Implementación de una aplicación de programación paralela a una red neuronal Deep Learning

Liliana Ochoa
Ingeniería de Sistemas
Universidad de Antioquia
liliana.ochoae@udea.edu.co

Yohan López
Ingeniería de Sistemas
Universidad de Antioquia
Yohan.lopez@udea.edu.co

***Abstract*— Las redes neuronales y el Deep Learning han logrado avances significativos en robótica, visión artificial, procesamiento y comprensión del lenguaje natural (NLP y NLU). Esto ha llevado a un incremento en el uso de estas tecnologías por parte de las empresas para optimizar y automatizar procesos. Estos modelos requieren grandes cantidades de datos y un largo proceso de aprendizaje, implicando múltiples pruebas con diversas combinaciones de parámetros y estructuras del modelo. Para mejorar la eficiencia, la paralelización de entrenamientos es esencial. En Python, la librería Horovod facilita esta paralelización en TensorFlow y PyTorch con mínimas modificaciones de código. Este proyecto tiene como objetivo desarrollar una herramienta que automatice la paralelización del código de entrenamiento de redes neuronales. Se propone crear un compilador source-to-source que aplique automáticamente las transformaciones necesarias para utilizar Horovod en códigos programados en TensorFlow.Keras y PyTorch. Esta herramienta sería valiosa para la comunidad, ahorrando tiempo y esfuerzo a los investigadores al evitar la necesidad de comprender y modificar manualmente el código para la paralelización.**

Palabras clave: redes neuronales, Deep Learning, paralelización, Horovod, TensorFlow, PyTorch, automatización, compilador source-to-source.

I. INTRODUCCIÓN

Una red neuronal artificial (RNA) es un modelo computacional inspirado en el comportamiento observado en su homólogo biológico. El cerebro puede considerarse un sistema altamente complejo, donde se calcula que hay aproximadamente 100 mil millones de neuronas formando un entramado de más de 500 billones de conexiones neuronales. El objetivo de la red neuronal es resolver los problemas de la misma manera que el cerebro humano, aunque las redes neuronales son más abstractas. Las redes neuronales actuales suelen contener desde unos miles a unos pocos millones de unidades neuronales.

En una red neuronal artificial cada neurona está conectada con otras a través de unos enlaces. En estos enlaces el valor de salida de la neurona anterior es multiplicado por un valor de peso.

Estos pesos en los enlaces pueden incrementar o inhibir el estado de activación de las neuronas adyacentes. Estos sistemas son capaces de descubrir y aprender patrones en un conjunto de datos para formarse a sí mismos, en lugar de ser programados de forma explícita, y sobresalen en áreas donde la detección de soluciones o características es difícil de expresar con la programación convencional.

Las redes neuronales se han utilizado para resolver una amplia variedad de tareas, como la visión por computador y el reconocimiento de voz, que son difíciles de resolver usando la ordinaria programación basada en reglas.

II. MARCO TEÓRICO

a. Contexto

Las empresas de hoy en día están necesitando, cada vez más, aumentar sus recursos computacionales para poder ser competitivas en un mercado altamente tecnológico. Por ello, la popularidad de las redes neuronales con Python va en aumento. Pues la agilidad que aporta este lenguaje y la potencia de procesamiento de datos de las RNA hace que ambos elementos en conjunto sean la opción más eficaz y eficiente para las empresas de alto rendimiento. Además, las redes neuronales y su subcampo, el deep learning, han conseguido resultados impresionantes en áreas como la robótica, la visión artificial, el Natural Language Processing (NLP) y el Natural Language Understanding (NLU).

b. Descripción del problema

Estos modelos computacionales utilizados en deep learning requieren de mucho tiempo en su etapa de entrenamiento, por lo que su paralelización se ha convertido en uno de los temas clave en la actualidad. Para las empresas es primordial optimizar estos tiempos, principalmente para no tener a sus investigadores esperando a obtener resultados sin poder avanzar.

Existen librerías de código mediante las cuales el programador, con poco esfuerzo, puede ejecutar de manera paralela estos entrenamientos, como por ejemplo Horovod[2] [3], que permite simplificar este proceso de paralelización mediante la anotación de directivas en el código secuencial en Python.

En este proyecto se implementa una herramienta que automatiza el proceso de paralelización para códigos Python que usan las librerías TensorFlow-Keras[4] (tf.keras de ahora en adelante) o PyTorch[5], creadas por Google y Facebook correspondientemente. La idea es que esta herramienta funcione a modo de compilador Source-to-Source[6] y permita al programador obtener código ejecutable en paralelo a partir de su código secuencial en Python.

c. Conceptos básicos

- Redes Neuronales: son modelos computacionales inspirados en la estructura y funcionamiento del cerebro humano, compuestos por nodos (neuronas) que procesan información y aprenden patrones a

través de la conexión de estos nodos.

- Deep Learning: es un subcampo del aprendizaje automático que utiliza redes neuronales profundas con muchas capas (deep neural networks) para modelar y resolver problemas complejos, como elreconocimiento de imágenes y el procesamiento del lenguaje natural.

- Paralelización: es una técnica que consiste en dividir un problema o tarea en partes más pequeñas que se pueden ejecutar simultáneamente en múltiples procesadores, acelerando el tiempo de cómputo.

- Horovod: biblioteca de código abierto diseñada para la paralelización del entrenamiento de modelos de deep learning en múltiples GPUs y nodos, facilitando el escalado eficiente en clústeres distribuidos.

- TensorFlow: biblioteca de código abierto desarrollada por Google para la construcción y entrenamiento de modelos de machine learning y deep learning, con soporte para la ejecución en diversas plataformas de hardware.

- PyTorch: biblioteca de código abierto desarrollada por Facebook para el desarrollo y entrenamiento de modelos de machine learning y deep learning, conocida por su flexibilidad y facilidad de uso con una interfaz intuitiva y dinámica.

- Compilador Source-to-Source: herramienta que traduce código fuente escrito en un lenguaje de programación a código fuente en otro lenguaje de programación, manteniendo el nivel de abstracción y facilitando la portabilidad entre diferentes lenguajes y plataformas.

III JUSTIFICACIÓN

TensorFlowy PyTorch ofrecen sus propias implementaciones (*TF.distributed* y *PyTorch.dataParallel*) para ejecutar entrenamientos de modelos deep learning de forma paralela, no obstante también existen librerías externas como Horovod, creada por Uber, que mediante la inserción de determinadas directivas en el código permite paralelizar estos entrenamientos tanto en TensorFlow como en PyTorch. Horovod le ahorra al programador el tener que conocer a bajo nivel el funcionamiento de su modelo ya que de manera transparente al usuario Horovod identifica los recursos que tiene disponibles y distribuye la ejecución del programa de manera eficiente entre ellos. Sin embargo, para que esto funcione es indispensable que el programador tenga los conocimientos necesarios para utilizar correctamente las directivas de Horovod, ya que éstas no son triviales y en la mayoría de los casos será necesario aplicar pequeñas modificaciones en el código original para poder hacer un uso correcto de ellas. Por este motivo se ha decidido implementar un compilador *Source-to-Source* el cual a partir de un código en Python, programado tanto con tf.keras como con PyTorch, sea capaz de realizar las adaptaciones necesarias para obtener un código paralelizable mediante Horovod.
Esto le aportaría al programador un nivel más de abstracción a la hora de paralelizar el entrenamiento de su modelo, ya que únicamente debería encargarse de escribir un código que funcione de manera secuencial.

Tecnologías a utilizar
Ya que el compilador permitirá transformar código Python secuencial a Python paralelo parece lógico utilizar Python (python 3.x) como el lenguaje para su implementación. Además, este lenguaje incorpora librerías tales como ast[7] (Abstract Syntax Tree[8]), astunparse o inspect que facilitan la tarea de inspeccionar y procesar el código mediante árboles sintácticos.

IV METODOLOGÍA

Punto de partida
Se ha realizado una búsqueda de posibles soluciones al problema planteado y no se han encontrado herramientas que a día de hoy automaticen el proceso de paralelización de código para tf.keras o pytorch. Por lo tanto, en el desarrollo de este compilador se parte de cero. La inexistencia de herramientas similares puede deberse al hecho de tratarse de una tecnología bastante reciente, Horovod se encuentra en la versión 0.19 a junio de 2020 y está en constante desarrollo. Además, el campo del deep learning ha generado mucha repercusión en los últimos años debido a la rápida evolución de librerías como TensorFlow y PyTorch. Estos factores dificultan la aparición de herramientas como la que se plantea en este proyecto, ya que para garantizar el funcionamiento y la aplicabilidad de éstas sería necesario realizar adaptaciones y cambios de manera frecuente para estar al día con respecto a las principales librerías de Deep learning.

1. Tensorflow.Keras (tf.keras)

Keras es una librería de Redes Neuronales de código abierto escrita en Python capaz de ejecutarse sobre TensorFlow, desarrollada y mantenida por François Chollet 2, ingeniero de Google.

1.1 Definición del Modelo
En tf.keras, se puede crear un modelo secuencial definiendo una serie de capas, que incluyen una capa de entrada, capas intermedias (ocultas) y una capa de salida. Por ejemplo, un modelo secuencial puede ser definido con una capa de entrada que recibe 500 parámetros, una capa oculta de 256

neuronas, seguida de capas ocultas de 128 y 64 neuronas respectivamente, y finalmente una capa de salida de 2 neuronas. La librería tf.keras deduce automáticamente la forma de los tensores entre capas después de la primera capa, por lo que solo es necesario especificar el tamaño de entrada para la primera capa. Todas las capas en este modelo son densas, es decir, cada neurona está conectada con todas las neuronas de la capa anterior. El modelo tendrá que aprender un total de 169,538 pesos y sesgos.

1.2 Configuración del Proceso de Aprendizaje

Una vez definido el modelo, se configura el proceso de aprendizaje utilizando el método model.compile(), donde se especifican la función de pérdida, el optimizador y las métricas de evaluación. En el ejemplo dado, se utiliza la función de pérdida categorical_crossentropy, el optimizador Stochastic Gradient Descent (SGD) con una tasa de aprendizaje de 0.01, y la métrica de precisión (accuracy).

1.3 Entrenamiento del Modelo

Con el modelo definido y configurado, se procede al entrenamiento utilizando el método model.fit(). Este método recibe los datos de entrenamiento y el número de épocas (iterations completas sobre el conjunto de datos de entrenamiento). Durante cada iteración, el optimizador ajusta los pesos del modelo para minimizar la función de pérdida, comparando las salidas del modelo con las salidas esperadas.

1.4 Evaluación y Guardado del Modelo

Después de entrenar el modelo, se puede evaluar su rendimiento con datos nuevos (datos de prueba) utilizando model.evaluate(), que devuelve la pérdida (loss) y la precisión (accuracy). Si los resultados son satisfactorios, el modelo entrenado se puede guardar usando model.save(), especificando la ruta del archivo donde se almacenará el modelo.

2. PyTorch

2.1 Definición del Modelo

En PyTorch, la definición de modelos de redes neuronales se realiza utilizando el módulo torch.nn. Las arquitecturas se pueden definir mediante una clase que herede de Sequential, similar a tf.keras. En esta clase, se especifican las capas del modelo y sus dimensiones, además de las funciones de activación. Por ejemplo, un modelo secuencial puede incluir una capa de entrada con 500 parámetros y 256 neuronas, seguida de capas ocultas con 128 y 64 neuronas respectivamente, y una capa de salida con 2 neuronas, aplicando funciones de activación ReLU en las capas ocultas y Softmax en la capa de salida.

2.2 Configuración del Proceso de Entrenamiento

a diferencia de tf.keras, en PyTorch no existe un método equival ente a model.compile().

En su lugar, el programador debe definir explícitamente la función de pérdida y el optimizadorssEntropyLoss y el optimizador Stochastic Gradient Descent (SGD) con una tasa de aprendizaje de 0.01. Estos componentes se invocan manualmente durante el proceso de entrenamiento.

2.3 Entrenamiento del Modelo

El entrenamiento en PyTorch requiere que el desarrollador programe manualmente el bucle de entrenamiento. Durante cada iteración del bucle de entrenamiento (épocas), se siguen estos pasos:

- Cargar subconjuntos de datos de entrenamiento (batch).
- Reiniciar los gradientes a cero.
- Realizar la propagación hacia adelante (forward propagation) para obtener predicciones.
- Calcular la pérdida (loss) utilizando la función de pérdida.
- Realizar la propagación hacia atrás (backward propagation) para calcular los gradientes.
- Actualizar los pesos del modelo mediante el optimizador.

Este proceso se repite para cada batch de datos y se monitoriza la pérdida acumulada para evaluar el progreso del entrenamiento.

2.4 Evaluación y Guardado del Modelo

Para evaluar el modelo en PyTorch, se debe programar un bucle de evaluación que utilice los datos de prueba. Durante este proceso:

- Se desactiva el cálculo de gradientes.
- Se realiza la propagación hacia adelante para obtener predicciones.
- Se comparan las predicciones con las etiquetas reales para calcular la precisión (accuracy).

Una vez satisfechos con los resultados, el modelo entrenado se puede guardar utilizando torch.save(), especificando la ruta del archivo donde se almacenará el modelo.

Paralelización con Horovod

Horovod permite la paralelización del entrenamiento de modelos en múltiples GPUs, facilitando la distribución del trabajo en entornos de aprendizaje profundo. A continuación, se explica su funcionamiento y uso en tf.keras y PyTorch.

3. Paralelización con Horovod

En Horovod, cada nodo se comunica con sus nodos vecinos en un proceso iterativo. Durante las primeras iteraciones, los valores recibidos se agregan al búfer de

datos del nodo, y en las siguientes, los valores recibidos reemplazan los valores en el búfer del nodo.

3.1 Horovod sobre tf.keras

Para utilizar Horovod con tf.keras, se deben realizar las siguientes modificaciones:

- Inicialización de Horovod: Importar la librería correspondiente y llamar a la función de inicialización.
- Asignar cada GPU a un único proceso:
    - Para TensorFlow versiones anteriores a la 2.0.0, se debe configurar la GPU visible para cada proceso.
    - Para TensorFlow 2.0.0 y superiores, se debe habilitar el crecimiento de memoria de la GPU y configurar la GPU visible para cada proceso.
    - Escalado del learning rate: Ajustar el learning rate proporcionalmente al número de GPUs utilizadas.
- Distribución del optimizador: Utilizar un optimizador distribuido que delega el cálculo del gradiente al optimizador original y aplica los gradientes resultantes.
- Propagación del estado inicial de las variables: Emitir el estado inicial de las variables globales a todas las GPUs para asegurar una inicialización consistente al inicio del proceso de entrenamiento.
- Otras modificaciones: Identificar y aislar el código que debe ejecutarse en una única GPU para evitar redundancias y asegurar una ejecución correcta en paralelo.

3.2Horovod sobre PyTorch

Para utilizar Horovod con PyTorch, se deben realizar las siguientes modificaciones:

- Inicialización de Horovod: Importar la librería correspondiente y llamar a la función de inicialización.
- Asignar cada GPU a un único proceso: Configurar la GPU visible para cada proceso si las GPUs están disponibles.
- Escalado del learning rate: Ajustar el learning rate proporcionalmente al número de GPUs utilizadas.
- División del conjunto de datos: Utilizar un sampler distribuido para repartir el conjunto de datos entre los nodos, asegurando que cada nodo trabaje con su subconjunto de datos.
- Distribución del optimizador: Utilizar un optimizador distribuido que delega el cálculo del gradiente al optimizador original y aplica los gradientes resultantes.
- Propagación del estado inicial de las variables: Emitir el estado inicial de las variables del modelo a todas las GPUs para asegurar una inicialización consistente al inicio del proceso de entrenamiento.
- Otras modificaciones: Identificar y aislar el código que debe ejecutarse en una única GPU para evitar redundancias y asegurar una ejecución correcta en paralelo.

Implementación del compilador

El primer paso en la implementación del compilador es comprender las anotaciones de Horovod necesarias para paralelizar un código escrito en tf.keras o PyTorch. Posteriormente, se elabora un compilador que automatiza este proceso.

Para ello, es esencial encontrar la manera más eficaz de analizar y modificar cualquier código escrito en Python. Python dispone de librerías como re (Regular Expression) y ast (Abstract Syntax Tree) que son útiles para esta tarea. Es necesario entender y diferenciar entre errores de compilación, errores de sintaxis y errores de ejecución.

4.1Tratamiento del código escrito en Python

4.1.1Expresiones regulares

Las expresiones regulares (regex) son patrones de búsqueda que pueden encontrar y modificar combinaciones de caracteres en una cadena de texto. Aunque pueden parecer una buena solución para transformar código secuencial en paralelo, presentan limitaciones. No pueden captar el contexto completo del código y son inadecuadas para manejar estructuras complejas o múltiples formas de una misma función.

4.1.2Árboles de sintaxis abstracta

Un árbol de sintaxis abstracta (AST) es una representación en forma de árbol de la estructura sintáctica del código fuente. Python ofrece la librería ast, que permite generar el AST del código mediante ast.parse(). Usar un AST para modificar el código evita errores de sintaxis y permite realizar modificaciones complejas con mayor precisión. Una vez familiarizados con la nomenclatura de la librería ast, es sencillo aplicar estas modificaciones.

4.2 Transformación del código secuencial a paralelo

El compilador recibe el código fuente original, genera su AST y realiza las modificaciones necesarias. Primero, elimina los comentarios multi-línea que pueden complicar el análisis. Luego, añade los imports necesarios para Horovod y otras funciones auxiliares.

4.2.1 Adaptación de tf.keras

En tf.keras, se modifican las funciones model.compile() y model.fit(). Se añade un optimizador distribuido y se ajustan los callbacks y el número de epochs según el número de GPUs disponibles.

4.2.2 Adaptación de PyTorch

En PyTorch, se crean DistributedSamplers para repartir el conjunto de datos entre todas las GPUs y se adaptan los DataLoaders. El optimizador se convierte en distribuido y se sincroniza el estado inicial del modelo en todas las GPUs.

Finalmente, se adaptan las funciones de evaluación y guardado del modelo. Se asegura que solo la GPU con índice 0 realiza estas tareas, evitando errores de ejecución y garantizando la coherencia en el guardado del modelo entrenado.

V RESULTADOS

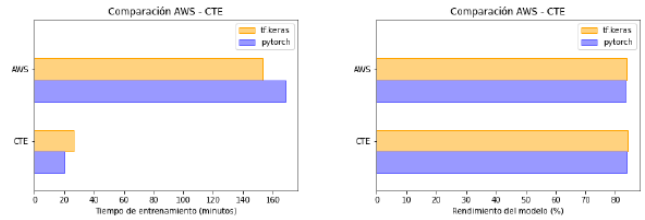
Para realizar las pruebas de rendimiento en AWS se han utilizado los mismos códigos Horovod que en las pruebas de CTE, no obstante es importante tener en cuenta que las instancias G3 cómo se ha comentado anteriormente utilizan GPUs NVIDIA M60. Estas tarjetas gráficas son bastante inferiores a las NVIDIA V100 en varios aspectos:

Ancho de bus de memoria: La V100 dispone de un ancho de bus de 4.096 bits, mientras que la M60 cuenta con únicamente 256 bits de ancho de bus de memoria.

Capacidad de memoria: Una NVIDIA V100 dispone de 16GB de memoria HBM2. Una M60 tiene una memoria GDDR5 de 8GB.

Ancho de banda de memoria: Hasta 900GB/s de ancho de banda es capaz de ofrecer la NVIDIA V100, muy superiores a los 160GB/s de la M60.

Por estas razones, es comprensible que los resultados obtenidos en AWS sean bastante peores que los de CTE. Si se quisiera utilizar instancias equipadas con NVIDIA V100 se deberían escoger instancias de la familia P39, cuyo coste lógicamente es mucho más elevado.



(a) Tiempos de entrenamiento (b) *Accuracy* de los modelos

Como era de esperar la diferencia en los tiempos de entrenamiento entre AWS y CTE es muy notable, el modelo tarda prácticamente siete veces más tiempo en entrenar en las instancias G3 de AWS. Sin embargo la *accuracy* del modelo una vez entrenado es muy similar, tanto para tf.keras como pytorch.

VI CONCLUSION

En este trabajose destacó el desarrollo de una herramienta para automatizar la paralelización del entrenamiento de redes neuronales en Python usando Horovod. Con esta herramienta, se simplificó el proceso de paralelización, ahorrando tiempo y esfuerzo a los investigadores.

Se implementó un compilador source-to-source en Python, aprovechando librerías como ast, astunparse, y inspect. Se realizaron pruebas de rendimiento en diferentes entornos, mostrando que la herramienta es efectiva y puede generar código paralelizable sin comprometer la precisión del modelo. Aunque se observaron diferencias en los tiempos de entrenamiento debido al hardware, la precisión de los modelos fue similar.

VII BIBLIOGRAFÍA

[1] Yang S. Build up a Neural Network with python - Towards Data Science. 2019. url: https://towardsdatascience.com/build-up-a-neural-network-with-python-7faea4561b31 (visitado 24-02-2024).

[2] Alexander Sergeev y Mike Del Balso. “Horovod: fast and easy distributed deep learning in TensorFlow”. En: (feb. de 2018). url: http://arxiv.org/abs/1802.05799 (visitado 24-02-2024).

[3] Meet Horovod: Uber’s Open Source Distributed Deep Learning Framework. url: https://eng.uber.com/horovod/ (visitado 24-02-2024).

[4] tensorflow/tensorflow/python/keras at master · tensorflow/tensorflow. url: https://github.com/tensorflow/tensorflow/tree/master/tensorflow/python/keras (visitado 24-02-2024).

[5] pytorch/pytorch: Tensors and Dynamic neural networks in Python with strong GPU acceleration. url: https://github.com/pytorch/pytorch (visitado 24-02-2024).

[6] Source-to-source compiler - Wikipedia. url: https://en.wikipedia.org/wiki/Source-to-source_compiler (visitado 24-02-2024).

[7] ast — Abstract Syntax Trees — Python 3.8.2rc2 documentation. url: https://docs.python.org/3/library/ast.html (visitado 24-02-2024).

[8] Iulian Neamtiu, Jeffrey S. Foster y Michael Hicks. Understanding Source Code Evolution Using Abstract Syntax Tree Matching. Saint Louis, Missouri: ACM, mayo de 2005.

[9] Abstract syntax tree - Wikipedia. url: https://en.wikipedia.org/wiki/Abstract_syntax_tree (visitado 24-02-2024).

[10] Pablo Domínguez. En qué consiste el modelo en cascada - Gestiona tu proyecto de desarrollo - OpenClassrooms. url: https://openclassrooms.com/en/courses/4309151-

gestionatu-proyecto-de-desarrollo/4538221-en-que-consiste-el-modelo-en-cascada
(visitado 24-02-2024).