Fuzzy

Fuzzy

Defn: A fuzzy set is a set which allow partial belonging to a set; that is defined by a degree of membership denoted by 'μ', that can take any value from 0 to 1.

Crisp set: If we remove all the values of belonging except 0 & 1 the set will collapses into a crisp set.

Membership function: The membership function of the set is the relationship between the elements of the set & their degree of belonging.

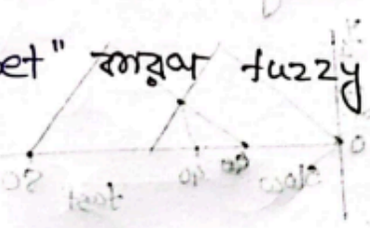
(2,0) . (2,0.1)

Example of Crisp set:

F = {i| i is an integer and $4 \leq i \leq 12$

if i = 4 to 12 then it will be 1 otherwise 0

"Fuzzy is one kind of crisp set" মানে fuzzy set-এ 0, 1 ও combination ও থাকে তাই।



Integer-43

a) List the differences between Crisp Set and Fuzzy Set.

Defination:

Crisp set: A crisp set is a collection of discrete value.

Fuzzy set: 'Already given'

Membership:

Crisp set: Elements either belong to the set (membership value 1) or do not belong which m. value is 0.

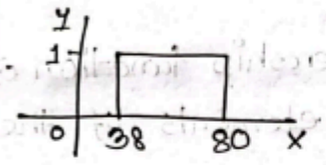


Fig: Crisp set

So if x is less than 38 or 80 < x then it will be zero. x will be 1 if it remains between 38 & 80.

More precisely, A set of red apples {apples, not apples}

Fuzzy: Elements have degree of membership between 0 and 1, indicating the extent to which they belong to the set.

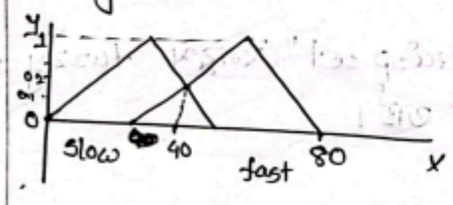


Fig: fuzzy set

So, here starting from zero it gradually speed up not directly goes to 1. For 40, there is an equal possibility of being slow or fast. It's a continuos value.

Example: The set of ripe apple, where the degree of membership varies for each apple based on its ripeness.

Boundary:

Crisp set has a clear, well defined boundary but fuzzy set lacks a precise boundary and elements can have partial membership.

S.No	Crisp Set	Fuzzy Set
1	Crisp set defines the value is either 0 or 1.	Fuzzy set defines the value between 0 and 1 including both 0 and 1.
2	It is also called a classical set.	It specifies the degree to which something is true.
3	It shows full membership	It shows partial membership.
4	Eg1. She is 18 years old. Eg2. Rahul is 1.6m tall	Eg1. She is about 18 years old. Eg2. Rahul is about 1.6m tall.
5	Crisp set application used for digital design.	Fuzzy set used in the fuzzy controller.
6	It is bi-valued function logic.	It is infinite valued function logic
7	Full membership means totally true/false, yes/no, 0/1.	Partial membership means true to false, yes to no, 0 to 1.

Advantages and Disadvantages of Fuzzy Logic System:

Advantages:

Tolerance for Imprecise Data: Fuzzy logic is good at handling imprecise, noisy, or incomplete data, making it valuable in real-world applications where such data are common.

Modeling Human Reasoning: It mimics human decision-making logic, which is not always black or white but often involves shades of gray. This makes fuzzy logic systems more intuitive and effective for certain types of decision-making processes.

Flexibility: Fuzzy logic can be combined with other methodologies, such as neural networks (forming neuro fuzzy systems), to enhance their capabilities, particularly in learning and adapting to new data.

Simplicity: In many cases, fuzzy logic can simplify complex problems, reducing the need for precise mathematical formulations. This can make the development of control and decision-making systems more straightforward.

Disadvantages:

Computational Complexity: The process of converting input data into fuzzy values, processing them through fuzzy rules, and then defuzzifying the results can be computationally intensive, especially for systems with a large number of rules or variables.

Difficulty in Designing Fuzzy Rules: The effectiveness of a fuzzy logic system heavily depends on the quality and comprehensiveness of the fuzzy rules set. Designing these rules requires expertise and a deep understanding of the application domain, which can be challenging.

Lack of Standardization: There are no universally accepted standards for the design and implementation of fuzzy logic systems, which can lead to inconsistency and difficulty in comparing different systems or transferring knowledge between them.

Subjectivity in Membership Functions: The design of membership functions (which define how input values are mapped to fuzzy values) is somewhat subjective and relies on the designer's intuition and experience. This subjectivity can lead to variability in system performance.

Fuzzy Logic: It's like a smart system that can handle uncertainty and make decisions based on "maybe" instead of just "yes" or "no." It works well when things aren't black or white.

Neurofuzzy System: Imagine combining that smart system (fuzzy logic) with a learning brain (neural network) that gets smarter over time. It not only handles uncertainty but also learns from new information to make even better decisions.

So, while fuzzy logic deals with uncertainty, a neurofuzzy system learns from it and adapts, making it a smarter, self-improving system.

Importance of Neuro Fuzzy System:

Make Better Decisions with Unclear Data: They are really good at making decisions when the information isn't clear or complete, much like how humans do, by understanding shades of meanings rather than just yes or no.

Learn and Adapt: These systems learn from experiences and can adjust to new situations on their own. This means they get better over time at whatever they're doing, from recognizing patterns to predicting outcomes.

Solve Complex Problems: They can handle very complicated issues, including those that change in unpredictable ways, by combining the strengths of fuzzy logic and neural networks.

Better Control for Machines and Systems: They offer more precise control in situations that are uncertain or keep changing, making them ideal for things like cars, planes, and factories.

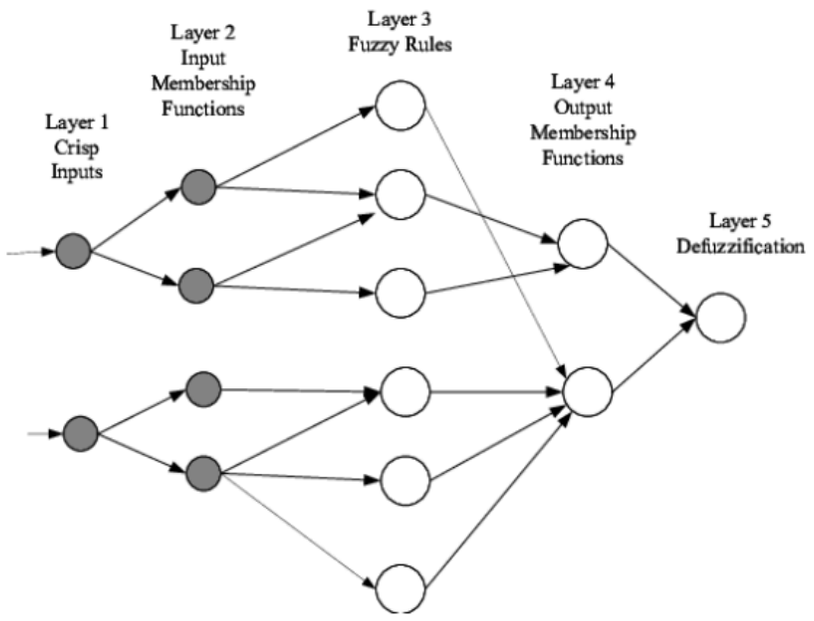
Fit to Your Needs and Grow: They can be tailored to specific tasks and are capable of dealing with larger and more complex situations as needed, making them versatile for a wide range of uses.

How can a neuro fuzzy system be built?

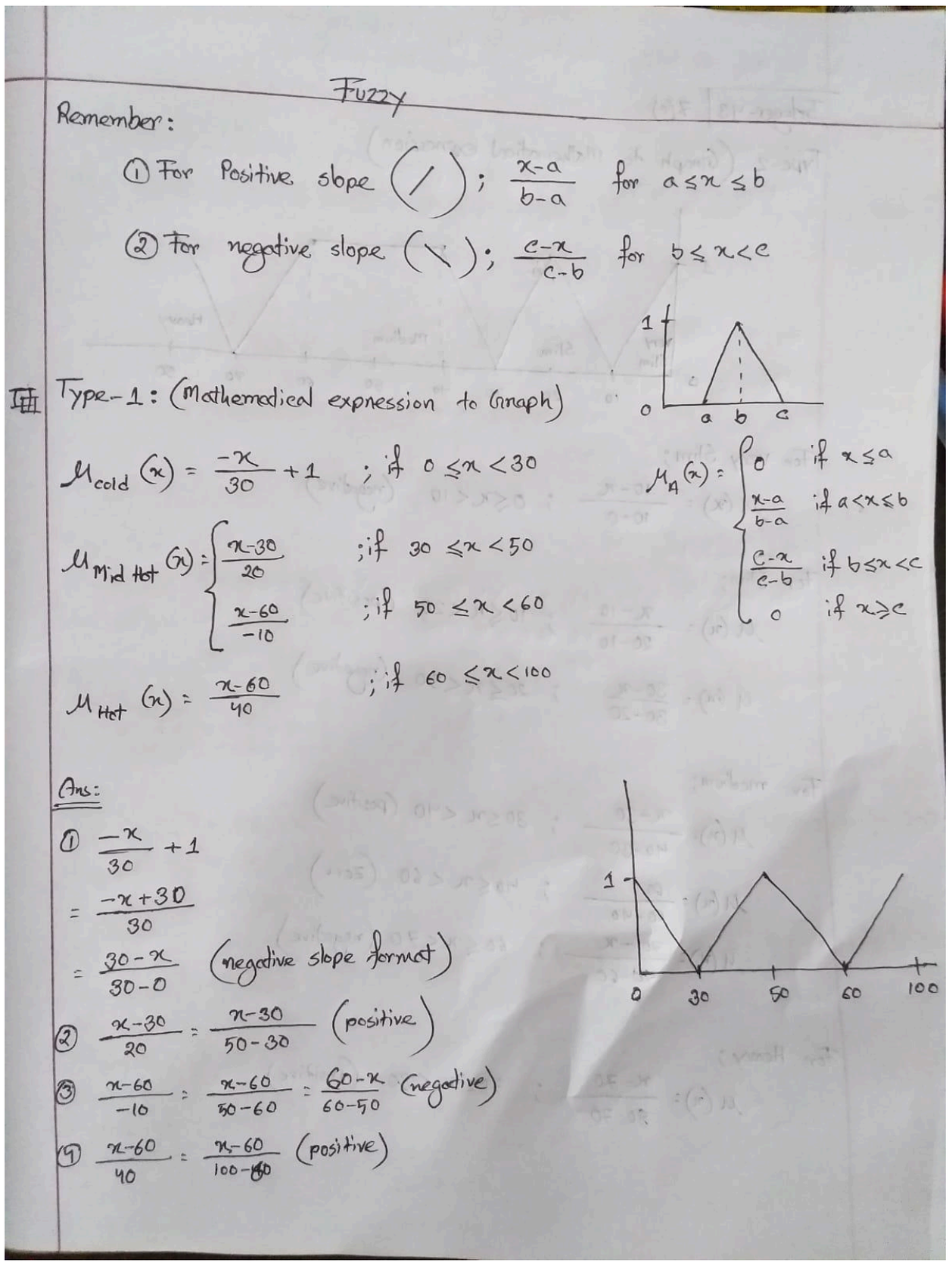
Neuro Fuzzy System:

- A neuro-fuzzy system is based on a fuzzy system which is trained by a learning algorithm derived from neural network theory.
- A neuro-fuzzy system can be built as a 3-layer feedforward neural network.
  - First layer represents input variables
  - Middle (hidden) layer represents fuzzy rules
  - Third layer represents output variables

  Fuzzy sets are encoded as (fuzzy) connection weights. It represents the data flow of input processing and learning within the model. Sometimes a 5-layer architecture is used, where the fuzzy sets are represented in the units of the second and fourth layer.
- A neuro-fuzzy system can be always interpreted as a system of fuzzy rules. It is also possible to create the system out of training data from scratch, as it is possible to initialize it by prior knowledge in the form of fuzzy rules.

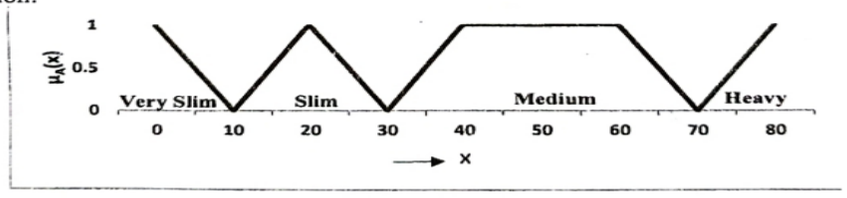


Graph math:


Fuzzy
Remember:
① For Positive slope (/); $\frac{x-a}{b-a}$ for $a \le x \le b$
② For negative slope (\); $\frac{c-x}{c-b}$ for $b \le x < c$
Type-1: (Mathemedical expression to Graph)
$\mu_{cold}(x) = \frac{-x}{30} + 1$; if $0 \le x < 30$
$\mu_{Mid\ Hot}(x) = \begin{cases} \frac{x-30}{20} & ; \text{if } 30 \le x < 50 \\ \frac{x-60}{-10} & ; \text{if } 50 \le x < 60 \end{cases}$
$\mu_{Hot}(x) = \frac{x-60}{40}$; if $60 \le x < 100$
$\mu_A(x) = \begin{cases} 0 & \text{if } x \le a \\ \frac{x-a}{b-a} & \text{if } a < x \le b \\ \frac{c-x}{c-b} & \text{if } b \le x < c \\ 0 & \text{if } x \ge c \end{cases}$
1
0
a b c
Ans:
① $\frac{-x}{30} + 1$
$= \frac{-x+30}{30}$
$= \frac{30-x}{30-0}$ (negative slope format)
② $\frac{x-30}{20} = \frac{x-30}{50-30}$ (positive)
③ $\frac{x-60}{-10} = \frac{x-60}{50-60} = \frac{60-x}{60-50}$ (negative)
④ $\frac{x-60}{40} = \frac{x-60}{100-60}$ (positive)
1
0 30 50 60 100

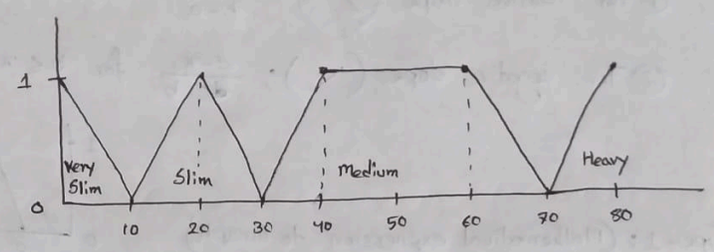
Integer-43

Assume you want to calculate the health of a person based on his/her weight. A graphical representation of the membership function for this task is given in the following figure. Analyze the graph and write down the mathematical expression of the membership function. [7]



Integer-43 7(c)

Type-2 (Graph to Mathematical expression)



For very Slim;

$$\mu(x) = \frac{10-x}{10-0} \quad ; \quad 0 \leq x < 10 \quad \text{(negative)}$$

For Slim;

$$\mu(x) = \frac{x-10}{20-10} \quad ; \quad 10 \leq x < 20 \quad \text{(positive)}$$

$$\mu(x) = \frac{30-x}{30-20} \quad ; \quad 20 \leq x < 30 \quad \text{(negative)}$$

For medium;

$$\mu(x) = \frac{x-30}{40-30} \quad ; \quad 30 \leq x < 40 \quad \text{(positive)}$$

$$\mu(x) = 1 \quad ; \quad 40 \leq x < 60 \quad \text{(zero)}$$

$$\mu(x) = \frac{70-x}{70-60} \quad ; \quad 60 \leq x < 70 \quad \text{(negative)}$$

For Heavy;

$$\mu(x) = \frac{x-70}{80-70} \quad ; \quad 70 \leq x < 80 \quad \text{(positive)}$$

Integer-43

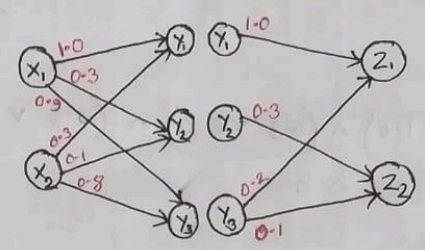
b) Illustrate max-average composition of fuzzy relation with a suitable example.

Remember: $\wedge$ = min

$\vee$ = max

Max - Average:

The composition of fuzzy relations is an operation that combines two fuzzy relations to obtain a new fuzzy relation.



Here, two fuzzy relations can be observed seperately, one is in between x and y, other one y and z. Now we will apply Max-average composition to obtain a fuzzy relation between x and z.

R_1:

	Y_1	Y_2	Y_3
x_1	1·0	0·3	0·9
x_2	0·3	0·1	0·8

R_2:

	Z_1	Z_2
Y_1	1·0	0·0
Y_2	0·0	0·3
Y_3	0·2	0·1

Final matrix:

	Z_1	Z_2
x_1	1·0	0·5
x_2	0·5	0·45

$\mu_{R_1}(x_1, Y_1) + \mu_{R_2}(Y_1, Z_1) = 1·0 + 1·0 = 2$

$\mu_{R_1}(x_1, Y_2) + \mu_{R_2}(Y_2, Z_1) = 0·3 + 0·0 = 0·3$

$\mu_{R_1}(x_1, Y_3) + \mu_{R_2}(Y_3, Z_1) = 0·9 + 0·2 = 1·1$

$\Rightarrow \mu_{R_1 \leftrightarrow R_2}(x_1, Z_1) = \frac{1}{2}(2 \vee 0·3 \vee 1·1) = 1·0$

$\mu_{R_1}(x_1, Y_1) + \mu_{R_2}(Y_1, Z_2) = 1·0 + 0·0 = 1·0$

$\mu_{R_1}(x_1, Y_2) + \mu_{R_2}(Y_2, Z_2) = 0·3 + 0·3 = 0·6$

$\mu_{R_1}(x_1, Y_3) + \mu_{R_2}(Y_3, Z_3) = 0·9 + 0·1 = 1·0$

$\Rightarrow \mu_{R_1 \leftrightarrow R_2}(x_1, Z_2) = \frac{1}{2}(1 \vee 0·6 \vee 1) = 0·5$

$\mu_{R_1}(x_2, Y_1) + \mu_{R_2}(Y_1, Z_1) = 0·3 + 0·1 = 0·4$

$\mu_{R_1}(x_2, Y_2) + \mu_{R_2}(Y_2, Z_1) = 0·1 + 0·0 = 0·1$

$\mu_{R_1}(x_2, Y_3) + \mu_{R_2}(Y_3, Z_1) = 0·8 + 0·2 = 1·0$

$\Rightarrow \mu_{R_1 \leftrightarrow R_2}(x_2, Z_1) = \frac{1}{2}(0·4 \vee 0·1 \vee 1·0) = 0·5$

$\mu_{R_1}(x_2, Y_1) + \mu_{R_2}(Y_1, Z_2) = 0·3 + 0·0 = 0·3$

$\mu_{R_1}(x_2, Y_2) + \mu_{R_2}(Y_2, Z_2) = 0·1 + 0·3 = 0·4$

$\mu_{R_1}(x_2, Y_3) + \mu_{R_3}(Y_3, Z_2) = 0·8 + 0·1 = 0·9$

$\Rightarrow \mu_{R_1 \leftrightarrow R_2}(x_2, Z_2) = \frac{1}{2}(0·3 \vee 0·4 \vee 0·9) = 0·45$

* If the question's structure, changes do the following ~~it~~ processes for the respective questions.

For max-min:

$$\mu_{R_1 \circ R_2}(x_1, z_1) = (1.0 \wedge 1.0) \vee (0.3 \wedge 0.0) \vee (0.9 \wedge 0.2)$$
$$= 1.0 \vee 0.0 \vee 0.2$$
$$= 1.0$$

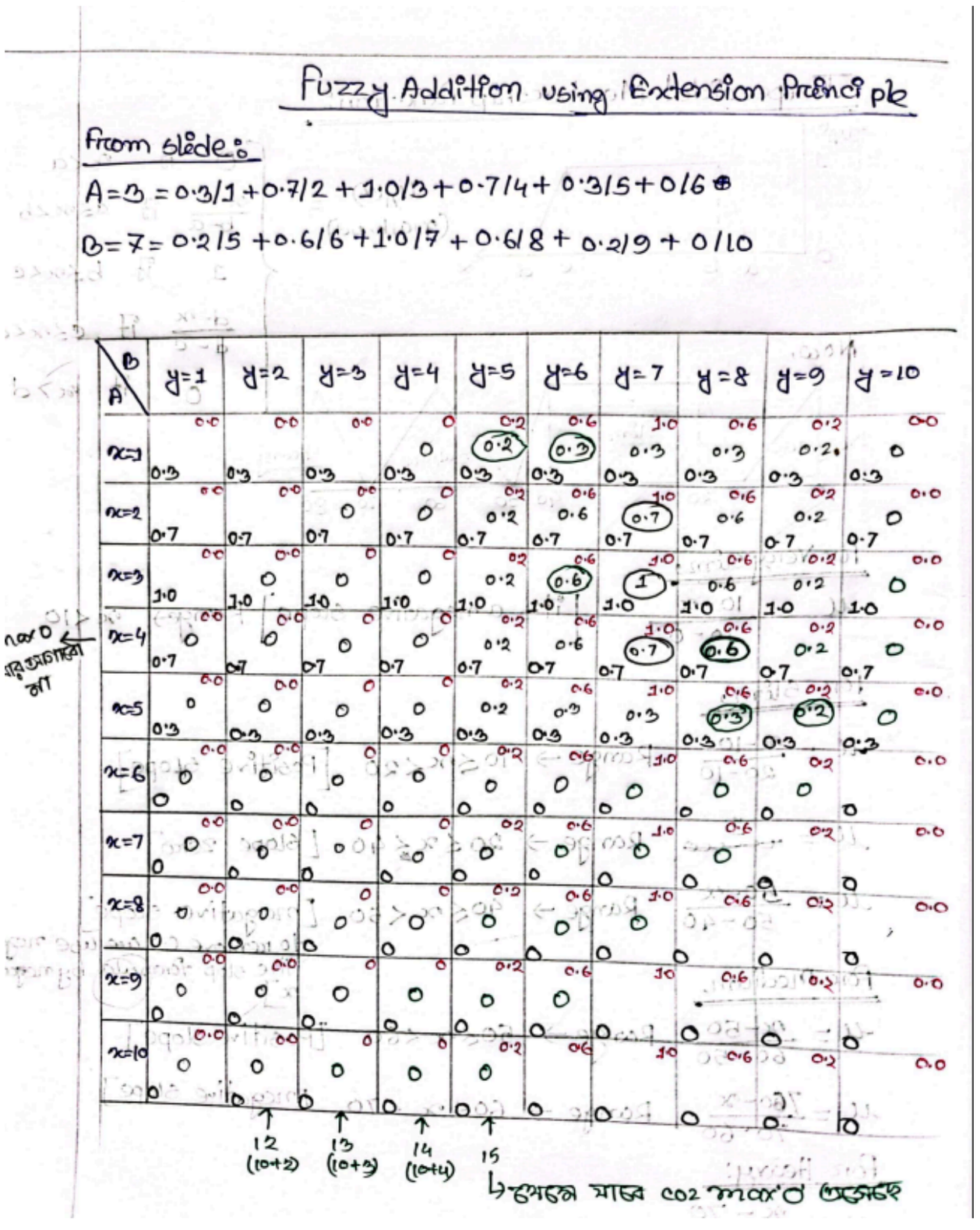
For min-max:

$$\mu_{R_1 \circ R_2}(x_1, z_1) = (1.0 \vee 1.0) \wedge (0.3 \vee 0.0) \wedge (0.9 \vee 0.2)$$
$$= 1.0 \wedge 0.3 \wedge 0.9$$
$$= 0.3$$

Max-product:

$$\mu_{R_1 \circ R_2}(x_1, z_1) = (1.0 \times 1.0) \vee (0.3 \times 0.0) \vee (0.9 \times 0.2)$$
$$= 1.0 \vee 0.0 \vee 0.18$$
$$= 1.0$$

Extension Principle math:



diagonally check করুব(xy) র value কত আছে।

Suppose for addition of 5

(x=4, + y=1), (x=3 + y=2) (x=2 + y=3) (x=1 + y=4)

it creates a diagonal from (x=4 to y=4)

** similarly for 6 → (x=5 to y=5) ; 7 → (x=6 to y=6)

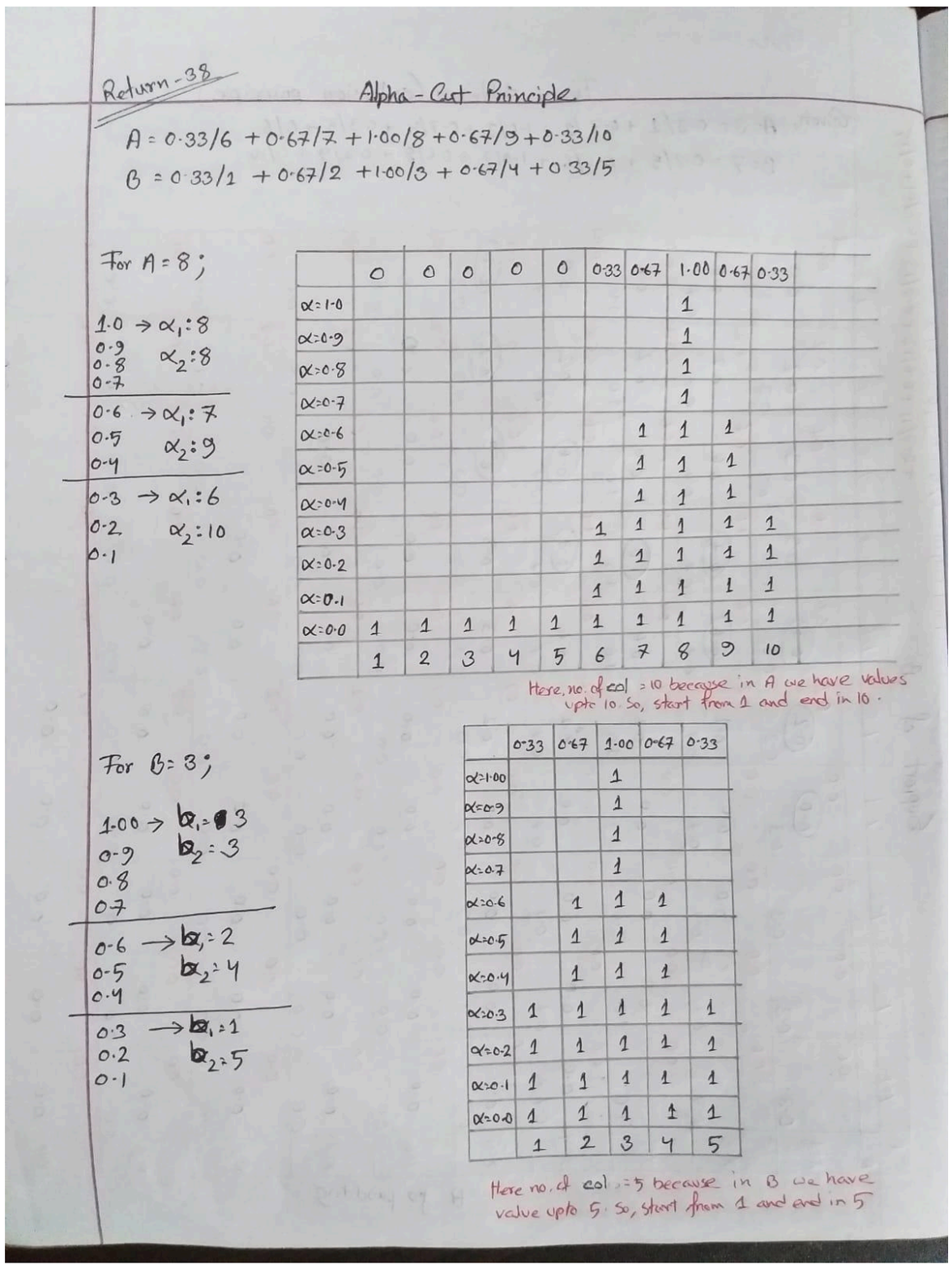
Output of C

= 0/5 + 0.2/6 + 0.3/7 + 0.6/8 + 0.7/9 + 1.0/10 + 0.7/11
+ 0.6/12 + 0.3/13 + 0.2/14 + 0/15

For addition of 12 → (x=10, y=2) - এই box থেকে diagonally আগাবো। same for 13 → (x=10 + y=3) ....
& rest of all.

Alpha cut:

Return-38



Return-38

Alpha-Cut Principle.

A = 0.33/6 + 0.67/7 + 1.00/8 + 0.67/9 + 0.33/10

B = 0.33/1 + 0.67/2 + 1.00/3 + 0.67/4 + 0.33/5

For A = 8;

1.0 → α_1 : 8
0.9 α_2 : 8
0.8
0.7

0.6 → α_1 : 7
0.5 α_2 : 9
0.4

0.3 → α_1 : 6
0.2 α_2 : 10
0.1

	0	0	0	0	0	0.33	0.67	1.00	0.67	0.33
α = 1.0								1		
α = 0.9								1		
α = 0.8								1		
α = 0.7								1		
α = 0.6							1	1	1	
α = 0.5							1	1	1	
α = 0.4							1	1	1	
α = 0.3						1	1	1	1	1
α = 0.2						1	1	1	1	1
α = 0.1						1	1	1	1	1
α = 0.0	1	1	1	1	1	1	1	1	1	1
	1	2	3	4	5	6	7	8	9	10

Here, no. of col = 10 because in A we have values upto 10. So, start from 1 and end in 10.

For B = 3;

1.00 → α_1 : 3
0.9 α_2 : 3
0.8
0.7

0.6 → α_1 : 2
0.5 α_2 : 4
0.4

0.3 → α_1 : 1
0.2 α_2 : 5
0.1

	0.33	0.67	1.00	0.67	0.33
α = 1.00			1		
α = 0.9			1		
α = 0.8			1		
α = 0.7			1		
α = 0.6		1	1	1	
α = 0.5		1	1	1	
α = 0.4		1	1	1	
α = 0.3	1	1	1	1	1
α = 0.2	1	1	1	1	1
α = 0.1	1	1	1	1	1
α = 0.0	1	1	1	1	1
	1	2	3	4	5

Here no. of col = 5 because in B we have value upto 5. So, start from 1 and end in 5

α-cut:

For 0·1; $C_{0.1} = (6,10) + (1,5) = (7,15)$

0·2; $C_{0.2} = (6,10) + (1,5) = (7,15)$

0·3; $C_{0.3} = (6,10) + (1,5) = (7,15)$

0·4; $C_{0.4} = (7,9) + (2,4) = (9,13)$

0·5; $C_{0.5} = (7,9) + (2,4) = (9,13)$

0·6; $C_{0.6} = (7,9) + (2,4) = (9,13)$

0·7, 0·8, 0·9, 1·0; $C_{0.7}, C_{0.8}, C_{0.9}, C_{1.0} = (8,8) + (3,3) = (11,11)$

							0·3	0·3	0·6	0·6	1·0	0·6	0·6	0·3	0·3
α=1·0											1				
α=0·9											1				
α=0·8											1				
α=0·7											1				
α=0·6									1	1	1	1	1		
α=0·5									1	1	1	1	1		
α=0·4									1	1	1	1	1		
α=0·3							1	1	1	1	1	1	1	1	1
α=0·2							1	1	1	1	1	1	1	1	1
α=0·1							1	1	1	1	1	1	1	1	1
α=0·0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

$C = 0/6 + 0.3/7 + 0.3/8 + 0.6/9 + 0.6/10 + 1/11 + 0.6/12 + 0.6/13 + 0.3/14 + 0.3/15$

GA and ACO

Integer-43 (3b)

Population:

R_1 = | 1 | 3 | 5 | 7 | 8 | 4 | 6 | 2 |

R_2 = | 3 | 4 | 1 | 5 | 2 | 7 | 6 | 8 |

Crossover:

R_1 = | 1 | 3 | 5 | 7 | 8 | 4 | 6 | 2 |

R_2 = | 3 | 4 | 1 | 5 | 2 | 7 | 6 | 8 |

$C1$ = | 3 | 1 | 5 | 7 | 8 | 4 | 2 | 6 |

In this crossover, we select the genes of one parent and maintain the order of the genes in child. The rest are taken from another parent and excluding the genes of Parent 1, the rest are filled from parent 2.

Davis order crossover

Mutation:

Before:

$C1$ = | 3 | 1 | 5 | 7 | 8 | 4 | 2 | 6 |

After:

$C1$ = | 3 | 2 | 5 | 7 | 8 | 4 | 1 | 6 |

(Swap mutation)

is our required single offspring.

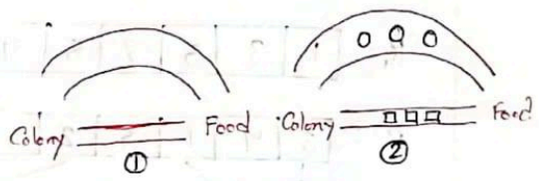
Integer-43 (6b)

Q: ACO algo with proper example:

Ans:

Initialization: The pheromone trails are initialized to a small positive value for all edges, and each ant is placed on a starting node. In the diagram, There are two paths to the food source, each with equal amount of pheromone.

Path Construction: Ants choose paths probabilistically, the path with higher pheromone will be chosen. For our example and as currently the amount of pheromone is same in all the path, Some ant took th higher path and some took the lower path.

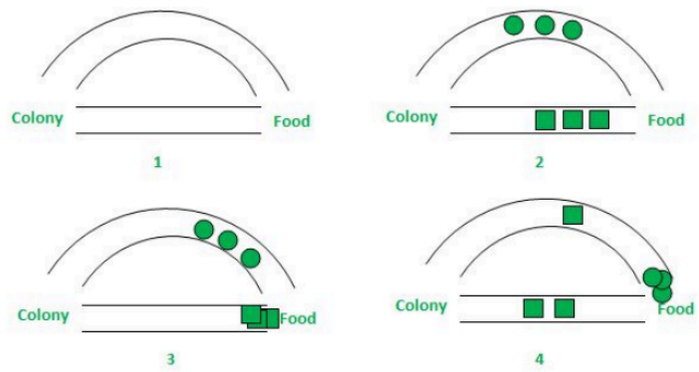


pheromone update: Two components –

- Pheromone Evaporation: To simulate the natural evaporation of pheromone over time, reducing the intensity of all pheromone trails.
- Pheromone Deposition – Ants deposit pheromone on the path they traveled, with more successful solutions getting more pheromone.

In the diagram, it seems the lower path has more ants, suggesting more pheromone have been laid down there.

Path Selection: Ants incrementally favours the path with higher pheromone (the lower path in the above diagram, showing the convergence towards the most efficient path between the colony and the food source.



Stage 1: All ants are in their nest. There is no pheromone content in the environment. (For algorithmic design, residual pheromone amount can be considered without interfering with the probability)
Stage 2: Ants begin their search with equal (0.5 each) probability along each path. Clearly, the curved path is the longer and hence the time taken by ants to reach food source is greater than the other.
Stage 3: The ants through the shorter path reaches food source earlier. Now, evidently they face with a similar selection dilemma, but this time due to pheromone trail along the shorter path already available, probability of selection is higher.
Stage 4: More ants return via the shorter path and subsequently the pheromone concentrations also increase. Moreover, due to evaporation, the pheromone concentration in the longer path reduces, decreasing the probability of selection of this path in further stages. Therefore, the whole colony gradually uses the shorter path in higher probabilities. So, path optimization is attained.

Loss vs Cost

Loss Function:

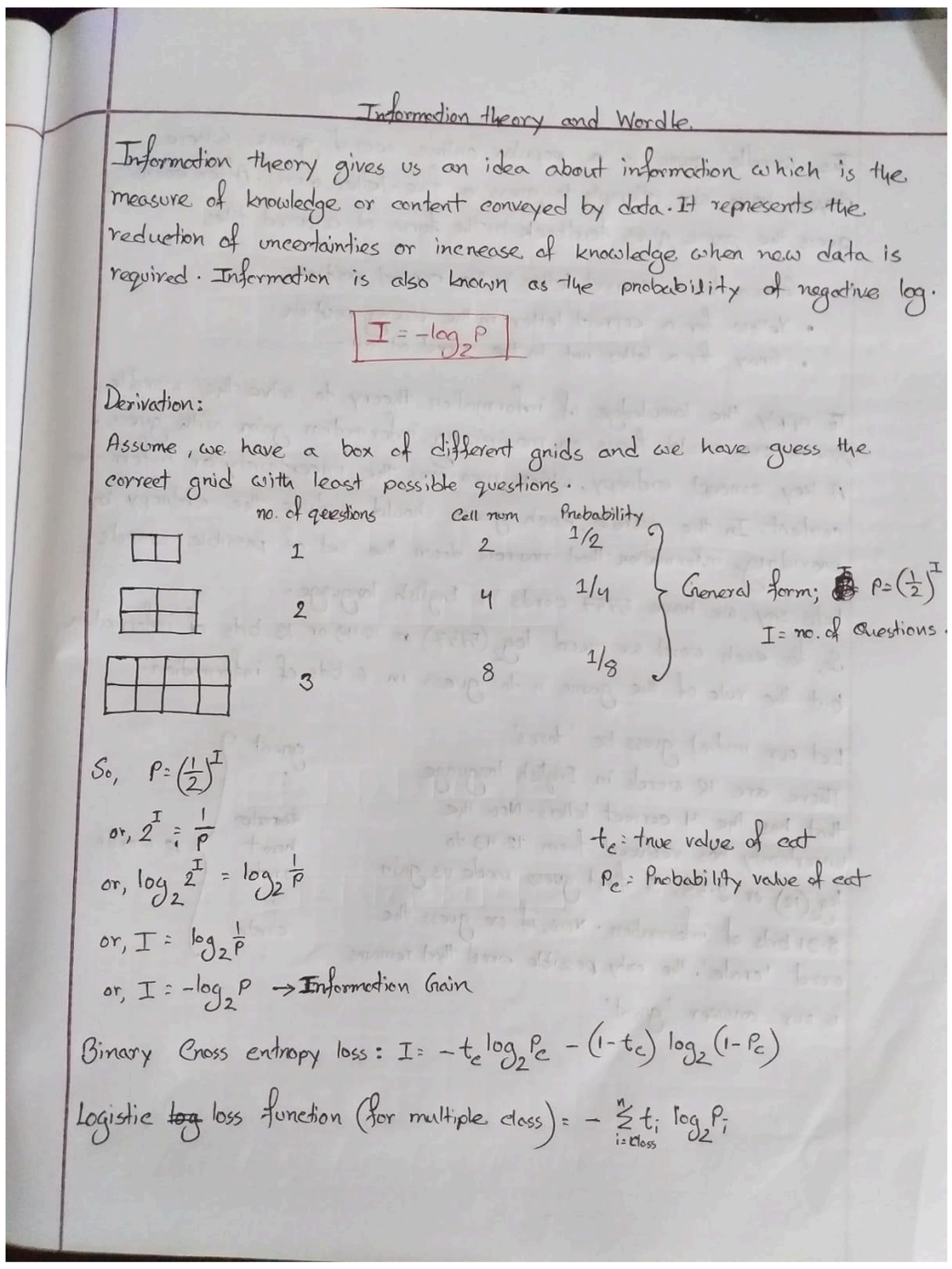
- The loss function computes the error for a single training example.
- It measures the discrepancy between the predicted value (output of the model) and the actual value (the true label or outcome).
- The goal is to minimize this error during training.
- Common examples include Mean Squared Error for regression tasks and Cross-Entropy for classification tasks.

Cost Function:

- The cost function aggregates the losses from all the training samples, usually taking the average. In other words, it's the average of the loss functions over the entire training dataset.
- It represents the total cost of a particular model, given the data and the model's parameters.
- The cost function is what optimization algorithms (like gradient descent) aim to minimize to train the model.
- It is also known as the "objective function" or "loss function" when referred to in the context of the entire training set.

Information Theory

Information refers to the quantity of data that you obtain from a particular observation or message. More unexpected observations yield more information.
Entropy is a broader statistical measure of the amount of disorder or unpredictability in a dataset. High entropy means the source can produce a very diverse set of data, and it's hard to predict what the next piece of data will be.



The wordle game is a popular online word game where players have six attempts to guess a five-letter word. After each guess, the game gives feedback in the form of coloured tiles,

- Green for a correct letter in the correct position
- Yellow for a correct letter in the wrong position
- Gray for a letter not in the word at all.

To apply the knowledge of information theory to solve the wordle game, one would focus on maximizing information gain with guess. A key concept entropy, which measures the uncertainty or information content. In the wordle, each guess should reduce the entropy by providing information that narrow down the set of possible words.

Lets say, we have 5757 words in English language.
So, for each word we need $\log_2(5757)$ or 12.49 or 13 bits of information but the rule of the game is to guess in 6 bits of information.

Let our initial guess be 'tares'
There are 12 words in English language that has the 4 correct letters. Now the uncertainty has reduced from 12.49 to $\log_2(12)$ or 3.58. So, first guess made us gain 8.91 bits of information. Now, if we guess the word 'orate'. the only possible word that remains is our answer 'great'

great
:
~~orate~~
heart
areto
:
orate
:
} 12 words

Now, what if we don't know the result. Then should we have guessed the initial guess.

So, let's say, initial guess was randomly taken and it was fuzzy. It was seen that 3543/5757 or 62% of the five letter words doesn't contain the letters of fuzzy. So, we need to guess a word that will give us highest information gain. For this reason, we can guess 'tares', as initial guess, it will have only 7% chance of giving us all grey. So, Information gain is much more than the fuzzy, which is 6.21 bits.

Optimizer:

SGD

Algorithm 8.1 Stochastic gradient descent (SGD) update at training iteration k

Require: Learning rate ϵ_k.
Require: Initial parameter $\boldsymbol{\theta}$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient estimate: $\hat{\boldsymbol{g}} \leftarrow +\frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} - \epsilon \hat{\boldsymbol{g}}$
end while

Algorithm 8.2 Stochastic gradient descent (SGD) with momentum

Require: Learning rate ϵ, momentum parameter α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient estimate: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end while

1. i) Why does the Stochastic Gradient Descent technique oscillate across the ridge? How can this oscillation be reduced? Optimizer

Ans:
Why SGD Oscillates:

- **Steep Gradients on Sides:** In areas where the loss function has a steep "ridge" or slope, the gradients are large but not necessarily pointing directly towards the global minimum. They often point across the valley.
- **Small Batch Size:** Because SGD updates parameters with a small batch of data at a time, the estimate of the gradient can have a lot of variance, leading to updates that don't consistently move in the optimal direction.

- **Fixed Step Size:** The learning rate in SGD is typically constant for each update, so it may not adapt to the topography of the loss function. A step that's too large can overshoot the minimum, causing the algorithm to bounce back and forth across the valley.

How to Reduce Oscillation:

Use momentum

- **Navigates Valleys Better:** Without momentum, standard Stochastic Gradient Descent (SGD) tends to oscillate back and forth across the narrow, steep sides of a valley because the gradient is much steeper in directions that do not point toward the optimum. With momentum, these oscillations are dampened, so the algorithm can proceed more directly toward the bottom of the valley.

- **Faster Convergence:** Momentum helps the algorithm to build up speed along surfaces that slope gently and consistently towards the optimum. This means that when the path to the solution is straightforward, momentum will accumulate, allowing the optimizer to take bigger steps and thus reach the solution faster.

SGD with momentum problem:

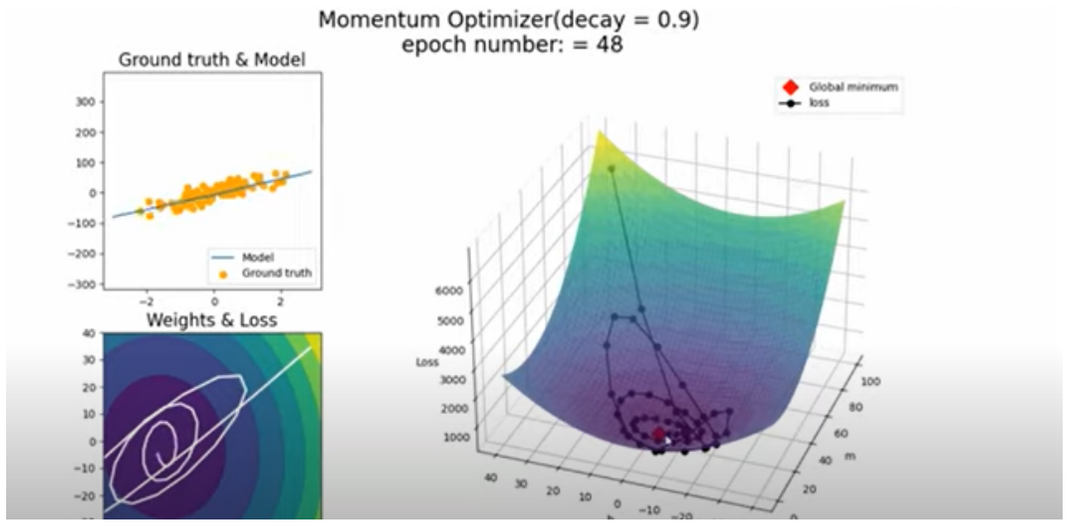
Think of SGD with momentum like skiing down a mountain blindfolded, where the slope represents the direction you should move to minimize the loss.

Following the Slope: You're trying to follow the slope you feel under your skis to reach the bottom of the mountain, which is like finding the minimum of the loss function.

Adding Momentum: Momentum is like having a memory of how you've been skiing. If you've been moving consistently in one direction, momentum helps you keep going that way, even if the slope changes suddenly.

Challenge - Blindly Following Slopes: The problem is, if the slope changes suddenly (like a sharp turn or a steep drop), momentum might keep you moving in the same direction, even though it's not the best way to go anymore. It's like if the mountain suddenly shifted, but you kept skiing in the same direction because you're relying too much on past momentum.

The main challenge is even if it find global minima it took time to calm down . because it goes back and forth in the global minima



SGD with nesterov

SGD with Classical Momentum:

- **Velocity Update:** First, compute the new velocity by applying the momentum to the previous velocity and then subtracting the gradient times the learning rate.
- **Parameter Update**: Update the parameters by adding the new velocity to the current parameters.

In classical momentum, the gradient used to update the velocity is based on the current position of the parameters.

Algorithm 8.3 Stochastic gradient descent (SGD) with Nesterov momentum

Require: Learning rate ϵ, momentum parameter α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
while stopping criterion not met **do**
  Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding labels $\boldsymbol{y}^{(i)}$.
  Apply interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$
  Compute gradient (at interim point): $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$
  Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \epsilon \boldsymbol{g}$
  Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
end while

SGD with Nesterov Momentum:

- **Interim Update**: First, make a temporary update to the parameters by adding the momentum term to the current parameters.
- **Gradient Computation:** Compute the gradient not at the current parameters, but at this temporary updated position.
- **Velocity Update:** Update the velocity by applying momentum and then subtracting the newly computed gradient.
- **Parameter Update:** Finally, update the parameters by adding the updated velocity to the current parameters.

The key innovation in Nesterov momentum is that it effectively looks ahead by computing the gradient after the momentum has been applied. This "look-ahead" gradient means Nesterov momentum can correct its course more rapidly if the momentum is leading in the wrong direction, allowing for potentially faster convergence and reduced oscillation.

In summary, while both methods use a form of momentum to accelerate the optimization process, Nesterov momentum incorporates the momentum term earlier when computing the gradient, which can often result in improved performance and faster convergence.

AdaGrad

Algorithm 8.4 The AdaGrad algorithm

Require: Global learning rate ϵ
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, perhaps 10^{-7}, for numerical stability
Initialize gradient accumulation variable $\boldsymbol{r} = \mathbf{0}$
while stopping criterion not met **do**
Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
Accumulate squared gradient: $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$
Compute update: $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. (Division and square root applied element-wise)
Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
end while

Small Constant (δ): A small constant added for numerical stability – it prevents division by zero.
Gradient Accumulation Variable (r): This keeps track of the sum of the squares of the gradients w.r.t. to each parameter θ.

The steps of the algorithm are as follows:

Initialize r: Set the gradient accumulation variable to zero.

Loop Until Convergence: Repeat the following steps until the stopping criterion is met (e.g., a certain number of iterations or a minimal change in θ).

Sample Mini-batch: Sample a mini-batch of data points and their corresponding targets.

Compute Gradient (g): Calculate the gradient of the loss function with respect to the parameters (θ) based on the mini-batch.

Accumulate Squared Gradient (r): Add the element-wise square of the gradient to the accumulation variable.

Compute Update (∆θ): Calculate the parameter update, which involves scaling the learning rate by the inverse square root of the accumulation variable (plus the small constant δ for numerical stability). This scaling is done element-wise.

Apply Update: Adjust the parameters by subtracting the computed update from the current parameter values.

The key idea of AdaGrad is to adapt the learning rate to the parameters, performing larger updates for infrequent parameters and smaller updates for frequent ones. This is beneficial for sparse data where some features can be infrequent but informative.

A known limitation of AdaGrad is that the accumulated squared gradients in the denominator can grow very large over time, causing the learning rate to shrink and the algorithm to stop learning entirely. This issue is addressed in later adaptations, like RMSprop and Adam, which use different forms of moving averages to prevent the learning rate from diminishing too quickly.

RMS prop

Algorithm 8.5 The RMSProp algorithm

Require: Global learning rate ϵ, decay rate ρ.
Require: Initial parameter $\boldsymbol{\theta}$
Require: Small constant δ, usually 10^{-6}, used to stabilize division by small numbers.
Initialize accumulation variables $\boldsymbol{r} = 0$
while stopping criterion not met **do**
  Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
  Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
  Accumulate squared gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$
  Compute parameter update: $\Delta \boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}$. $(\frac{1}{\sqrt{\delta + \boldsymbol{r}}}$ applied element-wise)
  Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta \boldsymbol{\theta}$
end while

Steps of RMSProp:

Initialize Accumulation Variables: Set the gradient accumulation variable r to 0. This variable will store a moving average of the squared gradients.

Loop Until Convergence: Repeat the following steps until a stopping criterion is met, such as a small change in θ or a maximum number of iterations.

Sample Mini-batch: Select a mini-batch of m examples from the training data.

Compute Gradient: Calculate the gradient g of the loss function with respect to the parameters θ, using the mini-batch.

Accumulate Squared Gradient: Update the accumulation variable **r** by decaying the previous value by a factor ρ (decay rate) and adding the element-wise square of the gradient g, scaled by (1 - ρ).

Compute Parameter Update: Calculate the update Δθ by dividing the learning rate ε by the square root of r + δ (where δ is a small constant to prevent division by zero), and then applying the element-wise multiplication with the gradient g.

Apply Update: Update the parameters θ by subtracting Δθ.

Integer-43

c) Examine the algorithm of RMSProp optimizer and explain how it contrasts from AdaGrad algorithm. [7]

Q1. How does the RMSProp algorithm modify the AdaGrad algorithm? Explain with proper equations. [10]

Both RMSProp and AdaGrad are adaptive learning rate algorithms, but they differ in how they handle the influence of past gradients. Here's how RMSProp modifies AdaGrad, explained with equations:

AdaGrad:

- **Update for exponential sum of squared gradients:** $\boldsymbol{r} \leftarrow \boldsymbol{r} + \boldsymbol{g} \odot \boldsymbol{g}$
- **Learning rate update:** $\Delta\boldsymbol{\theta} \leftarrow -\frac{\epsilon}{\delta + \sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}.$

Problems with AdaGrad:

- **Constantly increasing sum:** The sum of squared gradients keeps growing, leading to ever-decreasing learning rates and potentially stalling the learning process.

RMSProp modification:

- **Introduces a decay rate:** $\boldsymbol{r} \leftarrow \rho\boldsymbol{r} + (1-\rho)\boldsymbol{g} \odot \boldsymbol{g}$
- **Decay rate (ρ):** This parameter controls the influence of past gradients. 0 < ρ < 1, typically close to 1 (e.g., 0.9).

How it works:

- The decay rate acts like a forgetting factor. Older gradients are "forgotten" faster with a higher ρ, giving more weight to recent updates.
- This prevents the sum of squared gradients from growing indefinitely, allowing the learning rate to adapt more effectively.

Equations:

- **RMSProp update:** $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1-\rho)\boldsymbol{g} \odot \boldsymbol{g}$
- **Learning rate update:** $\Delta\boldsymbol{\theta} = -\frac{\epsilon}{\sqrt{\delta + \boldsymbol{r}}} \odot \boldsymbol{g}.$

Essentially:

- RMSProp incorporates a decay rate into the sum of squared gradients calculation, preventing it from constantly growing and allowing the learning rate to adapt more effectively.

Benefits of RMSProp:

- Faster convergence compared to AdaGrad.
- Less sensitive to noise in the gradients.
- Can be more effective in scenarios with non-stationary data.

Modification from AdaGrad:

- AdaGrad accumulates the squared gradients `g^2` directly, without any decay rate, leading to a monotonically increasing accumulation that can grow very large, causing the learning rate to become excessively small.
  AdaGrad update rule for `θ`:

$$\theta_{\text{new}} = \theta_{\text{old}} - \frac{\epsilon}{\sqrt{\sum_{i=1}^{t} g_i^2 + \delta}} g_t$$

- RMSProp modifies this by introducing a decay rate `ρ` that weights the accumulated squared gradient more towards recent gradients, preventing the accumulation from growing too large.
  RMSProp update rule for `θ`:

$$r = \rho r + (1 - \rho) g^2$$

$$\theta_{\text{new}} = \theta_{\text{old}} - \frac{\epsilon}{\sqrt{r + \delta}} g$$

Contrast with AdaGrad:

- While AdaGrad's accumulated squared gradient grows without bound, RMSProp's accumulation is a moving average, which does not grow indefinitely due to the decay rate.
- This means RMSProp can continue learning even when AdaGrad's updates become negligibly small because the denominator in RMSProp's update rule doesn't grow as large as AdaGrad's, which prevents the learning rate from diminishing too much.
- RMSProp is generally more robust and can perform better on problems where AdaGrad slows down too fast or stops making progress.

Algorithm 8.6 RMSProp algorithm with Nesterov momentum

Require: Global learning rate ϵ, decay rate ρ, momentum coefficient α.
Require: Initial parameter $\boldsymbol{\theta}$, initial velocity $\boldsymbol{v}$.
  Initialize accumulation variable $\boldsymbol{r} = \boldsymbol{0}$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute interim update: $\tilde{\boldsymbol{\theta}} \leftarrow \boldsymbol{\theta} + \alpha \boldsymbol{v}$
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\tilde{\boldsymbol{\theta}}} \sum_i L(f(\boldsymbol{x}^{(i)}; \tilde{\boldsymbol{\theta}}), \boldsymbol{y}^{(i)})$
    Accumulate gradient: $\boldsymbol{r} \leftarrow \rho \boldsymbol{r} + (1 - \rho) \boldsymbol{g} \odot \boldsymbol{g}$
    Compute velocity update: $\boldsymbol{v} \leftarrow \alpha \boldsymbol{v} - \frac{\epsilon}{\sqrt{\boldsymbol{r}}} \odot \boldsymbol{g}$. ($\frac{1}{\sqrt{\boldsymbol{r}}}$ applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \boldsymbol{v}$
  end while

Comparison with Regular RMSProp:

The key difference between RMSProp with Nesterov momentum and standard RMSProp is the addition of the Nesterov momentum term. In standard RMSProp, the gradient is computed based on the current position of the parameters, and the update is applied immediately. With Nesterov momentum, the gradient is computed at a "look-ahead" position, allowing the algorithm to correct its course more effectively if it is heading towards a sub-optimal direction.

The use of Nesterov momentum can potentially result in faster convergence and less oscillation during training, especially in the context of complex optimization landscapes common in deep learning.

Standard momentum might cause you to overshoot the minimum because you calculate the gradient and then add momentum, which might lead you to miss the lowest point if the momentum is too high.

Nesterov momentum helps to anticipate where the parameters are going to be after the update and calculates the gradient there. This 'foreseeing' step means you get a more accurate update because you're effectively looking at the landscape ahead and adjusting your movements accordingly.

Adam

Algorithm 8.7 The Adam algorithm

Require: Step size ϵ (Suggested default: 0.001)
Require: Exponential decay rates for moment estimates, ρ_1 and ρ_2 in $[0, 1)$. (Suggested defaults: 0.9 and 0.999 respectively)
Require: Small constant δ used for numerical stabilization. (Suggested default: 10^{-8})
Require: Initial parameters $\boldsymbol{\theta}$
  Initialize 1st and 2nd moment variables $\boldsymbol{s} = \mathbf{0}$, $\boldsymbol{r} = \mathbf{0}$
  Initialize time step $t = 0$
  while stopping criterion not met **do**
    Sample a minibatch of m examples from the training set $\{\boldsymbol{x}^{(1)}, \ldots, \boldsymbol{x}^{(m)}\}$ with corresponding targets $\boldsymbol{y}^{(i)}$.
    Compute gradient: $\boldsymbol{g} \leftarrow \frac{1}{m} \nabla_{\boldsymbol{\theta}} \sum_i L(f(\boldsymbol{x}^{(i)}; \boldsymbol{\theta}), \boldsymbol{y}^{(i)})$
    $t \leftarrow t + 1$
    Update biased first moment estimate: $\boldsymbol{s} \leftarrow \rho_1 \boldsymbol{s} + (1 - \rho_1)\boldsymbol{g}$
    Update biased second moment estimate: $\boldsymbol{r} \leftarrow \rho_2 \boldsymbol{r} + (1 - \rho_2)\boldsymbol{g} \odot \boldsymbol{g}$
    Correct bias in first moment: $\hat{\boldsymbol{s}} \leftarrow \frac{\boldsymbol{s}}{1 - \rho_1^t}$
    Correct bias in second moment: $\hat{\boldsymbol{r}} \leftarrow \frac{\boldsymbol{r}}{1 - \rho_2^t}$
    Compute update: $\Delta\boldsymbol{\theta} = -\epsilon \frac{\hat{\boldsymbol{s}}}{\sqrt{\hat{\boldsymbol{r}}} + \delta}$ (operations applied element-wise)
    Apply update: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \Delta\boldsymbol{\theta}$
  end while

First Moment (s):

- The first moment is the moving average of the gradients. It's like measuring the speed of a car; it tells you how fast the parameters are changing.
- It's calculated as a combination of the gradient (how much we want to change parameters based on how wrong they were) and the previous first moment (how much we changed parameters in the past).
- This helps to smooth out the updates, which can make the learning process more stable. It's beneficial because it can prevent the optimizer from making too large updates in any single iteration, which can cause the learning process to become unstable.

Second Moment (r):

- The second moment is the moving average of the squares of the gradients. It's akin to measuring the acceleration of a car; it tells you how fast the speed itself is changing.
- It's calculated as a combination of the squared gradient (which emphasizes larger errors) and the previous second moment.
- This helps to adapt the learning rate based on the recent history of gradients. If the gradients are consistently large, the optimizer will take smaller steps. If the gradients are

small, it can take larger steps. This is beneficial because it allows for finer control over the learning process, letting the algorithm take confident steps in flat areas (where the gradients are small) and cautious steps in steep areas (where the gradients are large).
- Both moments are corrected to counteract their initialization at zero, which is biased towards zero, especially during the initial time steps. Bias correction allows the moments to more accurately represent the actual gradients and gradient squares.

Benefits of Using Both Moments:

- The first moment accounts for the magnitude of the gradients (speed), providing momentum to the optimization process, which can help escape local minima.
- The second moment adjusts the learning rate based on the variance of the gradients (acceleration), which ensures that each parameter's update is scaled appropriately, preventing too large updates that could overshoot minima.
- Together, they allow Adam to be more effective in practice than algorithms that consider only the first moment (like SGD with momentum) or only the second moment (like RMSProp). They enable Adam to combine the benefits of both approaches, making it robust to different kinds of objective functions and different parts of the parameter space.

First Moment (Mean of Gradients):

- Think of the first moment as a measure of the direction and speed of your parameter updates. In physical terms, it's like momentum: if you've been moving in a certain direction, you're likely to keep moving in that direction.
- Mathematically, it's a moving average of the gradients. Because Adam starts with an initial moving average of zero, the first few updates are smaller than they should be (since the average of gradients starts from zero and slowly "warms up"). This is the "bias" towards zero.

Second Moment (Unscaled Variance of Gradients):

- The second moment is a measure of the variability or spread of your gradients. It's like a measure of the terrain's roughness as you're trying to roll a ball down a hill. If the terrain (gradients) is very rough (high variance), you'll want to be more careful with your steps (updates).
- It is the moving average of the squared gradients. Just like the first moment, this average also starts from zero and slowly "warms up," leading to a bias towards zero.

Bias Correction:

- Because both first and second moments are biased towards zero at the start (due to initial values of zero), Adam performs a correction to make them more accurate. Without this correction, the algorithm would think the gradients are smaller than they really are and make smaller updates, especially early on.
- The bias correction adjusts these moments by scaling them up based on how many steps (iterations) you've taken. The idea is to compensate for the initial underestimation so that the corrected moments accurately reflect the true scale of the gradients.
- Practically, it's like saying, "I've been underestimating the slope of this hill because I'm just starting to roll; let me adjust my expectations to be more realistic about the steepness."

How Bias Correction Works:

- For the first moment, we divide by $1 - \beta_1^t$, where β_1 is the decay rate for the first moment, and t is the current time step. Since β_1 is less than 1, raising it to the power of t makes it very small, so we're dividing by a number just less than 1, scaling the first moment up.
- For the second moment, we divide by $1 - \beta_2^t$, with similar logic.

Benefit of Bias Correction:

- This process ensures that at the start, when the estimates of the moments are too low, the corrections help the optimizer to make appropriately larger updates.
- As more iterations pass and t gets larger, the correction factor $1 - \beta_1^t$ (or $1 - \beta_2^t$) gets closer to 1, which means the correction effect diminishes, as the moving averages have "warmed up" and are no longer biased.

Differences:

Stochastic Gradient Descent (SGD):

- Uses a mini-batch of data to compute the gradient and update the parameters.
- Learning rate is fixed or manually decayed over time.
- Can oscillate and take a longer time to converge due to the variance in gradient estimates.

SGD with Momentum:

- Similar to SGD but uses a velocity vector to smooth out updates.
- Accumulates gradient direction from previous steps to dampen oscillations and converge faster.

SGD with Nesterov Momentum:

- An enhancement of momentum SGD that calculates the gradient at a "look-ahead" position based on the current momentum.
- Allows for correction before the parameter update, potentially leading to faster convergence.

AdaGrad:

- Adjusts the learning rate based on the historical sum of squared gradients.
- Learning rate is scaled inversely proportional to the square root of the accumulation of past squared gradients.
- Can be too aggressive with learning rate reduction, effectively stopping early in training.

RMSProp:

- Modifies AdaGrad by using a moving average of squared gradients instead of summing all past squared gradients.
- Prevents the learning rate from diminishing to an extremely small value, which addresses AdaGrad's aggressive decay.

RMSProp with Nesterov Momentum:

- Combines RMSProp's adaptive learning rate with Nesterov momentum.
- Benefits from the anticipatory update of Nesterov momentum and the adaptive learning rate of RMSProp.

Adam:

- Combines ideas from both momentum and RMSProp.
- Maintains a moving average of both gradients and squared gradients and includes bias corrections for both of these estimates.
- Adjusts the learning rate for each parameter based on the first and second moments.

Key Differences:

- **Learning Rate Adaptation:** SGD does not adapt the learning rate per parameter, while AdaGrad, RMSProp, and Adam do, using different strategies to adjust it based on historical gradient information.
- **Momentum:** Only the algorithms with "momentum" in the name explicitly incorporate momentum to accelerate the convergence.
- **Nesterov Momentum:** Provides a "look-ahead" feature by computing the gradient after a preliminary update based on current momentum, which can lead to more effective parameter updates.
- **Gradient Accumulation:** AdaGrad accumulates the square of gradients over all past steps, while RMSProp and Adam use a moving average, preventing excessive diminishing of the learning rate.
- **Bias Correction:** Adam specifically includes a bias correction step for both first and second moment estimates, which helps to provide more accurate estimates early in training.

Each algorithm has its advantages and is suited to different types of problems. For instance, Adam is often preferred for training deep learning models due to its robustness and good performance on a wide range of problems. RMSProp is also commonly used, especially in recurrent neural networks. Classic SGD and SGD with momentum are simpler and can still be effective, especially when coupled with learning rate schedules.

Integer-43

Q: Explain the importance of optimizer in NN.

In neural networks, an optimizer is a crucial component that governs the update of the network's weights in response to the error output from the loss function. The role of the optimizer is to minimize (or, in some cases, maximize) the loss function, which measures the difference between the network's predictions and the actual data.

Here's why the optimizer is so important:

1. **Convergence**: An optimizer determines how quickly and effectively a neural network can converge to the lowest possible loss, which typically corresponds to the most accurate predictions the model can provide.
2. **Navigating the Loss Landscape**: Neural networks, especially deep ones, often have complex loss landscapes with many local minima, maxima, and plateaus. An effective optimizer navigates this landscape to find a good minimum that corresponds to a well-performing model.
3. **Speed of Training**: Different optimizers converge at different speeds. Some, like SGD, may take more time compared to more advanced ones like Adam, which can converge faster due to adaptive learning rates.
4. **Handling Vanishing/Exploding Gradients**: Some optimizers are designed to mitigate common issues like vanishing and exploding gradients, which can halt the learning process or lead to unstable training.

Word Embedding

Word Embedding is one such technique where we can represent the text using vectors.

BoW is a method to extract features from text for use in modeling, such as with machine learning algorithms. A bag of words is a representation of text that describes the occurrence of words within a document. It involves two things: a vocabulary of known words and a measure of the presence of known words.
It's called a "bag" of words because any information about the order or structure of words in the document is discarded. The model is only concerned with whether known words occur in the document, not where in the document.

TF-IDF stands for Term Frequency-Inverse Document Frequency, which reflects how important a word is to a document in a collection or corpus. The TF-IDF value increases proportionally to the number of times a word appears in the document and is offset by the frequency of the word in the corpus, which helps to adjust for the fact that some words appear more frequently in general. TF-IDF is often used in text mining and information retrieval to reflect the importance of a term to a document relative to the corpus.

Integer-43

Question 5. [Marks: 14]

a) List the different instances of cosine similarity. [3]
b) Illustrate the idea of Negative Sampling using the following sentence. [4]

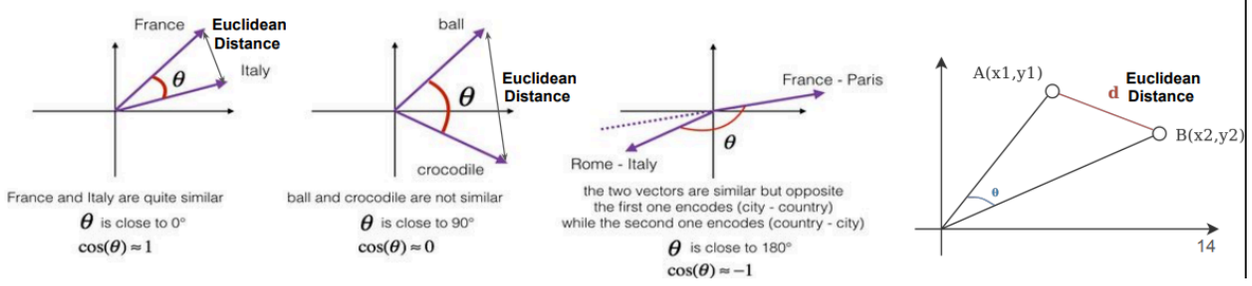
"The cat chased a red ball across the grassy yard."

a) Cosine similarity measures the cosine of the angle between two non-zero vectors in a multi-dimensional space, often used to compare the similarity between two documents or word vectors in text analysis. **Explain the diagrams**

$$\text{CosineSimilarity(u, v)} = \frac{u \bullet v}{||u||_2 . ||v||_2} = cos(\theta) \quad = \frac{u \bullet v}{\sqrt{u \bullet u}\sqrt{v \bullet v}}$$

where **(u.v)** is the dot product (**or inner product**) of two vectors, denominator is the **L2 norm (or length)** of the vector **u** and **v**, and **ϴ** is the angle between **u** and **v**.

This similarity depends on the angle between **u** and **v.** If **u** and **v** are very similar, their **cosine similarity will be close to 1**; if they are dissimilar, the cosine similarity will have a smaller value.



b) Negative sampling is a technique used to reduce the computational complexity of training a word embedding model. Instead of predicting the presence or absence of each word in the vocabulary for a given context (as in a softmax classifier), negative sampling simplifies the problem to predicting whether a given word is likely to appear in a context. It does this by selecting a small subset of "negative" words (words not in the context) to update the weights for a given "positive" word (a word in the context).

Let's illustrate negative sampling using the sentence "The cat chased a red ball across the grassy yard":

Positive Sample: First, we pick a target word and its context words based on a window size. For example, if we choose "chased" as the target word with a window size of 1, our positive samples would be ("the", "chased") and ("chased", "a").

Negative Samples: Then, for each positive pair, we randomly select 'k' words from the vocabulary as negative samples. These words are chosen based on a probability distribution that down-weights frequent words. The frequency of each word raised to the power of 3/4 and then normalized across the vocabulary.

Training: The model is then trained to distinguish the positive samples from the negative samples. For the word "chased", we would train the model to predict '1' for the word "the" and "a" as they are in the context of "chased", and '0' for the 'k' negative samples, which could be any words that are not in the immediate context of "chased".

For example, if we choose 'k' to be 2, and the negative words randomly selected are "orange" and "yard" (assuming they are not within the chosen context window for "chased"), we would have the following training samples:

Positive: ("the", "chased") -> 1
Positive: ("chased", "a") -> 1
Negative: ("chased", "orange") -> 0
Negative: ("chased", "yard") -> 0
By doing this for each word in the sentence, and across many sentences in a large corpus, the model learns word embeddings that are good at predicting the actual context words while distinguishing them from random words, effectively capturing the semantic relationships between words with much less computation than a full softmax classifier.

Integer-43

Question 6. [Marks: 14]

a) Explain the importance of inverse document frequency in the concept of TF-IDF.

Inverse Document Frequency (IDF) is a crucial component of the TF-IDF (Term Frequency-Inverse Document Frequency) weighting scheme, commonly used in information retrieval and text mining to reflect the importance of a term to a document in a corpus.

Here's why IDF is important in the concept of TF-IDF:

1. **Discrimination of Common Words**: IDF decreases the weight for commonly used words and increases the weight for words that are not used as much across the corpus. Common words like "the", "is", and "and" typically don't contain useful information about the content of a document, as they appear in most documents (high document frequency). The IDF component effectively down-weights these words.
2. **Highlighting Unique Terms**: Conversely, if a term appears rarely across the corpus, it is likely to be more significant in understanding the content of a document where it does appear. IDF assigns higher weights to such terms, making them more influential in the context of the document.
3. **Scaling Term Relevance**: The logarithmic scale of IDF allows for a level of discrimination where the importance of a term does not increase linearly with its rarity, preventing extremely rare terms from dominating the term weight.
4. **Improved Query Relevance**: In search engines and information retrieval systems, using TF-IDF with IDF helps in returning documents that are more relevant to the user's query, as it prefers documents that contain rare terms found in the user's query.

The formula for IDF is typically calculated as follows:

$$\text{IDF}(t) = \log \frac{N}{n_t}$$

Where:

- N is the total number of documents in the corpus.
- n_t is the number of documents where the term t appears.

By multiplying TF (the frequency of the term in the document) with IDF, you get the TF-IDF score, which balances the term frequency with its importance across the entire corpus. This score is then used as a weighting factor for terms in tasks like document classification, clustering, and information retrieval.

Important:
The hierarchical softmax classifier is a method used to speed up the training of word embeddings, particularly in models like Word2Vec. It replaces the traditional softmax classifier when the vocabulary size is large. Here's why it was necessary and its drawbacks:

Necessity of Hierarchical Softmax Classifier:

- **Computational Efficiency:** In a standard softmax classifier, the computation of probabilities for all words in the vocabulary is very expensive, especially when the vocabulary size is large (like 10,000 or more words). The complexity is O(n) for n vocabulary size because it requires computing and normalizing the exponential score for each word.

- **Reducing Complexity:** Hierarchical softmax uses a binary tree representation of the output layer with words as leaves. This significantly reduces the complexity of the probability computation from O(n) to O(log n), where n is the number of words in the vocabulary. Only a path from the root to the leaf node needs to be considered during each training step, which involves much fewer computations.

- **Frequency-based Structuring:** By structuring the tree so that more frequent words are closer to the root, the model can learn these words faster, which is efficient given that frequent words are more likely to appear in training samples.

Alternative for hierarchical softmax classifier is negative sampling

Activation functions
Integer-43

b) Identify the issues of Sigmoid activation function. Are the issues resolved by the hyperbolic tangent activation function? [4]

The Sigmoid activation function, commonly used in neural networks, has a couple of well-known issues:

1. **Vanishing Gradient Problem**: The Sigmoid function squashes its input values into a very small output range in a non-linear fashion. When the input values are very positive or very negative, the function saturates at 1 or 0, with a derivative extremely close to 0. This means that for such values, the gradient is almost zero, and when backpropagation occurs, the weights and biases for the neurons in earlier layers of the network are barely updated. As a result, the learning process is very slow, and it may stop entirely if the gradients become too small, effectively "vanishing."
2. **Output Not Centered at Zero**: The output of the Sigmoid function is always positive, which means the output of the neurons using this activation function is not zero-centered. This can lead to gradients being pushed towards positive or negative values, causing an inefficient zig-zagging dynamic in the gradient updates for the weights. This is also known as the problem of "neurons that fire together wire together," which can be problematic during optimization.
3. **Computationally Expensive**: Exponential functions (which the Sigmoid function is based on) are more computationally intensive than other activation functions like ReLU.

The hyperbolic tangent function (tanh) is a scaled version of the Sigmoid and addresses some of its issues:

1. **Zero-Centered**: The output of the tanh function is zero-centered because it squashes inputs to a range between -1 and 1. This means that the data flowing through the network will, on average, have a mean of 0, which helps with the convergence during gradient descent.
2. **Stronger Gradients**: Since the range is between -1 and 1, the gradients are stronger (higher derivatives) than those of the Sigmoid function, which helps alleviate the vanishing gradients problem to some extent.

However, tanh still suffers from the vanishing gradient problem, although to a lesser degree than the Sigmoid function. Like the Sigmoid, it can also saturate, resulting in small gradients during backpropagation and slow learning.