International Computer Science Competition Qualification 2025 Round Problem Submission
Chi Nguyen (10/10/2006)
Boston University Class of 2028 majoring in Biomedical Engineering
Applicant Senior (current sophmore)

Problem A: Neural Network Components

Consider this illustration of a neural network (Multi-layer Perceptron), that predicts whether a restaurant customer was satisfied with their visit:

- $w_{21}^{(1)}$: **Synaptic weight** of the network: it connects the first input node "Duration stayed" to the second node "Tip received" that tells how strongly the input *"Tip received"* influences one particular neuron. Each weight in a neural network modifies how much influence, or the strength of the connection between two neurons. If the training data result shows that customers who give higher tips are more often satisfied, the model will learn that this input "Tip received" is strongly correlated with satisfaction. As a result, the corresponding weight(s) connected to "Tip received" will become large (positive). The size of the tip indicates whether the customer was satisfied. If customers who stay longer (large duration stayed) report higher satisfaction, the weights connected to "duration stayed" will become large and positive. If staying longer correlates with lower satisfaction (for example, customers maybe waiting too long for food makes them unhappy), the weights might become negative. If Duration Stayed doesn't really affect satisfaction, those weights will stay small (close to zero).

- Σ: **Summation function.** This summation node calculates the weighted sum of inputs by adding up all weighted inputs and bias before applying an activation function, basically combining "Duration stayed" and "Tip received" signals to calculate how much influence they should have.

- f: **Activation function:** It determines how a neuron firing and activating strength that passes the signal to the next layer in the network, took an input 'x' multiplied by the weight 'w' and applied transformation to the weighted sum of inputs in each node. This

Activation function introduces non-linearity (example: sigmoid, ReLU, tanh to compute the neuron's output) so the network can capture and learn all complex relationship patterns after training. Without the Activation function, the neural network could only understand linear patterns. In this specific application, satisfaction is only high if both tip and stay duration are high. Results in high tips and short duration stay explain that customers are somehow satisfied. Low tip and long stay present customers aren't satisfied.

-  **Input layer:** This Red Circle represents features (inputs) from the restaurant visit, which are “Duration stayed” and “Tip received.”

-  **Bias input:** This bias input shifts activation function left or right by adjusting the output along with the weighted sum of the inputs to the neuron. The bias acts like the intercept in linear regression by shifting the curve graph up, down, left, or right. It is always equal to 1 (a constant bias) that connects to each neuron with its own bias weight (labeled as b). It handles the zero inputs by ensuring that a neuron can still produce a non-zero output. With bias: the model can predict satisfaction even if one input is small, because the bias shifts the activation threshold. The effect of shifting the boundary by a constant amount (b * w1) into the mathematical equation present how a

neuron computes: z=w1x1+w2x2+⋯+b..Bias helps the neural network to learn and fit the data, which enhances the network's ability to capture complex relationships.

Final Output neuron: This Green Circle represents the output layer neuron display final prediction result after training and learning. Inputs of "Tip" and "Duration Stayed" information are collected to this Output, so it can takes those signals and decides whether the customers are likely satisfied or not satisfied.

- Box A: **Hidden Layer.** This includes a group of hidden neurons that processes mixes features (“tip” + “duration” information) both tip and stay duration are high indicating satisfaction, high tips and short duration stay explain that customers are somehow satisfied, and low tip and long stay present customers aren't satisfied. In this example application, data is simple and less complex so it involves only 1 hidden layer, presenting fewer dimensions. But for a different application dealing with more complex data with large dimensions or features, 3 more to 5 hidden layers are used.

- Box B: **Output Layer.** It is the final decision layer that translates the hidden layer’s signals to produce the final result: the prediction whether the restaurant visit was satisfying or not.

- ŷ: **Prediction Output probability** of the whole network = probability output presenting (satisfaction level) “satisfied” vs. “not satisfied” for this application. To answer the prediction about whether the customers walk into the restaurant likely satisfied or not satisfied, it makes ŷ (y-hat) a **probability** between 0 and 1. For example, ŷ is a large value closer to 1 like 0.85 indicating 85% chance customers are satisfied. And a small ŷ value of 0.15 indicates 15% chance customers are satisfied indicating they're likely dissatisfied.

Problem B: Cake Calculator

```
#Problem B: Cake Calculator

import sys

def cake_calculator(flour: int, sugar: int) -> list:
    """
    Calculates the maximum number of cakes that can be made and the
leftover ingredients.

    Args:
        flour: An integer larger than 0 specifying the amount of available
flour.
        sugar: An integer larger than 0 specifying the amount of available
sugar.

    Returns:
        A list of three integers:
        [0] the number of cakes that can be made
        [1] the amount of leftover flour
        [2] the amount of leftover sugar

    Raises:
        ValueError: If inputs flour or sugar are not positive.
    """
```

```
# My Code response here:
   # Check if Input valid: Makes sure input value the user enters for
flour and sugar are positive (larger than 0)
   if flour <= 0 or sugar <= 0:      #checks if both flour and sugar
amount is zero or negative
   #only when both flour > 0 and sugar > 0 consider fully satisfies
requirement and program continue
       raise ValueError("Flour and sugar must be positive integers.")

   # Calculate maximum cakes based on each ingredient
   cakes_from_flour = flour // 100 #Finds how many full cakes user could
bake with only 100 units of flour
   cakes_from_sugar = sugar // 50 #Finds how many full cakes you could
bake with only 50 units of sugar

   # Determine number of cakes that can be made
   cake = min(cakes_from_flour, cakes_from_sugar)

   # Calculate leftovers
   remaining_flour = flour - (cake * 100) # amount of leftover flour
   remaining_sugar = sugar - (cake * 50) # amount of leftover sugar

   return [cake, remaining_flour, remaining_sugar]


# --- Main execution block. DO NOT MODIFY  ---
if __name__ == "__main__":
   try:
       # 1. Read input from stdin
       flour_str = input().strip()
       sugar_str = input().strip()

       # 2. Convert inputs to appropriate types
       flour = int(flour_str)
       sugar = int(sugar_str)

       # 3. Call the cake calculator function
       result = cake_calculator(flour, sugar)
```

```
        # 4. Print the result to stdout in the required format
        print(f"{result[0]} {result[1]} {result[2]}")

    except ValueError as e:
        # Handle errors during input conversion or validation
        print(f"Input Error or Validation Failed: {e}", file=sys.stderr)
        sys.exit(1)
    except EOFError:
        # Handle cases where not enough input lines were provided
        print("Error: Not enough input lines provided.", file=sys.stderr)
        sys.exit(1)
    except Exception as e:
        # Catch any other unexpected errors
        print(f"An unexpected error occurred: {e}", file=sys.stderr)
        sys.exit(1)
```

Explanation:

Problem requirements:

- The baker's recipe included:

  - 100 units of flour per cake

  - 50 units of sugar per cake

- The task is to determine:
  - Maximum number of cakes that can be baked

  - Leftovers amount of flour and sugar

- Inputs: flour and sugar (both must be positive integers larger than 0)

- Output: A list of three integers [cakes, remaining_flour, remaining_sugar]

Code Explanation:

1. Function Definition: The function accepts `flour` and `sugar` as integers and return a list of three integers

```
def cake_calculator(flour: int, sugar: int) -> list:
```

2. Check if input for flour and sugar amount valid: Checks that both inputs are positive integers. If either flour or sugar ≤ 0, it raises a ValueError, program will not run

```
if flour <= 0 or sugar <= 0:      #checks if both flour and sugar amount
is zero or negative
   #only when both flour > 0 and sugar > 0 consider fully satisfies
requirement and program continue
       raise ValueError("Flour and sugar must be positive integers.")
```

3. **Calculate Maximum Cakes from Each Ingredient:**

```
  cakes_from_flour = flour // 100 #Finds how many full cakes user could
bake with only 100 units of flour
   cakes_from_sugar = sugar // 50 #Finds how many full cakes you could
bake with only 50 units of sugar
```

If the input value for flour = 250 units, this equation using integer division (//) yields **2**, because 250 ÷ 100 = 2 remainder 50. So with 250 units of flour, we can make 2 full cakes. If input value for sugar is 120, 120 ÷ 50 = 2 remainder 20. So with 120 units of sugar, we can make **2 full cakes**.

4. Determine the Actual Number of Cakes: The `min()` function ensures we don't use more of an ingredient that is available.

```
cake = min(cakes_from_flour, cakes_from_sugar)
```

5. Calculate Leftover Flour and Sugar: It subtracts the amount used in making the cakes to calculate and return leftover flour and sugar. Returns a list `[number_of_cakes, leftover_flour, leftover_sugar]`.

```
remaining_flour = flour - (cake * 100) # amount of leftover flour
   remaining_sugar = sugar - (cake * 50) # amount of leftover sugar

   return [cake, remaining_flour, remaining_sugar]
```

Problem C: The School Messaging App

You and your friends have developed a messaging app that operates over the school network. To transmit messages, the app encodes characters into binary sequences (combinations of zeros and ones). However, due to the school's strict data policy, each student is allocated only a small amount of data per day. To make the most of this limited bandwidth, you aim to implement a more efficient encoding strategy.

Information entropy helps determine the theoretical minimum number of bits needed on average to encode messages. For a set of symbols with probabilities $p_1, p_2, \ldots, p_n$, the entropy H is given by:

$$H = -\sum_{i=1}^{n} p_i \times \log_2(p_i)$$

You know from previous message traffic that characters have the following frequencies:

Character	Probability	Character	Probability
A	0.20	G	0.05
B	0.15	H	0.05
C	0.12	I	0.04
D	0.10	J	0.03
E	0.08	K	0.02
F	0.06	L	0.10

Question 1: Standard text encodings use the same number of bits for each character. Why would using different length codes for characters with different probabilities help you transmit more messages within your data limit? Give an example.

1. Standard encodings (like ASCII) use fixed-length code, use a fixed number of bits (e.g., 8) for every character, which is wasteful for less common and rare symbols when some characters appear far more often than others.
Example: When using Standard (Fixed-length) Encoding, to encode these 12 characters above, every character always takes **4 bits** no matter what, result 12 characters × 4 bits = 48 bits. Meanwhile using Variable-length encoding, the Huffman coding, is more efficient because it assigns shorter codes to more frequent characters and longer codes to less frequent ones. If some characters appear more frequently, it's more efficient to assign them shorter codes and less frequent characters longer codes.This reduces the average number of bits per character, allowing you to transmit more characters within the same amount of data limit. Most frequent characters: A (0.20), B (0.15), C (0.12), Least frequent: K (0.02), J (0.03). We could assign:
A → 0, B → 10, C → 110, …K → 11110 resulting Variable-length: weighted average ≈ 2.8 bits per character → 12 × 2.8 ≈ 34 bits (still smaller than bits of Fixed-length encoding of 48 bitts). This just saves **14 bits** for just 12 characters.

2.Calculate the entropy for the 12-character set above. What does this value represent in terms of optimal encoding?

Compute $-p_i \log_2(p_i)$ for each character

A: $-0.20\log_2(0.20) = -0.20 \times (-2.3219) \approx 0.4644$

B: $-0.15\log_2(0.15) = -0.15 \times (-2.73697) \approx 0.4105$

C: $-0.12\log_2(0.12) = -0.12 \times (-3.05889) \approx 0.3671$

D: $-0.10\log_2 0.10 = -0.10 \times (-3.3219) \approx 0.3322$

E: $-0.08\log_2 0.08 = -0.08 \times (-3.64386) \approx 0.2915$

F: $-0.06\log_2 0.06 = -0.06 \times (-4.05889) \approx 0.2435$

G: $-0.05\log_2 0.05 = -0.05 \times (-4.3219) \approx 0.2161$

H: $-0.05\log_2 0.05 \approx 0.2161$

I: $-0.04\log_2 0.04 = -0.04 \times (-4.64386) \approx 0.1858$

J: $-0.03\log_2 0.03 = -0.03 \times (-5.05889) \approx 0.1518$

K: $-0.02\log_2 0.02 = -0.02 \times (-5.64386) \approx 0.1129$

L: $-0.10\log_2 0.10 \approx 0.3322$

Sum all terms

$H \approx 0.4644+0.4105+0.3671+0.3322+0.2915+0.2435+0.2161+0.2161+0.1858+0.1518+0.1129+0.3$

3.22 bits per character

Explanation: This entropy value H ≈ 3.22 bits per character presents the theoretical minimum average number of bits needed to encode each character. Using an variable-length encoding, you cannot encode the messages with fewer than ~3.32 bits per character on average. By compare, a fixed-length encoding for 12 characters would require 4 bits per character, so you can save data by using a Variable-length code with around 3.32 bits per character less than 4 bits of fixed-length encoding

The Fano method is one way for creating efficient variable-length prefix codes.[2] You have implemented this algorithm and generated the following code for your character set:

Character	Probability	Fano Code	Character	Probability	Fano Code
A	0.20	000	G	0.05	001
B	0.15	100	H	0.05	1011
C	0.12	010	I	0.04	0111
D	0.10	1100	J	0.03	1101
E	0.08	0110	K	0.02	1111
F	0.06	1010	L	0.10	1110

Question 3: Calculate the average code length of your Fano code and compare it to the theoretical entropy limit. How efficient is your Fano code?

Average code length L formula:

pi = probability of character i
li= length of its code

$$L = \sum_{i=1}^{n} p_i \cdot l_i$$

Compute pi x li for each character

A: 0.20 × 3 = 0.60

B: 0.15 × 3 = 0.45

C: 0.12 × 3 = 0.36

D: 0.10 × 4 = 0.40

E: 0.08 × 4 = 0.32

F: 0.06 × 4 = 0.24

G: 0.05 × 3 = 0.15

H: 0.05 × 4 = 0.20

I: 0.04 × 4 = 0.16

J: 0.03 × 4 = 0.12

K: 0.02 × 4 = 0.08

L: 0.10 × 4 = 0.40

Sum all

- 0.60 + 0.45 = 1.05

- +0.36 = 1.41
- +0.40 = 1.81
- +0.32 = 2.13
- +0.24 = 2.37
- +0.15 = 2.52
- +0.20 = 2.72
- +0.16 = 2.88
- +0.12 = 3.00
- +0.08 = 3.08
- +0.40 = 3.48

Average code length ≈3.48 bits per character

Compare to entropy

- Entropy H≈3.32 bits per character
- Fano code average length Lavg ≈ 3.48 bits per character

Efficiency = Entropy H/Code average length Lavg= 3.32/3.48=0.95%≈95.4%

In conclusion, the Fano Code is really efficient, only slightly above the Entropy value. Each character uses 3.48 bits instead of the minimum 3.32 bits, saving a lot of data compared to fixed-length codes (4 bits per character).

Problem D: : Word Search Puzzle: My CODE here

```
import sys

import random

import string



# --- Step 1: Generate the crossword ---

def create_crossword(words: list) -> list:

    size = 10

    grid = [['' for _ in range(size)] for _ in range(size)]



    directions = [

        (0, 1), (0, -1), (1, 0), (-1, 0),

        (1, 1), (1, -1), (-1, 1), (-1, -1)

    ]



    # Place longest words first

    words_sorted = sorted(words, key=lambda w: -len(w))



    for word in words_sorted:

        word_upper = word.upper()

        placed = False

        attempts = 0
```

```
        while not placed:

            attempts += 1

            if attempts > 1000:

                raise Exception(f"Cannot place word '{word_upper}' in the
grid.")



            x = random.randint(0, size - 1)

            y = random.randint(0, size - 1)

            dx, dy = random.choice(directions)



            end_x = x + dx * (len(word_upper) - 1)

            end_y = y + dy * (len(word_upper) - 1)

            if not (0 <= end_x < size and 0 <= end_y < size):

                continue



            conflict = any(

                grid[x + dx * i][y + dy * i] not in ('', word_upper[i])

                for i in range(len(word_upper))

            )



            if not conflict:

                for i in range(len(word_upper)):

                    nx = x + dx * i

                    ny = y + dy * i
```

```
                    grid[nx][ny] = word_upper[i]
                placed = True

    # Fill empty cells with random letters
    for i in range(size):
        for j in range(size):
            if grid[i][j] == '':
                grid[i][j] = random.choice(string.ascii_uppercase)

    return grid

# --- Step 2: Find words in the grid ---
def find_words(grid, words):
    directions = [
        (0, 1), (0, -1), (1, 0), (-1, 0),
        (1, 1), (1, -1), (-1, 1), (-1, -1)
    ]
    found_words = {}
    for word in words:
        w = word.upper()
        for r in range(len(grid)):
            for c in range(len(grid[0])):
                for dx, dy in directions:
```

```
                    if all(
                        0 <= r+i*dx < len(grid) and
                        0 <= c+i*dy < len(grid[0]) and
                        grid[r+i*dx][c+i*dy] == w[i]
                        for i in range(len(w))
                    ):
                        found_words[word] = (r, c, dx, dy)
                        break
                if word in found_words:
                    break
            if word in found_words:
                break
    return found_words


# --- Main execution block. DO NOT MODIFY. ---
if __name__ == "__main__":
    try:
        # Read words from first line (comma-separated)
        words_input = input("Enter words separated by commas: ").strip()
        words = [word.strip() for word in words_input.split(',')]


        # Generate the word search puzzle
        puzzle = create_crossword(words)
```

```
        # Print the puzzle
        print("\n--- Word Search Puzzle ---")
        for row in puzzle:
            print(''.join(row))

        # Find and display word positions
        locations = find_words(puzzle, words)
        print("\n--- Word Locations ---")
        for word, (r, c, dx, dy) in locations.items():
            print(f"{word.upper()}: Start=({r},{c}),
Direction=({dx},{dy})")

    except ValueError as e:
        print(f"Input Error: {e}", file=sys.stderr)
        sys.exit(1)
    except EOFError:
        print("Error: Not enough input lines provided.", file=sys.stderr)
        sys.exit(1)
    except Exception as e:
        print(f"An unexpected error occurred: {e}", file=sys.stderr)
        sys.exit(1)
```

Output:

```
Enter words separated by commas: learning,science,fun

--- Word Search Puzzle ---
SNGIFUNXQQ
FUMLMRKEHF
CBYEJRFFNS
EBBAGNXZLK
CVVRDPOOMW
NKQNFLZWFL
EHFIBDTBKX
IVGNNMWYJS
CCHGJYHZSZ
SYOIXKZGKU

--- Word Locations ---
LEARNING: Start=(1,3), Direction=(1,0)
SCIENCE: Start=(9,0), Direction=(-1,0)
FUN: Start=(0,4), Direction=(0,1)
```

How it works explanation:

1. Generates a 10×10 word search puzzle: The `create_crossword` function places all input words in a grid of size 10×10.

2. Places words in random directions: Words can appear horizontally, vertically, or diagonally (all 8 possible directions) and longest words are placed first to improve placement success.

3. Checking conflicts requirements while placing words: To ensure letters do not overwrite existing letters unless they match the word being placed.

4. Fills remaining empty cells with random letters: Uses uppercase letters to complete the grid to make real word search puzzle.

5. Finds and reports exact word locations: The `find_words` function searches for all words in the grid and the output returns the starting row, column, and direction vector for each word

Problem E:

Problem E: Functional Completeness of NAND ★★★

Logic gates are a fundamental building blocks of computers, as they control how binary information is transmitted through a circuit. A logic gate performs a binary function:

$$f(a, b) \to \{0, 1\}, \quad \text{where } a, b \in \{0, 1\}.$$

Some gates are particularly important due to a property known as *functional completeness*. A logic gate (or set of gates) is said to be functionally complete if any Boolean function can be constructed using only that gate (or gates).

One such gate is the NAND (NOT AND) gate, which outputs 0 only when $a = b = 1$. Thus, it outputs 1 if at least one input is 0.

Prove that the NAND gate is functionally complete.

Hint: Can you express the AND, OR, and NOT operations using only NAND?

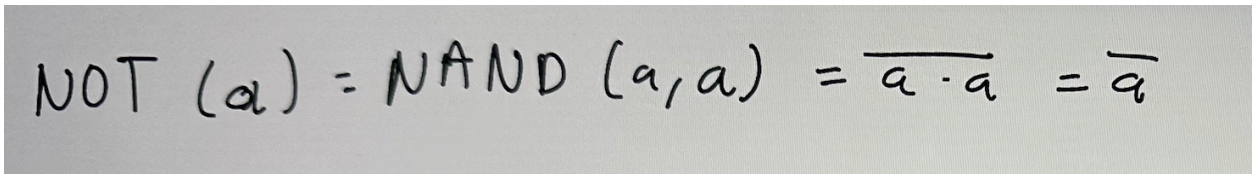
5

We known that by the given information, NAND gate outputs 0 only when both inputs are 1. A set of gates is functionally complete if it can implement any Boolean function.The set {AND, OR, NOT} is functionally complete.

Proof Process that the NAND gate is functionally complete

Express NOT using NAND

- To invert a single input a , just connect both inputs of NAND to a :



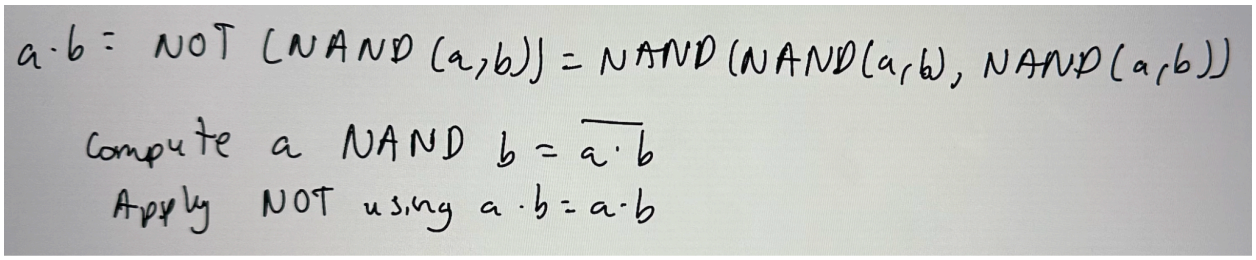
NOT(A) can be implemented as NAND(A, A).

- If A = 0, NAND(0, 0) = 1.
- If A = 1, NAND(1, 1) = 0.

Therefore, NOT(A) = A NAND A.

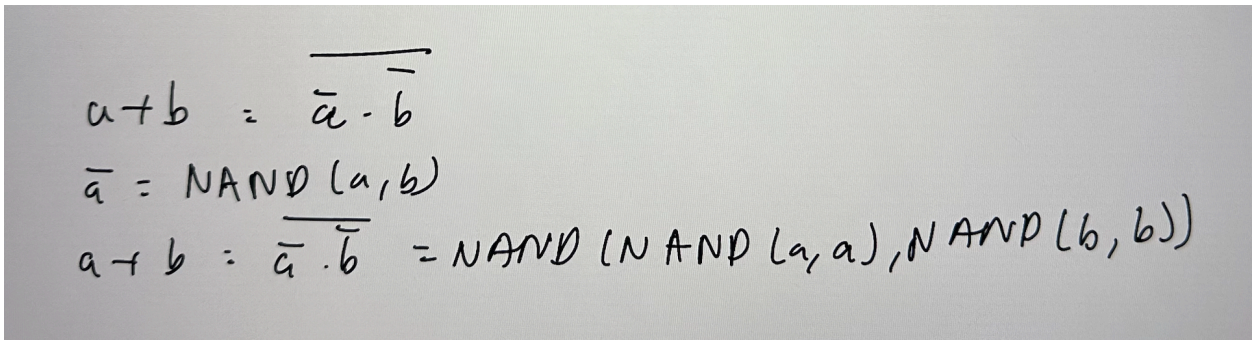
Express AND using NAND

AND(A, B) can be referred to as NOT(NAND(A, B)). Since NOT can be implemented using NAND (from Step 1), AND can be implemented using only NAND gates.



Express OR using NAND

OR(A, B) can be implemented using DeMorgan's Law: A + B = NOT(NOT(A) NAND NOT(B)). Since NOT and NAND can be implemented using only NAND gates, OR can also be implemented using only NAND gates.



Conclusion

- We can construct NOT, AND, OR using only NAND gates. Since any Boolean function can be expressed using AND, OR, and NOT, it proves that any Boolean function can also be expressed using only NAND gates.NAND is functionally complete