Proyecto Final: Programación Concurrente y Distribuida

Implementación y Paralelización de una Red Neuronal (MLP)

Descripción de la Actividad

10 de noviembre de 2025

Índice

1 Contexto del Problema

Hoy en día, el *Deep Learning* es el motor de la inteligencia artificial. Sin embargo, el entrenamiento de estas redes neuronales es una tarea computacionalmente *extremadamente* costosa, que involucra operaciones masivas de álgebra lineal. Este costo solo puede ser manejado mediante el uso de cómputo paralelo, ya sea en CPUs multi-núcleo o en GPUs (Unidades de Procesamiento Gráfico).

En este proyecto, dejaremos de lado las "cajas negras" (como `TensorFlow`, `Keras`, `PyTorch`) y construiremos una red neuronal desde sus cimientos para entender dónde están los cuellos de botella computacionales y cómo podemos resolverlos usando diferentes paradigmas de programación concurrente.

2 Objetivos del Proyecto

El objetivo principal de este proyecto **no es** construir la red neuronal más precisa, sino **analizar y comparar el rendimiento** de diferentes implementaciones secuenciales y paralelas.

Al finalizar, el estudiante deberá ser capaz de:

- Implementar un algoritmo de *Deep Learning* (MLP) desde cero, entendiendo su base matemática.
- Identificar los cuellos de botella computacionales (pista: multiplicación de matrices).
- Aplicar técnicas de paralelismo en CPU con memoria compartida (`OpenMP`) y memoria distribuida (`multiprocessing`).
- Aplicar técnicas de paralelismo masivo en GPU (`CUDA` o `PyCUDA`).
- Medir y analizar métricas de rendimiento como el *Speedup*, el *Overhead* y la Ley de Amdahl.
- Entender la problemática de la transferencia de datos (Host-Device) en cómputo GPGPU.

3 Especificaciones Técnicas

- **Dataset: MNIST**. Consiste en 60,000 imágenes de entrenamiento y 10,000 de prueba de dígitos escritos a mano (0-9).
- **Formato de Datos:** Cada imagen es de 28x28 píxeles. Deberán "aplanar" cada imagen a un vector de **784** características (28 * 28 = 784).
- **Arquitectura (ObligatorIA):** Perceptrón Multicapa (MLP) de **3 capas**.
  - **Capa de Entrada:** 784 neuronas (fijo).
  - **Capa Oculta:** `N` neuronas (a elección del estudiante). Se recomienda un valor entre 256 y 1024.
  - **Capa de Salida:** 10 neuronas (fijo, una para cada dígito).
- **Restricción Fundamental: NO** se permite el uso de librerías de Deep Learning (`TensorFlow`, `Keras`, `PyTorch`, `Caffe`, etc.).
  - **Permitido en Python:** `NumPy` (para manejo de matrices), `multiprocessing`, `PyCUDA`, `matplotlib` (para gráficas).
  - **Permitido en C/C++:** `OpenMP`, `CUDA`, librerías estándar.

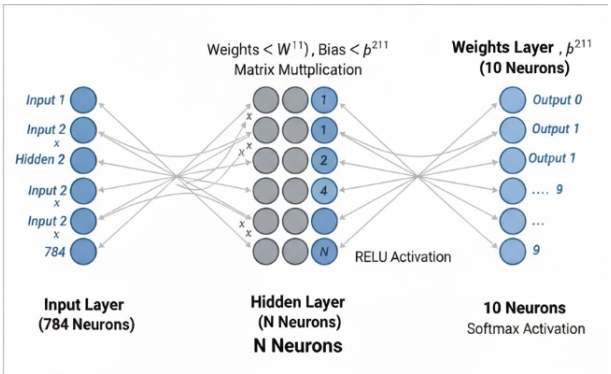

Figura 1: Arquitectura del Perceptrón Multicapa (MLP) a implementar

4 Fundamentación Matemática (MLP y Backpropagation)

Para implementar la red, deben entender el *forward pass* (cómo se hace una predicción) y el *backward pass* (cómo aprende la red).

4.1 Notación

- x: Vector de entrada (una imagen aplanada, 784×1).
- y: Etiqueta real (ej. "7"), en formato *one-hot* (un vector de 10×1 con un 1 en la posición 7).
- $W^{[1]}, b^{[1]}$: Pesos ($784 \times N$) y biases ($N \times 1$) de la capa oculta.
- $W^{[2]}, b^{[2]}$: Pesos ($N \times 10$) y biases (10×1) de la capa de salida.
- α: Tasa de aprendizaje (ej. 0.01).

4.2 Fase 1: Propagación Hacia Adelante (Forward Propagation)

Es el proceso de predicción.

1. **Cálculo Capa Oculta (Pre-activación):**

$$z^{[1]} = W^{[1]T} \cdot x + b^{[1]}$$

2. **Activación Capa Oculta:** Usaremos la activación **ReLU**, que es simple y eficiente.

$$a^{[1]} = \text{ReLU}(z^{[1]}) = \max(0, z^{[1]})$$

3. **Cálculo Capa Salida (Pre-activación):**

$$z^{[2]} = W^{[2]T} \cdot a^{[1]} + b^{[2]}$$

4. **Activación Capa Salida:** Usaremos **Softmax** para convertir los resultados en probabilidades.

$$a^{[2]} = \hat{y} = \text{Softmax}(z^{[2]})_j = \frac{e^{z_j^{[2]}}}{\sum_{k=1}^{10} e^{z_k^{[2]}}}$$

4.3 Fase 2: Cálculo de la Pérdida (Loss)

Usamos *Categorical Cross-Entropy* para medir qué tan mala fue la predicción ($\hat{y}$) comparada con la realidad (y).

$$L(y, \hat{y}) = -\sum_{j=1}^{10} y_j \log(\hat{y}_j)$$

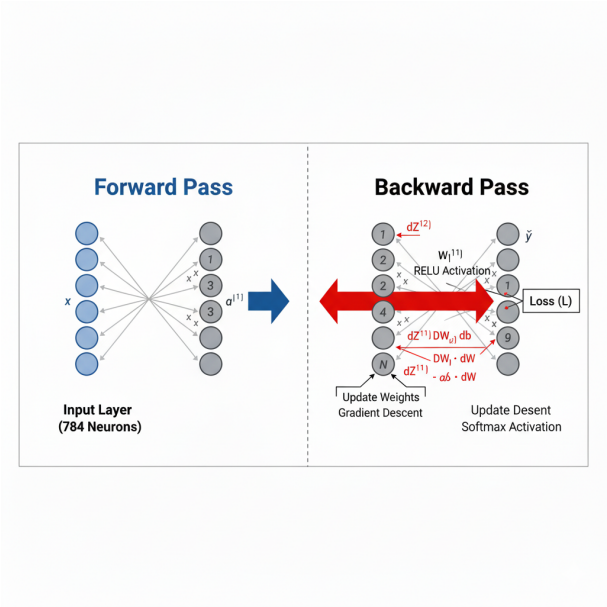

Figura 2: Visión conceptual del Forward y Backward Pass.

4.4 Fase 3: Propagación Hacia Atrás (Backward Propagation)

Aquí es donde la red aprende. Usamos la "regla de la cadena" (cálculo) para encontrar cómo un pequeño cambio en cada peso afecta la pérdida total.

$$
\begin{aligned}
dz^{[2]} &= a^{[2]} - y \\
dW^{[2]} &= a^{[1]} \cdot dz^{[2]T} \\
db^{[2]} &= dz^{[2]} \\
da^{[1]} &= W^{[2]} \cdot dz^{[2]} \\
dz^{[1]} &= da^{[1]} * \text{ReLU}'(z^{[1]}) \\
dW^{[1]} &= x \cdot dz^{[1]T} \\
db^{[1]} &= dz^{[1]}
\end{aligned}
$$

(Nota: Las ecuaciones están simplificadas para un solo ejemplo x. En la práctica, se promedian los gradientes sobre un "mini-batch" de varias imágenes).

4.5 Fase 4: Actualización de Pesos

Finalmente, actualizamos los pesos en la dirección opuesta al gradiente, escalado por la tasa de aprendizaje α.

$$
\begin{aligned}
W^{[1]} &= W^{[1]} - \alpha \cdot dW^{[1]} \\
b^{[1]} &= b^{[1]} - \alpha \cdot db^{[1]} \\
W^{[2]} &= W^{[2]} - \alpha \cdot dW^{[2]} \\
b^{[2]} &= b^{[2]} - \alpha \cdot db^{[2]}
\end{aligned}
$$

5 Fases de Implementación (Las 6 Versiones)

Deberán implementar y medir el tiempo de entrenamiento (ej. para 10 *epochs*) en los siguientes escenarios.

5.1 Escenario 1: Baseline Secuencial

El punto de referencia para todas las comparaciones.

- **1a. Python Secuencial:** Implementar el MLP completo (Fases A, B, C, D) usando **solo** `NumPy` para las operaciones matriciales.
- **1b. C Secuencial:** Re-implementar el MLP en C (o C++) puro. Deberán escribir sus propias funciones para la multiplicación de matrices y el manejo de memoria.

5.2 Escenario 2: Paralelismo en CPU

- **2a. Python + `multiprocessing`:** Usando la base de NumPy, deberán aplicar **Paralelismo de Datos**. La idea es dividir el *mini-batch* de entrenamiento entre varios procesos *worker*. Cada *worker* calcula los gradientes (Fase C) para su sub-lote. El proceso maestro recolecta, promedia los gradientes y actualiza los pesos (Fase D).
- **2b. C + `OpenMP`:** Usando la base de C, deberán identificar los bucles `for` más costosos (pista: están en la multiplicación de matrices de las Fases A y C) y paralelizaros usando directivas `#pragma omp parallel for`.

5.3 Escenario 3: Paralelismo Masivo en GPU

- **3a. `CUDA` o 3b. `PyCUDA`:** Deberán escribir un *kernel* de `CUDA` para acelerar la operación más costosa: la multiplicación de matrices (GEMM). Deberán portar el modelo de C (para `CUDA`) o el de Python (para `PyCUDA`) para que llame a este *kernel* en la GPU, gestionando la transferencia de memoria (Host-a-Device y Device-a-Host).

6 Entregable Final y Evaluación

El entregable consiste en tres partes obligatorias.

6.1 Código Fuente

- Un repositorio (`git`) con carpetas separadas para cada una de las implementaciones.
- El código debe estar limpio, comentado y con un `README.md` que explique cómo compilarlo y ejecutarlo.

6.2 Informe Técnico

Este es un componente principal de la calificación. Debe incluir:

- **Introducción:** Objetivos del proyecto.
- **Arquitectura:** Describir la arquitectura elegida (¿Por qué escogieron `N` neuronas?).
- **Metodología de Pruebas:** Descripción del hardware (CPU, # de núcleos, GPU) y cómo se midieron los tiempos.
- **Análisis de Rendimiento :**
  - **Tabla Comparativa:** Una tabla con el tiempo total de entrenamiento (ej. 10 *epochs*) para todas las implementaciones.
  - **Gráfica de *Speedup* (C+`OpenMP`):** Una gráfica que muestre el *speedup* (Tiempo Secuencial / Tiempo Paralelo) de la versión C+`OpenMP` usando 1, 2, 4, 8, ... N hilos (hasta el máximo de su CPU).

    - **Análisis Ley de Amdahl:** Discutir la gráfica anterior. ¿Es el *speedup* lineal? ¿Por qué sí o por qué no? (Mencionar *overhead* y porción secuencial).
    - **Análisis Python `multiprocessing`:** Comparar el *speedup* de `multiprocessing` vs. `OpenMP`. ¿Cuál es más eficiente y por qué? (Mencionar *overhead* de creación de procesos y serialización).
    - **Análisis `CUDA/PyCUDA`:**
        - Presentar una gráfica de *profiling* que muestre el tiempo desglosado en: (1) Transferencia CPU→GPU, (2) Ejecución del Kernel, (3) Transferencia GPU→CPU.
        - **Análisis de `batch_size`:** Ejecutar la versión de `CUDA` con un `batch_size` pequeño (ej. 16) y uno grande (ej. 512). Explicar por qué el rendimiento cambia tan drásticamente.
- **Conclusiones:** Resumen de los hallazgos. ¿Cuándo vale la pena usar cada paradigma? ¿Qué aprendieron?

6.3 Presentación Oral (Obligatoria)

- Deberán preparar una presentación de 10-15 minutos.
- El objetivo es sustentar los hallazgos del informe técnico, enfocándose en el **Análisis de Rendimiento**.
- Se deben presentar las diapositivas (ej. en PDF) junto con el informe.
- Todos los miembros del equipo deben participar en la presentación.