Propuesta de Trabajo Profesional de Ingeniería en Informática

Oxidized Neural Orchestra

Tutor: Ing. Ricardo Andrés Veiga

Co-Tutor: Dr. Ing. José Ignacio Alvarez Hamelin

Alumnos

Alejo Ordoñez *(Padrón 108397)*
alordonez@fi.uba.ar

Lorenzo Minervino *(Padrón 107863)*
lminervino@fi.uba.ar

Marcos Bianchi Fernández *(Padrón 108921)*
mbianchif@fi.uba.ar

Facultad de Ingeniería, Universidad de Buenos Aires

Índice

1 Acta de Acuerdo

En la Ciudad Autónoma de Buenos Aires, al día veinticinco del mes de noviembre del año dos mil veinticinco se reúnen en el Departamento de Informática de la Facultad de Ingeniería de la Universidad de Buenos Aires el profesor Ing. Ricardo A. Veiga y el profesor Dr. Ing. J. Ignacio Alvarez-Hamelin, con los estudiantes de la carrera de Ingeniería Informática el Sr. Alejo Ordoñez (Padrón: 108397), el Sr. Lorenzo Minervino (Padrón: 107863) y el Sr. Marcos Bianchi Fernández (Padrón: 108921) para tratar la elección y acuerdo del tema del Trabajo Práctico Profesional para el Ciclo Superior de la carrera.

Teniendo en cuenta la propuesta presentada por los alumnos más las observaciones y mejoras propuestas por los profesores, se ha acordado el plan de trabajo para el desarrollo e implementación del trabajo práctico profesional "Oxidized Neural Orchestra" que figura en el documento adjunto.

El acuerdo consiste en las siguientes pautas:

1. Los alumnos Sr. Alejo Ordoñez, Sr. Lorenzo Minervino y Sr. Marcos Bianchi Fernández realizarán todas las etapas para el análisis y realización de pruebas del proyecto.
2. El trabajo a realizar será presentado para cumplir los requisitos de la materia Trabajo Práctico Profesional.
3. El profesor Ing. Ricardo A. Veiga acepta la función de tutor (Facultad de Ingeniería, Universidad de Buenos Aires), y el profesor Dr. Ing. J. Ignacio Alvarez-Hamelin acepta la función de co-tutor para dicho trabajo.

Alejo Ordoñez _____

Lorenzo Minervino _____

Marcos Bianchi Fernández _____

Ing. Ricardo A. Veiga _____

Dr. Ing. José Ignacio Alvarez Hamelin _____

2 Resumen

El entrenamiento de modelos de aprendizaje profundo, requiere de grandes recursos computacionales y largos tiempos de ejecución; los algoritmos que se utilizan hacen uso intensivo de CPU y requieren de capacidades de memoria muy grandes para poder almacenar tanto los datos de entrenamiento, como el modelo en sí. Junto con la revolución de la aplicación de la inteligencia artificial, se ha vuelto un tema candente el minimizar éste tiempo de entrenamiento. Para lograrlo, la estrategia más escalable es su ejecución distribuida.

El problema es que el utilizar un algoritmo distribuido de optimización para entrenar un modelo de aprendizaje profundo, viene acompañado tanto de desafíos a la hora de plantear la estrategia de distribución, como de peores (o directamente inutilizables) resultados de convergencia si no se realiza con cuidado.
En particular, si se optara por la distribución por datos, es decir, particionar el *dataset* entero y hacer que cada uno de los nodos del sistema distribuido iterase, por ejemplo, el algoritmo de *backpropagation* sobre la partición que lo tocó, y luego de la finalización del cómputo del algoritmo en cada uno de estos nodos, se intentara hacer un *reduce* de los resultados que se calcularon por separado; el modelo no quedaría optimizado sobre la totalidad del dataset dado que cada uno de los nodos habría incurrido en mucho *overfitting*.
Además, la agregación de soluciones especializadas para particiones de datos no puede computarse con algún método simple como el cálculo de un promedio del resultado para cada peso del modelo, ya que el resultado de cada iteración de los algoritmos de optimización que se utilizan para el entrenamiento, depende del anterior.

A pesar de las dificultades que existen, entre ellas las ya mencionadas, hay varias soluciones distribuidas que logran sortearlas y que proveen grandes reducciones en los tiempos de ejecución de los algoritmos de entrenamiento, a partir de la suma del poder de cómputo y de la capacidad de memoria de los nodos en los sistemas que las ejecutan. En la actualidad, dos de las soluciones más llamativas son *Parameter Server* y *All Reduce*; con más reducción en el tiempo de ejecución y mejor estabilidad en la convergencia de la optimización, respectivamente.

En este trabajo se estudian los algoritmos mencionados y sus derivados. Para ello, se busca implementar un sistema distribuido de entrenamiento de modelos de aprendizaje profundo, que funcione como base para el análisis y como punto de partida para la realización de modificaciones de los algoritmos con el objetivo de mejorar el panorama actual.

3 Estado del arte

El entrenamiento distribuido surge como una extensión de las técnicas de paralelización en *HPC* (High Performance Computing). Con los primeros enfoques de paralelismo de datos en frameworks como **TensorFlow** [1] y **PyTorch** [2], y sus implementaciones más recientes optimizadas para arquitecturas heterogéneas, la evolución de estos métodos refleja la tensión constante entre escalabilidad y coherencia de los parámetros del modelo.

Como ya se mencionó, los principales desafíos del desarrollo de algoritmos distribuidos de entrenamiento de modelos de aprendizaje profundo, son el encontrar una estrategia que, satisfactoriamente, disminuya el tiempo del entrenamiento o que provea una solución a la incapacidad de un flujo existente de almacenar el dataset o incluso el modelo en sí. Ésto sería mucho más sencillo si las iteraciones no fueran dependientes; podríamos simplemente utilizar una estrategia análoga a un *fork-join* y combinar la solución.
Pero en un *gradient-descent*, la dirección de un descenso en la superficie de costo depende del punto en el que se encuentra el proceso en un momento dado, es decir, depende de la configuración de la matriz de pesos del modelo en esa iteración.

En un esquema de paralelización del entrenamiento por datos, existe el riesgo de incurrir en *overfitting* sobre las particiones si los pesos no se sincronizan con la frecuencia suficiente. Por otro lado, si la sincronización se realiza con demasiada frecuencia, el tiempo de comunicación entre los nodos puede volverse dominante. Este costo no es en absoluto despreciable: si la comunicación representa una fracción significativa del tiempo total de entrenamiento, la distribución del cómputo pierde sentido, ya que la ejecución paralela termina siendo más lenta que el entrenamiento secuencial.

Este equilibrio entre convergencia y eficiencia de comunicación ha motivado una amplia línea de investigación. En la actualidad, existen dos enfoques fundamentales que sirven como base para el desarrollo de nuevas técnicas: los ya mencionados **Parameter Server** [3] y **All-Reduce** [4].
En el enfoque de Parameter Server, el modelo se divide en un conjunto de parámetros centralizados (no necesariamente en un único nodo), a los cuales los nodos trabajadores envían gradientes y desde los cuales reciben actualizaciones, que surgen de la agregación de estas contribuciones. Esto permite entrenamiento asincrónico y buena escalabilidad, resultando en una reducción significativa del tiempo de entrenamiento, aunque a costa de la coherencia entre copias del modelo.
En contraste, en All-Reduce todos los nodos intercambian gradientes de manera directa (utilizando, por ejemplo, implementaciones de anillo), garantizando sincronización exacta entre réplicas pero penalizando el rendimiento cuando el número de nodos crece.
Uno de los claros casos de combinación de estos algoritmos es **Strategy-Switch** [5], que inicia iterando sobre All-Reduce y, guiado por una regla empírica, sigue con Parameter Server asincrónico una vez que el modelo en entrenamiento se estabiliza; logrando así mantener la precisión del entrenamiento de *All-Reduce* y la reducción significativa del tiempo total de entrenamiento de *Parameter Server*.

En este trabajo se estudian en profundidad ambos algoritmos base y se investigan los métodos que de ellos derivan. Y se busca a partir de dicho análisis proponer mejoras mediante la combinación de estrategias o la optimización de sus componentes de comunicación y sincronización. Además se busca desarrollar un sistema que funcione como base para el análisis comparativo del desempeño de las distintas estrategias de distribución, en términos de reducción de tiempos, convergencia y escalabilidad; y para la aplicación de las posibles mejoras que se encuentren.

4 Objetivos

El trabajo tiene como objetivo principal la implementación de un sistema distribuido de entrenamiento de modelos de aprendizaje profundo, que sirva como base para la investigación de los trabajos previos, actuales y que surjan sobre esta temática. En particular, el desarrollo del trabajo conlleva:

1. Desarrollar un sistema distribuido de entrenamiento de modelos de aprendizaje profundo en Rust.
2. Implementar y comparar los distintos algoritmos que se utilicen para la ejecución del entrenamiento distribuido, sobre el sistema implementado.
3. Implementar una interfaz funcional externa para poder usar el sistema a desarrollar en Python.
4. Simular la ejecución sobre distintas configuraciones del sistema distribuido, para así obtener datos que se puedan analizar, y obtener comparaciones de los distintos algoritmos que se implementen, usando Python.
5. Confeccionar un informe detallado de la evolución del trabajo y los resultados obtenidos.

En lo académico, se busca utilizar y aplicar conocimientos adquiridos principalmente de las materias Aprendizaje Profundo, Redes y Sistemas Distribuidos; y de Fundamentos de Programación, Algoritmos y Estructuras de Datos, Paradigmas de Programación, Análisis Matemático II, Álgebra Lineal, Probabilidad y Estadística, Taller de Programación, Ciencia de Datos, Programación Concurrente, Simulación y Base de Datos. Además, existe un desafío adicional relacionado a la gestión del proyecto, la división de tareas y la estimación de tiempos, para lo cual se aplican contenidos relacionados a las materias Gestión del Desarrollo de Sistemas Informáticos e Ingeniería de Software (I y II).

5 Tecnologías

Las tecnologías que van a ser utilizadas para el desarrollo de este proyecto son:

- **Rust**: Se opta por el lenguaje de programación Rust para la implementación del sistema principal, porque ofrece: en primer lugar: la capacidad de editar código a bajo nivel, por la robustez del lenguaje, siendo que los requerimientos mínimos de compilación son más estrictos que la mayoría del resto de lenguajes, y que ofrece *fearless-concurrency*, haciendo checkeos estáticos de posibles problemas con la concurrencia de los programas, y por la relevancia que está cobrando en estos últimos tiempos.
- **Python**: Se va a usar Python para el análisis de datos obtenidos a partir de las comparaciones de los distintos algoritmos de machine-learning distribuido que serán llevados a cabo en el desarrollo del trabajo, y, por ser uno de los lenguajes más utilizados en la industria del aprendizaje profundo, para implementar una Interfaz de Funciones Externas (en inglés Foreign Function Interface, FFI) del resultado del sistema principal.
- **Docker**: Se hará uso de Docker para simular la ejecución del sistema en distintos entornos, y para agilizar la automatización de esto último.

6 Planificación

6.1 Gestión

Se establece un compromiso por parte de cada estudiante para dedicar un total de 500 horas al desarrollo del trabajo profesional. Esto representa, en promedio 15 horas semanales por persona a la ejecución de las tareas asignadas. Este compromiso se mantendrá a lo largo de 32 semanas (dos cuatrimestres). Además, se tiene previsto llevar a cabo encuentros periódicos en formato virtual entre los miembros del equipo y los tutores, cada semana. El propósito de estas reuniones es informar el avance y desarrollo del proyecto en curso. Asimismo, se abordarán aspectos como la definición de prioridades en las labores a realizar y la planificación requerida para la próxima etapa del proceso.

6.2 Tareas

Las principales tareas para llevar a cabo el desarrollo de este trabajo son:

1. Leer y analizar los trabajos previos, actuales y que surjan sobre el entrenamiento distribuido de modelos de aprendizaje profundo.
2. Investigar sobre la implementación de TensorFlow distribuido.
3. Investigar sobre la implementación distribuida de Pytorch.

Tanto TensorFlow como PyTorch son implementaciones previas del tipo de sistema que se desarrollará en este trabajo.

4. Desarrollar el sistema distribuido que sirva como base para la implementación y análisis de los algoritmos actuales en Rust. Este sistema proveerá una base sobre la cual poder probar distintos algoritmos de entrenamiento distribuido de modelos de machine learning. La idea es hacerlo tan *parametrizable* como sea posible para facilitar la posterior investigación y el desarrollo de estrategias que optimicen los tiempos de ejecución.
5. Implementar *Parameter Server* sobre el sistema desarrollado en 4.
6. Implementar *All-Reduce* sobre el sistema desarrollado en 4.
7. Implementar *Strategy-Switch* sobre el sistema desarrollado en 4. y utilizando las implementaciones de los algoritmos en 5. y 6.

Estos tres últimos puntos refieren al punto de partida de la implementación del sistema funcional y servirán como referencia para la comparación con las futuras mejoras que se estudien y desarrollen.

8. Estudiar sobre optimizaciones de comunicación entre nodos e implementarlas.
9. Estudiar sobre optimizaciones de sincronización de las copias del modelo en los distintos nodos e implementarlas.
10. Implementar una interfaz funcional externa para poder usar el sistema en Python. La idea de este punto es proveer una API fácil de usar para aquellos usuarios que trabajen con modelos de machine learning en este lenguaje, siendo que Python es el lenguaje más popular para este tipo de proyectos.
11. Testear el sistema desarrollado, con tests unitarios y de integración.
12. Simular la ejecución de los algoritmos implementados sobre el sistema distribuido base en distintas configuraciones de nodos; esto es, lograr una métrica que muestre el rendimiento del sistema, según los parámetros que este use, para distintas combinaciones de máquinas, con

distintas capacidades de cómputo, que trabajen en la ejecución.
13. Estudiar sobre optimizaciones de carga de cómputo en configuraciones heterogéneas e implementarlas.
14. Analizar los resultados obtenidos de la comparación de los algoritmos, documentar y volcar el análisis utilizando gráficos en Python. Esto abarca también los resultados obtenidos en las simulaciones mencionadas en 12.
15. Documentar el código generado y el proceso de desarrollo (decisiones que se tomaron, inconvenientes encontrados, etc.).
16. Realizar un informe detallado de la evolución del trabajo y los resultados obtenidos.

6.3 Carga horaria

Nro.	**Tarea**	**Duración (horas)**	**Responsables**
1	Leer y analizar los trabajos previos, actuales y que surjan	100	A, L y M
2	Investigar sobre la implementación de *TensorFlow* distribuido	50	A, L y M
3	Investigar sobre la implementación distribuida de *Pytorch*	50	A, L y M
4	Desarrollar el sistema distribuido en Rust	150	A, L y M
5	Implementar *Parameter Server*	100	A, L y M
6	Implementar *All-Reduce*	100	A, L y M
7	Implementar *Strategy-Switch*	100	A, L y M
8	Estudiar sobre optimizaciones de comunicación entre nodos e implementarlas	150	A, L y M
9	Estudiar sobre optimizaciones de sincronización de las copias del modelo en los distintos nodos e implementarlas	150	A, L y M
10	Implementar una interfaz funcional externa para poder usar el sistema en Python	50	A, L y M
11	Testear el sistema desarrollado, con tests unitarios y de integración	50	A, L y M
12	Simular la ejecución de los algoritmos en distintas configuraciones de nodos	100	A, L y M
13	Estudiar sobre optimizaciones de carga de cómputo en configuraciones heterogéneas e implementarlas	150	A, L y M
14	Analizar los resultados obtenidos y volcar el análisis utilizando gráficos en Python	100	A, L y M
15	Documentar el código generado y el proceso de desarrollo	50	A, L y M
16	Escribir el informe final	50	A, L y M
	Total	**1500**	

7 Referencias

[1] T. Developers, *TensorFlow*. (2021). Zenodo. doi: 10.5281/zenodo.4758419.

[2] A. Paszke *et al.*, «PyTorch: An Imperative Style, High-Performance Deep Learning Library». 2019. Disponible en: https://arxiv.org/abs/1912.01703

[3] M. Li *et al.*, «Scaling distributed machine learning with the parameter server», en *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, en OSDI'14. USA: USENIX Association, 2014, pp. 583-598.

[4] Z. Li, J. Davis, y S. Jarvis, «An Efficient Task-based All-Reduce for Machine Learning Applications», en *Proceedings of the Machine Learning on HPC Environments*, en MLHPC'17. New York, NY, USA: Association for Computing Machinery, 2017. doi: 10.1145/3146347.3146350.

[5] N. Provatas, I. Chalas, I. Konstantinou, y N. Koziris, «Strategy-Switch: From All-Reduce to Parameter Server for Faster Efficient Training», *IEEE Access*, vol. PP, pp. 1-1, ene. 2025, doi: 10.1109/ACCESS.2025.3528248.