CS2006: Operating Systems (Fall 2025)

Semester Project

Groups

This is a group project. There should be **2 members per group**. There is no restriction on the selection of group members. Students are allowed to make groups according to their preferences. However, **cross-section groups are not allowed.**

Deadline

Please start your work on time, as deadline will **not** be extended.

- **Project Deadline:** 5th December 2025
- **Late submissions**
  - On 1-12 hours late, deduction of 10% marks.
  - On 12-24 hours late, 20% marks deduction
  - Rest will be marked zero.

Instructions

You can use only C++ and system calls and libraries studied in OS labs (e.g. fork(), wait(), pipes (Named/unnamed), pthread library, mutexes, semaphores etc.) to implement the mentioned scenario. The solution should be implemented in Linux (preferably Ubuntu) as demo will be on machines where Ubuntu is installed. Even if you use any other Linux/Mac distro or any other OS, make sure your program runs as expected in Ubuntu environment.

Submission

All submissions MUST be uploaded on **Google Classroom** strictly following deadlines. Solutions sent on emails will not be graded. To avoid last-minute problems (unavailability of Google Classroom, load shedding, network down, etc.), you are strongly advised to start working on the project from day one. Each group will **submit a .zip file ,**named according to the roll numbers of the group members, containing following:

- All .**cpp source files**
- A **project report** covering the following:
    - **Title & Team Details:** Mention the project title and names of all group members.
    - **Problem Statement:** Clearly define the problem being solved.
    - **System Design & Architecture**: Describe how layers are processes, neurons are threads and explain IPC and synchronization.
    - **Implementation Details**: Explain forward/backward passes, dynamic process/thread creation, input/output handling, and resource management.
    - **Sample Output**: Include screenshots or tables showing forward and backward pass outputs layer-wise.
    - **Work Division**: Describe each team member's contribution.
    - **Challenges Faced**: Mention difficulties encountered and how they were solved.

Plagiarism

Plagiarism is strictly prohibited. Any group submitting plagiarized work will receive zero marks. **Code similarity beyond 20% will automatically be considered plagiarism.**

- 25% marks deduction for students with percentages between 21% - 30%.
- 50% marks deduction for students with percentages between 31% - 50%.
- Straight away 0 for those having percentage greater than 50%.

Multi-Core Neural Network Simulation Using Processes and Threads

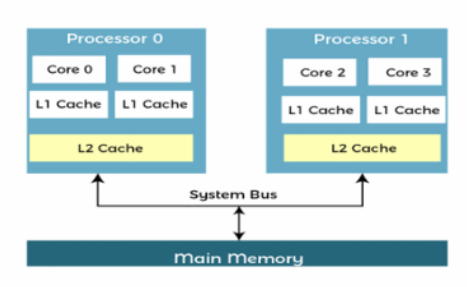
Problem Statement

This project aims to simulate a neural network architecture on a multi-core processor by leveraging operating system-level constructs such as processes and threads. Each layer of the neural network is implemented as a separate process, while individual neurons within a layer are represented as threads. This design takes advantage of parallel processing capabilities in modern multi-core systems to enhance computational efficiency during forward and backward signal propagation. Inter-process communication (IPC) is handled via pipes, allowing seamless transfer of weights, biases, and computed outputs between layers. The project focuses on illustrating forward and backward passes and does not perform actual weight updates or full error handling.

Project Description

Neural networks are a widely used machine learning technique that simulates the structure and functioning of the human brain to address complex computational problems. A typical neural network comprises interconnected layers of neurons, where each neuron accepts multiple inputs, processes them, and generates an output signal.

In modern multi-core processors, multiple processing units (cores) can execute threads simultaneously, thereby significantly increasing computational throughput. To harness this capability, the proposed operating system will parallelize neural network computations across multiple cores.



User Inputs and Configuration

The neural network configuration is determined by user input at runtime:

- **Number of hidden layers**: The user specifies the total number of hidden layers in the network.
- **Number of neurons in each layer**: The user sets the number of neurons in the hidden layer and the output layer. Both layers should have equal number of neurons.

Network weights and input values are read from a file named **input.txt**. Each layer reads the weights corresponding to its neurons and applies them to the input signals it receives from the preceding layer.

Neural Network Architecture

- **Input Layer**

  The input layer initializes the network with raw input values from the file. It contains exactly two neurons, each represented as a separate thread. Each neuron computes its weighted contribution to the next layer using the following formula:

$$neuron\_output = \sum_{n}^{i=1} input_i \text{ x } weight_i$$

  where

  - n = number of inputs to the neuron
  - $input_i$ = input from previous layer
  - $weight_i$ = corresponding weight from file

The outputs from all neurons are aggregated and sent to the subsequent layer process via a pipe as input.

- **Hidden Layers**

Each hidden layer is implemented as a separate process. The neurons within a hidden layer are represented as threads, each calculating their activation based on the input received from the previous layer. The process workflow is as follows:

  - Read input values from the pipe connected to the previous layer.
  - Read the corresponding weights for each neuron from input.txt.
  - Spawn threads to perform the weighted input calculations concurrently.
  - Aggregate the outputs of all neurons and send them through a pipe to the next layer.
  - Illustratively handle backward signals by printing propagated values.

This structure allows multiple neuron threads to execute simultaneously across processor cores, maximizing parallelism.

- **Output Layer**

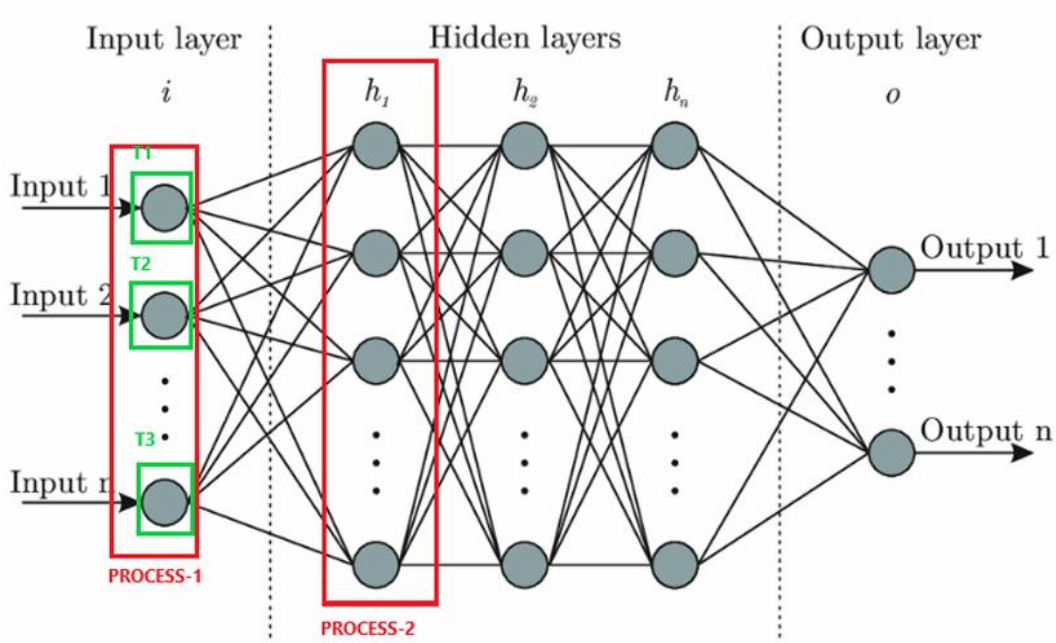
The output layer reads the aggregated inputs from the previous layer and applies weights using threads. After that, it computes the sum of all weighted inputs to generate output. Then the following two formulas are applied to the generated output:

  - $f(x_1) = \frac{output^2 + output + 1}{2}$

  - $f(x_2) = \frac{output^2 - output}{2}$

These outputs are sent backward to the previous layer through the pipe, simulating a basic backward pass. Once the backward pass reaches the input layer, the network performs one additional forward pass, using $f(x_1)$and $f(x_2)$ as new input values for the input neurons.

All output values, including those from both forward passes and the computed $f(x_1)$and $f(x_2)$, are written to output.txt.



Process and Thread Representation

Each layer of the neural network will be represented as a separate process. Each neuron within a layer will be implemented as an individual thread.

Inter-Process Communication (IPC)

The system will employ pipes as the primary mechanism for inter-process communication. Through these pipes, neural network parameters such as inputs, weights and outputs will be exchanged between processes. A batch-based processing approach enables distributed computation, where each process handles a portion of the input batch sequentially through the pipeline.

Concurrency Control and Locking Mechanisms

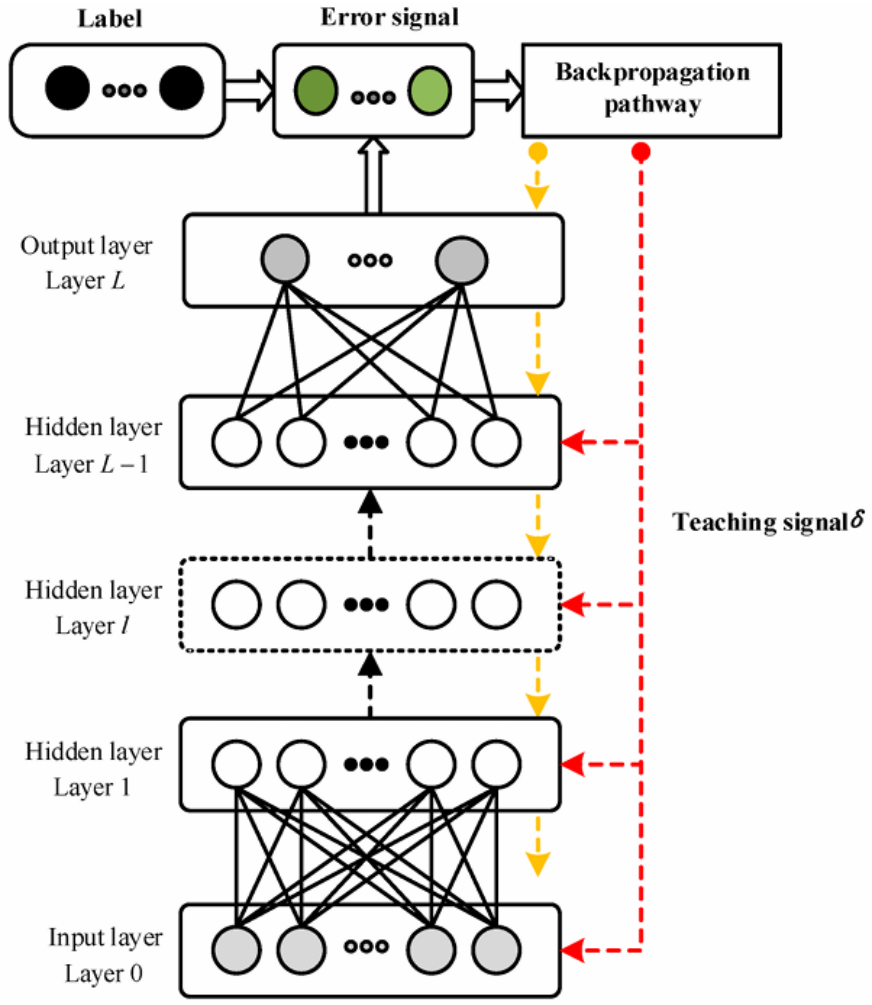
To prevent race conditions and ensure data integrity, the operating system will use locking mechanisms (such as mutexes or semaphores) to control access to shared resources. Only

one thread or process will be allowed to modify shared data (e.g., weights and biases) at any given time.

Backpropagation

No actual weight updates or gradient calculations are performed. The backward signal from the output layer (i.e., $f(x_1)$and $f(x_2)$) is sent back through the layers via pipes for demonstration. Once the backward propagation reaches the input layer, the network performs another forward pass using these values as new inputs.

This simplified mechanism visually represents backpropagation flow without altering any network parameters.



Implementation Details:

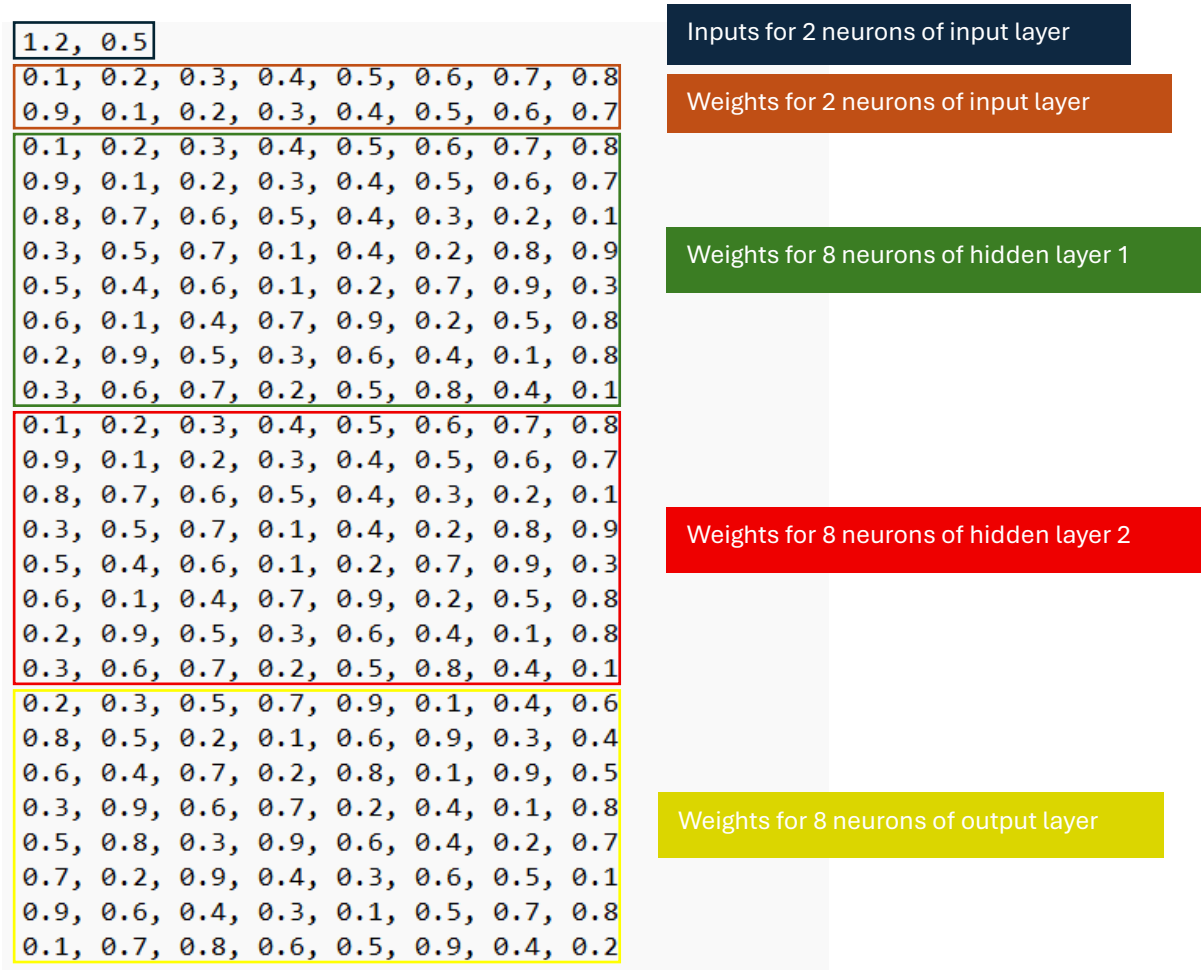
- The program should print appropriate status messages during execution, showing the progress of each layer and the results of both forward passes.
- Your program should not be hardcoded for a fixed number of layers or neurons.
- Inputs and weights for neurons should be read from a file (e.g., file.txt), not hardcoded in the program.
- Output result is stored in an output.txt file.
- Close all files and pipes after use, destroy semaphores/mutexes and remove pipes using unlink() when done.

Input File

A sample input file is given for your help. Your input file should follow the same format as similar input file will be provided to you during demo for testing.

Example Scenario :

- Input layer with 2 neurons
- 2 hidden layers with 8 neurons in each
- Output layer with 8 neurons



Project Rubrics

Component	Marks
Understanding of Problem & QnA (Viva)	50
Clear explanation of process/thread mapping for layers/neurons, benefits of multi-core parallelism.	15
Answers questions accurately about OS-level implementation and OS concepts	35
Design & Architecture	45
Layer-process and neuron-thread mapping	7
Dynamic creation of processes/threads at runtime from user input	15
Inter-process communication design & workflow	10
Resource lifecycle management (pipes creation, closing, unlink, cleanup)	6
Logical data flow correctness (input → hidden → output)	7
Forward Pass Implementation	24
Correct weighted-sum computation for neurons	8
Proper thread synchronization (mutex/semaphore)	8
Aggregation of outputs & communication via IPC	8
Backward Pass Simulation & Visualization	30
Correct computation of $f(x_1)$and $f(x_2)$at output layer	7
Backward propagation via pipes to each layer	8
Display intermediate outputs for each layer during backpropagation	5
Second forward pass using backward outputs as new inputs	10
Inter-Process Communication (IPC)	18
Correct use of pipes for data transfer	6
Reliable, error-free communication between processes	6
Proper cleanup of pipes (unlink) after execution	6
Concurrency Control & Thread Synchronization	18
Proper use of mutexes/semaphores to avoid race conditions	8
Ensuring thread-safe access to shared resources	10

Input/Output Handling	15
Reading weights, and inputs from input.txt	8
Writing all forward/backward outputs to output.txt	7
Testing & Validation	15
Correct execution using provided input file during demo	10
Handling multiple hidden layers / neurons dynamically	5
Clarity of Output & Status Messages	10
Project Report	25
TOTAL	**250**