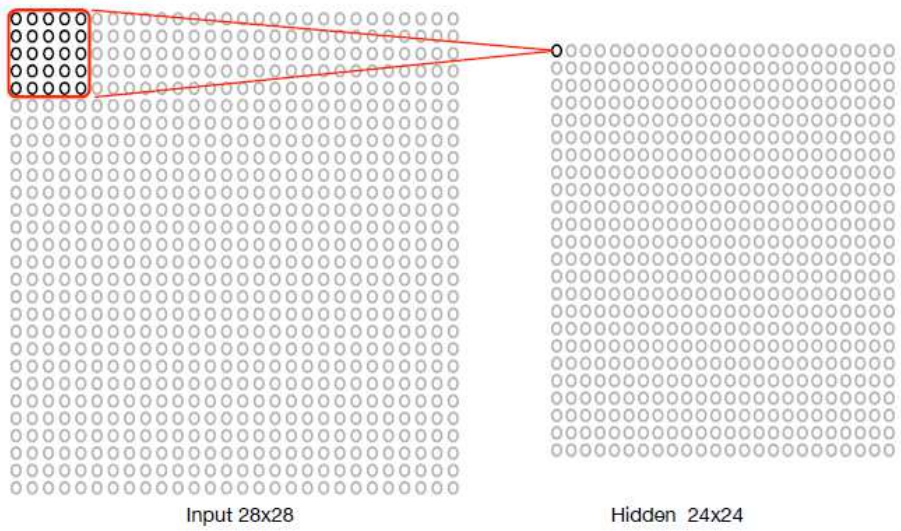


Figura 35: Filtrando la capa input con una capa 5x5
Fuente: Elaboración Propia

Función de Activación ReLU (Rectified Linear Activation Function)

La función ReLU (Figura N° 36) tiene un gradiente constante y, por lo tanto, no tiene un problema de degradación que desaparece. Esta es la forma más simple y eficiente de la función de activación que funciona en un 95% de los casos, y para la presente investigación nos ha dado mejores resultados.

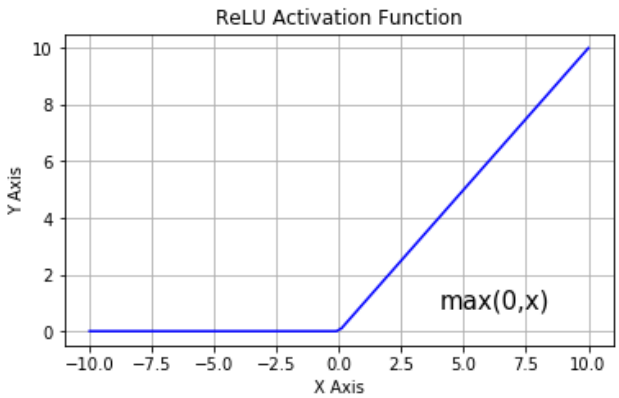


Figura 36: Función ReLU
Fuente: Elaboración Propia

Capa de Agrupación (Pooling Layer)

Al construir la Red Neuronal Convolucional, es común insertar capas de agrupación (Pooling Layer) después de cada capa de convolución para reducir el tamaño espacial de la representación para disminuir el número de parámetros, lo que reduce la complejidad computacional. Además, la combinación de capas también ayuda con el problema de sobreajuste. Básicamente, seleccionamos un tamaño de agrupación para reducir la cantidad de parámetros seleccionando los valores máximos, promedio o suma dentro de estos píxeles. Max Pooling una de las técnicas de agrupación más comunes, se puede demostrar en la Figura N° 37.

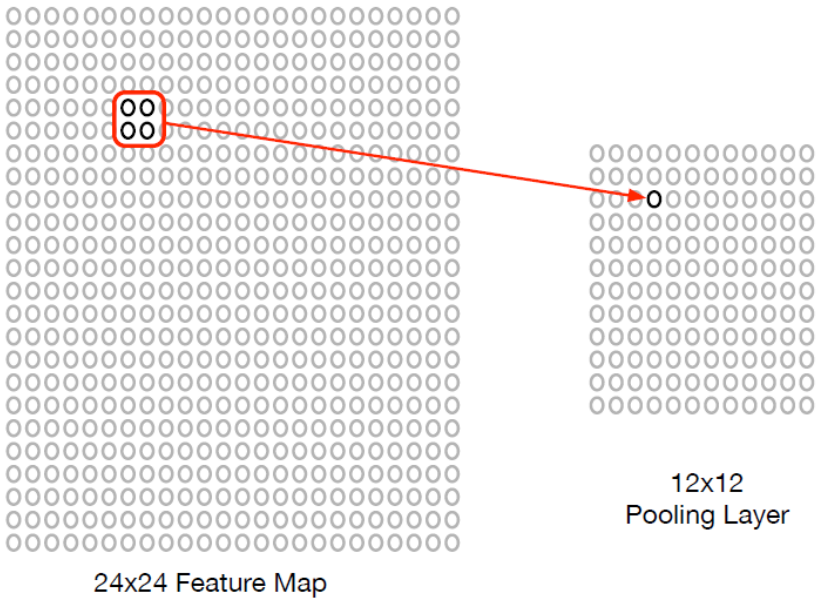


Figura 37: Max Polling 4x4
Fuente: Elaboración Propia

Capas Totalmente Conectadas (Full Connected Layers)

Una red totalmente conectada es nuestra RNC, donde cada parámetro está vinculado entre sí para determinar la verdadera relación y el efecto de cada parámetro en las etiquetas. Debido a que nuestra complejidad espacio-

temporal se reduce enormemente gracias a las capas de convolución y agrupación, podemos construir una red totalmente conectada al final para clasificar nuestras imágenes.

Softmax

Cada imagen de MNIST representa un número del 0 al 9. Para obtener la probabilidad de una imagen dada represente cada número. Por ejemplo, nuestro modelo puede especular que la probabilidad de que una imagen que contenga 9 y que represente el número 9 es del 80%, pero la probabilidad de juzgarla como 8 es del 5% (porque tanto 8 como 9 tienen el círculo pequeño de la mitad superior), y entonces dale valores más pequeños que representen otros números.

El modelo softmax (3) se puede usar para asignar probabilidades a diferentes objetos. Incluso después de entrenar modelos más elaborados, el paso final también debe usar softmax para asignar probabilidades.

$$\begin{bmatrix} y_0 \\ \vdots \\ y_9 \end{bmatrix} \quad \begin{bmatrix} \begin{pmatrix} W_{0,0} & \cdots \end{pmatrix} & \begin{pmatrix} x_0 \end{pmatrix} & \begin{pmatrix} b_0 \end{pmatrix} \\ \vdots \quad \ddots \quad \vdots & \vdots & \vdots \\ \begin{pmatrix} W_{9,0} & \cdots \end{pmatrix} & \begin{pmatrix} x_9 \end{pmatrix} & \begin{pmatrix} b_9 \end{pmatrix} \end{bmatrix} \tag{3}$$

Entrenamiento la RNC

Para entrenar la RNC primero debemos definir un indicador para evaluar si el modelo es bueno. De hecho, en el aprendizaje automático, generalmente definimos indicadores para indicar que un modelo es malo, este indicador se denomina costo o pérdida, y luego intentamos minimizar este indicador. Sin embargo, estos dos métodos son los mismos.

$$H_{y'}(y) = -\sum_i y'_i \log(y_i) \tag{4}$$

Se requiere que TensorFlow use un algoritmo de descenso de gradiente para minimizar la entropía cruzada (4). El algoritmo de descenso de gradiente es un proceso de aprendizaje simple, TensorFlow simplemente mueve cada variable poco a poco en una dirección que mantiene los costos bajos.

Lo que realmente hace TensorFlow es que agrega una serie de nuevos cálculos en la parte posterior de la gráfica que describe sus cálculos para la propagación hacia atrás y el gradiente de pendiente. Entonces, te devuelve una sola operación. Al ejecutar esta operación, utiliza el algoritmo de descenso de gradiente para entrenar el modelo, ajustar sus variables y reducir continuamente los costos.

Las iteraciones serán de 20000, en cada paso de la iteración, se toma al azar 100 lotes de puntos de datos en los datos de entrenamiento. Luego usamos estos puntos de datos como parámetros para reemplazar los marcadores de posición anteriores ***train_step***.

El uso de una pequeña cantidad de datos aleatorios para el entrenamiento se denomina entrenamiento estocástico, más específicamente, entrenamiento de descenso de gradiente aleatorio. Idealmente, queremos usar cada uno de nuestros datos para realizar cada paso de la capacitación porque nos brinda mejores resultados de capacitación, pero obviamente, esto requiere una gran cantidad de sobrecarga de cómputo. Por lo tanto, podemos usar diferentes subconjuntos de datos para cada entrenamiento. Esto no solo reducirá la sobrecarga computacional, sino que también maximizará las características generales del conjunto de datos.

Evaluamos la RNC

Primero verificamos qué etiquetas son correctas. ***tf.argmax*** es una función muy útil, puede dar un objeto tensor en una cierta dimensión del valor del índice de sus datos. Dado que la etiqueta es un vector que consta de 0, el valor máximo de la posición del índice se ubica en una etiqueta de categoría, como ***tf.argmax(y_conv,1)*** el retorno de la predicción del modelo para cualquier entrada ***X*** al valor de la etiqueta, y ***tf.argmax(y_,1)*** el representante de la etiqueta correcta, puede usar ***tf.reduce_mean*** si la predicción es una coincidencia de etiqueta verdadera (la misma que la posición del índice indica una coincidencia).

El valor del resultado final debe ser de aproximadamente el ~99.18%, siendo un resultado aceptable debido a que utilizamos una red neural de aprendizaje profundo. El mejor modelo puede incluso obtener una precisión de más del 99.7% como consta en el portal de Data Science Kaggle

4.4. COMPROBACIÓN DE LA HIPOTESIS

En la Tabla N° 8, se presenta los resultados finales de los cuatro algoritmos planteados para la presente experimentación.

Tabla 8: Resultados globales para cada modelo predictivo

MODELOS	CPU		CPU+GPU	
	Precisión (%)	Tiempo (seg)	Precisión (%)	Tiempo (seg)
CN	84.20	69	86.66	56
KNN	96.16	1792	96.16	947
RNCB	96.86	713	97.66	268
RNC	99.27	1819	99.13	180

Para lograr un mejor entendimiento de los resultados los datos serán transformados a sus expresiones logarítmicas para datos distantes como

lo recomienda (Montgomery, Peck, & Vining, 2013), quedando la tabla de la siguiente manera:

Tabla 9: Resultados globales expresados logarítmicamente

MODELOS	CPU		CPU+GPU	
	Precisión (%)	Tiempo (seg)	Precisión (%)	Tiempo (seg)
CN	1.92531209	1.83884909	1.93781868	1.74818803
KNN	1.98299445	3.25333801	1.98299445	2.97634998
RNCB	1.98614446	2.85308953	1.98971672	2.42813479
RNC	1.99681802	3.2598327	1.99620511	2.25527251

Prueba de Hipótesis Especifica 1:

H_0: Los modelos predictivos basados en Visión Computacional no mejoran la precisión al ser aplicados en la computación paralela.

H_1: Los modelos predictivos basados en Visión Computacional mejoran la precisión al ser aplicados en la computación paralela.

De la Tabla 10, extraemos los datos de la precisión para hacer al análisis correspondiente para la aceptación o rechazo de la hipótesis. En la Hipótesis Especifica 1 no se utilizará los equivalentes logarítmicos de los resultados.

Tabla 10: Resultados de la precisión para cada modelo predictivo

MODELOS	CPU	CPU+GPU	Media (%)
	Precisión (%)	Precisión (%)	
CN	84.20	86.66	85.43
KNN	96.16	96.16	96.16
RNCB	96.86	97.66	97.26
RNC	99.27	99.13	99.2

Con un nivel de significancia del 0.05, podemos decir que los valores obtenidos del modelo RN se encuentra en el rango de 85.43% ±2.5, con el modelo KNN sus valores se encuentran en el rango de 96.16%±2.5, con el modelo RNCB sus valores se encuentran en el rango 97.26%±2.5 y con el modelo RNC sus valores se encuentran en el rango 99.2%±2.5. Por los resultados obtenidos podemos manifestar que el paralelismo no afecta en la precisión cuando se aplica a CPU y a CPU+GPU.

Entonces, se acepta la hipótesis nula “Los modelos predictivos basados en Visión Computacional no mejoran la precisión al ser aplicados en la computación paralela”.

Prueba de Hipótesis Especifica 2:

H_0: Los modelos predictivos basados en Visión Computacional no mejoran el tiempo de procesamiento al ser aplicados en la computación paralela.

H_1: Los modelos predictivos basados en Visión Computacional mejoran el tiempo de procesamiento al ser aplicados en la computación paralela.

De la Tabla 11, extraemos los datos del tiempo de ejecución para hacer al análisis correspondiente para la aceptación o rechazo de la hipótesis.

Tabla 11: Resultados del tiempo para cada modelo predictivo

MODELOS	CPU	CPU+GPU	Media
	Tiempo (seg)	Tiempo (seg)	(seg)
CN	69	56	62.5
KNN	1792	947	1369.5
RNCB	713	268	490.5
RNC	1819	180	999.5

Con un nivel de significancia del 0.05, podemos decir que los valores obtenidos del modelo RN no se encuentra en el rango de 62.5seg ±2.5%, con el modelo KNN sus valores no se encuentran en el rango de 1369.5seg±2.5%, con el modelo RNCB sus valores no se encuentran en el rango 490.5seg±2.5% y con el modelo RNC sus valores no se encuentran en el rango 999.5seg±2.5%. Por los resultados obtenidos podemos manifestar que el paralelismo afecta significativamente en el tiempo de procesamiento cuando se aplica a CPU y a CPU+GPU.

Para realizar un análisis más exhaustivo utilizaremos los datos en su expresión logarítmica.

Tabla 12: Resultados del tiempo expresado logarítmicamente

MODELOS	CPU	CPU+GPU
	Tiempo (seg)	Tiempo (seg)
CN	1.83884909	1.74818803
KNN	3.25333801	2.97634998
RNCB	2.85308953	2.42813479
RNC	3.2598327	2.25527251

Realizando la evaluación por bloques expresados en la Tabla 12 y Figura 38; podemos decir que, si bien hay una mejorar en el tiempo de procesamiento para la CN; no obstante, no se encuentra en los rangos de mejora a comparación de los demás modelos predictivos.

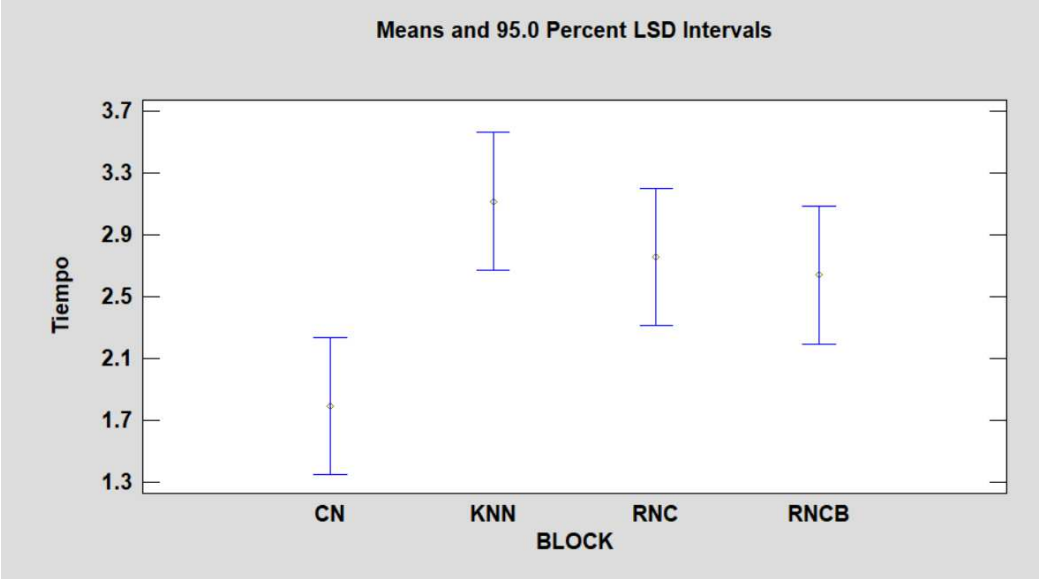


Figura 38: Medias comparativas entre los modelos predictivos
Fuente: Elaboración Propia

Entonces, se rechaza la hipótesis nula solo para los modelos predictivos KNN, RNC y RNCB; y por lo tanto se acepta la hipótesis alterna “Los modelos predictivos basados en Visión Computacional mejoran el tiempo de procesamiento al ser aplicados en la computación paralela”.

V. CONCLUSIONES

Los modelos predictivos basados en Visión Computacional mejoran al ser aplicados en la computación paralela solo en el tiempo de procesamiento, puesto que los modelos predictivos son los mismo tan solo cambia sus recursos computacionales.

Los modelos predictivos basados en Visión Computacional no mejoran la precisión al ser aplicados en la computación paralela, ya que los modelos predictivos no cambian, siguen siendo los mismos cuando se trabaja con CPU y cuando se trabaja con CPU+GPU.

Los modelos predictivos basados en Visión Computacional (KNN, RNCB y RNC) mejoran el tiempo de procesamiento al ser aplicados en la computación paralela, debido a que solo se incrementa los recursos computacionales al sumar el GPU al CPU, parametrizando los cálculos matemáticos.

VI. RECOMENDACIONES

Recomendar para futuras investigaciones por qué en la CN no se logró el rango de mejora a comparación de los otros modelos predictivos, lograr entender por qué el paralelismo no tuvo un grado de mejora significativo para la CN.

Profundizar en futuras investigaciones sobre las RNC que lograron una de las mejores puntuaciones en la precisión y logro menos tiempo de ejecución, de este modo la eficiencia demostrada por el algoritmo destaco de forma sobresaliente sobre los demás algoritmos planteado en la experimentación.

Recomendar futuras investigaciones a base de la presente investigación, pero empleando otros dataset para poder determinar si los algoritmos se comportan del mismo modo en diferentes casos de estudio.

VII. REFERENCIAS BIBLIOGRÁFICAS

ARENAS, K. J. M. A. (2016). *Diseño de un sistema de clasificación de señales de tránsito vehicular utilizando redes neuronales convolucionales*.

Ascarza Mendoza, F. J. (2018). *Segmentación Automática de Textos, mediante Redes Neuronales Convolucionales en Imágenes Documentos Históricos*. 67. Retrieved from http://tesis.pucp.edu.pe/repositorio/handle/123456789/13050

Bansal, D., Chhikara, R., Khanna, K., & Gupta, P. (2018). Comparative Analysis of Various Machine Learning Algorithms for Detecting Dementia. *Procedia Computer Science*, *132*, 1497–1502. https://doi.org/10.1016/j.procs.2018.05.102

Branson, S., Wegner, J. D., Hall, D., Lang, N., Schindler, K., & Perona, P. (2018). From Google Maps to a fine-grained catalog of street trees. *ISPRS Journal of Photogrammetry and Remote Sensing*, *135*, 13–30. https://doi.org/10.1016/j.isprsjprs.2017.11.008

Deshpande, M. (2018). *Deep Learning for Human Beings*.

Giancarlo Zaccone. (2017). *Getting Started With TensorFlow | TensorFlow*. Retrieved from https://www.tensorflow.org/get_started/get_started

Gu, J., Wang, Z., Kuen, J., Ma, L., Shahroudy, A., Shuai, B., … Chen, T. (2017). Recent advances in convolutional neural networks. *Pattern Recognition*. https://doi.org/10.1016/j.patcog.2017.10.013

Haykin, S. (2008). Neural Networks and Learning Machines. In *Pearson Prentice Hall New Jersey USA 936 pLinks*. https://doi.org/978-0131471399

Hernández, Á. B., Perez, M. S., Gupta, S., & Muntés-Mulero, V. (2017). Using machine learning to optimize parallelism in big data applications. *Future Generation Computer Systems*. https://doi.org/10.1016/j.future.2017.07.003

Hernandez Sampieri, R., Fernandez Collado, C., & Baptista Lucio, M. del P. (2010). Metodología de la investigación. In *Metodología de la investigación*. https://doi.org/- ISBN 978-92-75-32913-9

Imandoust, S. B., & Bolandraftar, M. (2013). Application of K-Nearest Neighbor (KNN) Approach for Predicting Economic Events: Theoretical Background. *International Journal of Engineering Research and Applications*, *3*(5), 605–610.

Impedovo, S., Mangini, F. M., & Barbuzzi, D. (2014). A novel prototype generation technique for handwriting digit recognition. *Pattern Recognition*, *47*(3), 1002–1010. https://doi.org/10.1016/j.patcog.2013.04.016

John von Neumann. (2002). *Publication Series of the John von Neumann Institute for Computing (NIC) NIC Series Volume 3* (Vol. 3). https://doi.org/10.1371/journal.pone.0051754

Kasap, B., & van Opstal, A. J. (2018). Dynamic parallelism for synaptic updating in GPU-accelerated spiking neural network simulations. *Neurocomputing*, *302*, 55–65. https://doi.org/10.1016/j.neucom.2018.04.007

Kim, S. B., Kim, D. Y., & Wise, K. (2014). The effect of searching and surfing on recognition of destination images on Facebook pages. *Computers in Human Behavior*, *30*, 813–823. https://doi.org/10.1016/j.chb.2013.07.010

Kokkinos, Y., & Margaritis, K. G. (2018). Managing the computational cost of model selection and cross-validation in extreme learning machines via Cholesky, SVD, QR and eigen decompositions. *Neurocomputing*, *295*, 29–45. https://doi.org/10.1016/j.neucom.2018.01.005

Kussul, E., & Baidyk, T. (2004). Improved method of handwritten digit

recognition tested on MNIST database. *Image and Vision Computing*, *22*(12 SPEC. ISS.), 971–981. https://doi.org/10.1016/j.imavis.2004.03.008

Lin, G., & Shen, W. (2018). ScienceDirect ScienceDirect Research on convolutional neural network based on improved Relu piecewise activation function Research on convolutional neural network based on improved Relu piecewise activation function. *Procedia Computer Science*, *131*, 977–984. https://doi.org/10.1016/j.procs.2018.04.239

Luis Aguilar Ibáñez. (2019). *ANÁLISIS COMPARATIVO EN LA IMPLEMENTACIÓN DE LA RED NEURONAL BACKPROPAGATION USANDO EL MÉTODO DE COMPONENTES PRINCIPALES Y EL MÉTODO CLÁSICO*.

Mohri, M. (2012). Foundations of Machine Learning - Book. In *The MIT Press*. https://doi.org/ISBN 978-0-262-01825-8

Montgomery, D. C., Peck, E. A., & Vining, G. G. (2013). *Introduction to linear regression analysis: Solutions manual to accompany*.

Qiao, J., Wang, G., Li, W., & Chen, M. (2018). An adaptive deep Q-learning strategy for handwritten digit recognition. *Neural Networks*. https://doi.org/10.1016/j.neunet.2018.02.010

Romero Ugalde, H. M., Carmona, J. C., Reyes-Reyes, J., Alvarado, V. M., & Mantilla, J. (2015). Computational cost improvement of neural network models in black box nonlinear system identification. *Neurocomputing*, *166*, 96–108. https://doi.org/10.1016/j.neucom.2015.04.022

Ruiz, C. A., & Matich, D. J. (2001). *Redes neuronales - Conceptos Básicos y Aplicaciones.*

Samer, C. H., Rishi, K., & Rowen. (2015). Image Recognition Using Convolutional Neural Networks. *Cadence Whitepaper*, 1–12.

Shustanov, A., & Yakimov, P. (2017). CNN Design for Real-Time Traffic Sign Recognition. *Procedia Engineering*, *201*, 718–725. https://doi.org/10.1016/j.proeng.2017.09.594

Szeliski, R. (2010). *Computer Vision: Algorithms and Applications*. https://doi.org/10.1007/978-1-84882-935-0

Universidad de las Ciencias Informáticas (Havana, C., Martínez, S. S., Abed, A. R., Ábalo, R. G., & Lorenzo, M. M. G. (2006). Revista cubana de ciencias informáticas. *Revista Cubana de Ciencias Informáticas*, *1*(4), 48–57.

Zinchenko, Y. (2017). *Big Data and Google Cloud Platform*. (April).

VIII. ANEXOS: MATRIZ DE CONSISTENCIA

TITULO	FORMULACIÓN DEL PROBLEMA	OBJETIVOS	HIPÓTESIS	VARIABLES
ANÁLISIS DE MODELOS PREDICTIVOS BASADOS EN VISIÓN COMPUTACIONAL APLICADOS AL PARALELISMO	**Problema General:**	**Objetivo General:**	**Hipótesis General:**	**Variable Independiente**
	¿Los modelos predictivos basados en Visión Computacional mejorarán al ser aplicados en la computación paralela?	Evaluar los modelos predictivos basados en Visión Computacional al ser aplicados en la computación paralela.	Los modelos predictivos basados en Visión Computacional mejoran al ser aplicados en la computación paralela.	VI: Computación paralela Dimensiones: - Granularidad - Grado de Paralelismo
	Problemas Específicos:	**Objetivos Específicos:**	**Hipótesis Específicos:**	**Variable Dependiente**
	¿Los modelos predictivos basados en Visión Computacional mejorarán la precisión al ser aplicados en la computación paralela?	Evaluar la precisión de los modelos predictivos basados en Visión Computacional al ser aplicados en la computación paralela.	Los modelos predictivos basados en Visión Computacional mejoran la precisión al ser aplicados en la computación paralela.	VD: Modelos predictivos Dimensiones: - Precisión del Modelo - Tiempo de procesamiento
	¿Los modelos predictivos basados en Visión Computacional mejorarán el tiempo de procesamiento al ser aplicados en la computación paralela?	Evaluar el tiempo de procesamiento de los modelos predictivos basados en Visión Computacional al ser aplicados en la computación paralela.	Los modelos predictivos basados en Visión Computacional mejoran el tiempo de procesamiento al ser aplicados en la computación paralela.	