# Data Mining & Machine Learning – Assignment 3

(20% of total grade)

* If two homework submissions are found to be similar to each other, both submissions will receive 0.
* Homework solutions must be submitted through Canvas. If you have multiple files, please include all files as one zip file.
* For coding assignments, it is strongly recommended to use **Jupytor notebook** and submit **.ipynb** file.
* Answers with **math expressions** and **graphs** can be handwritten and scanned.
* If you find any **typo/error** in the assignment, let me know.

**1. [5 pts] In a decision tree, we can use 'Information gain' or 'Gini' as the measure in attribute selection.**
**When class distribution is very skewed (e.g., most class values are 0 or 1), explain in detail which one is better. Explain this using the formula of Information Gain and Gini.**

When selecting attributes in a decision tree with a highly imbalanced class distribution, two commonly used impurity measures are Information Gain (IG) and Gini Index (GI). Let's delve into how each performs and explore why Information Gain might be a more suitable choice for skewed data.

1. **Information Gain (IG)**:
   - **Formula**: IG is calculated as the difference between the entropy of the original dataset and the entropy of its subsets after splitting on a specific attribute.

$$IG(A) = \text{Entropy}(T) - \sum_{v \in \text{Values}(A)} \frac{|T_v|}{|T|} (T_v)$$

where T is the original dataset, Tv represents subsets created by splitting on attribute A, and Entropy quantifies the uncertainty in classification before and after the split.

   - **Performance with Skewed Data**: Since entropy reaches its maximum at a 50-50 distribution and declines as distribution becomes skewed (approaching 0 or 1), Information Gain favors attributes that create a subset with more homogeneity, which is beneficial in a skewed distribution. This helps in identifying the most impactful splits when most instances belong to one class.

2. **Gini Index (GI)**:

   - **Formula**: The Gini Index is calculated as:

$$GI(T) = 1 - \sum_{i=1}^{c} p_i^2$$

where pipi is the probability of a class ii in dataset TT, and cc is the number of classes.

- **Performance with Skewed Data**: Unlike IG, the Gini Index doesn't inherently account for the overall entropy of the dataset before the split. This can lead to misleading attribute choices in skewed data, as it doesn't weigh the importance of homogeneity in the class distributions. Thus, it may select splits that do not significantly improve the classification of minority classes in a skewed dataset.

**2. [5 pts] When a problem is non-linear separable, the Perceptron Training Rule never stops, while the Delta Rule converges to a local optimum and stops. Explain the reason in detail using the update rule of each method.**

For non-linearly separable data, the Perceptron Training Rule and the Delta Rule have distinct behaviors due to their respective weight update mechanisms.

1. **Perceptron Training Rule**:
    - **Update Formula**: The Perceptron Training Rule updates weights when an error is made, defined as:

$$w = w + \eta(y - \hat{y})x$$

where w represents the weight vector, η is the learning rate, y is the true class label, and y^ is the predicted output.

- **Behavior**: For linearly separable data, the perceptron's rule ensures convergence by adjusting weights to find a decision boundary. However, when the data is non-linearly separable, the perceptron cannot find a hyperplane that accurately separates the classes. Consequently, the perceptron oscillates as it repeatedly updates the weights, attempting to classify misclassified points but unable to settle on a solution because there is no true solution for separating the data.

2. **Delta Rule** (often associated with gradient descent in neural networks):

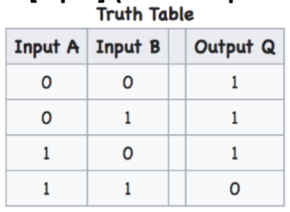
- **Update Formula**: The Delta Rule aims to minimize the total error by gradient descent, using the following update:

$$w = w + \eta(y - \hat{y})\hat{y}(1 - \hat{y})x$$

Here, y^ is derived from a continuous activation function (such as the sigmoid), allowing the rule to adjust weights based on error gradients, which smooths out updates towards the error minimum.

- **Behavior**: Unlike the Perceptron Training Rule, the Delta Rule doesn't aim for a perfect classification boundary. Instead, it minimizes the squared error, converging to a local minimum even when the data is non-linearly separable. Once it reaches this minimum, it stops, indicating a state where the error has been reduced to its lowest possible value for the given configuration, even if some misclassifications still occur.

**3. [5 pts] (Refer to p. 9-10 in the slides) Implement the following NAND table using a perceptron.**


Truth Table

| Input A | Input B | | Output Q |
|---|---|---|---|
| 0 | 0 | | 1 |
| 0 | 1 | | 1 |
| 1 | 0 | | 1 |
| 1 | 1 | | 0 |

To implement a NAND gate using a perceptron, we must configure the weights and bias to ensure that the perceptron generates the desired output for every combination of inputs, aligning with the truth table.

formula is

$$y = f(w1 \cdot A + w2 \cdot B + b)$$

where,

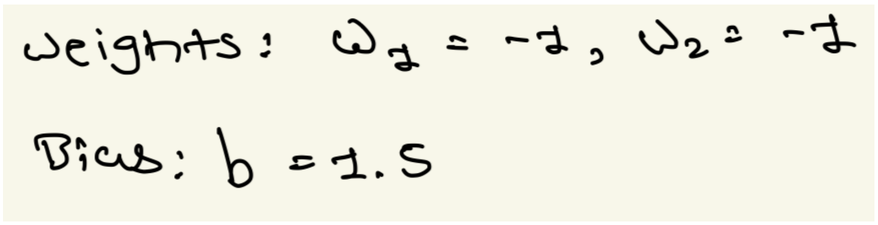
* $w_1$ & $w_2$ are the weights for input A & B,

* b is the bias term,

* f is the activation function, typically step function for Perceptron!

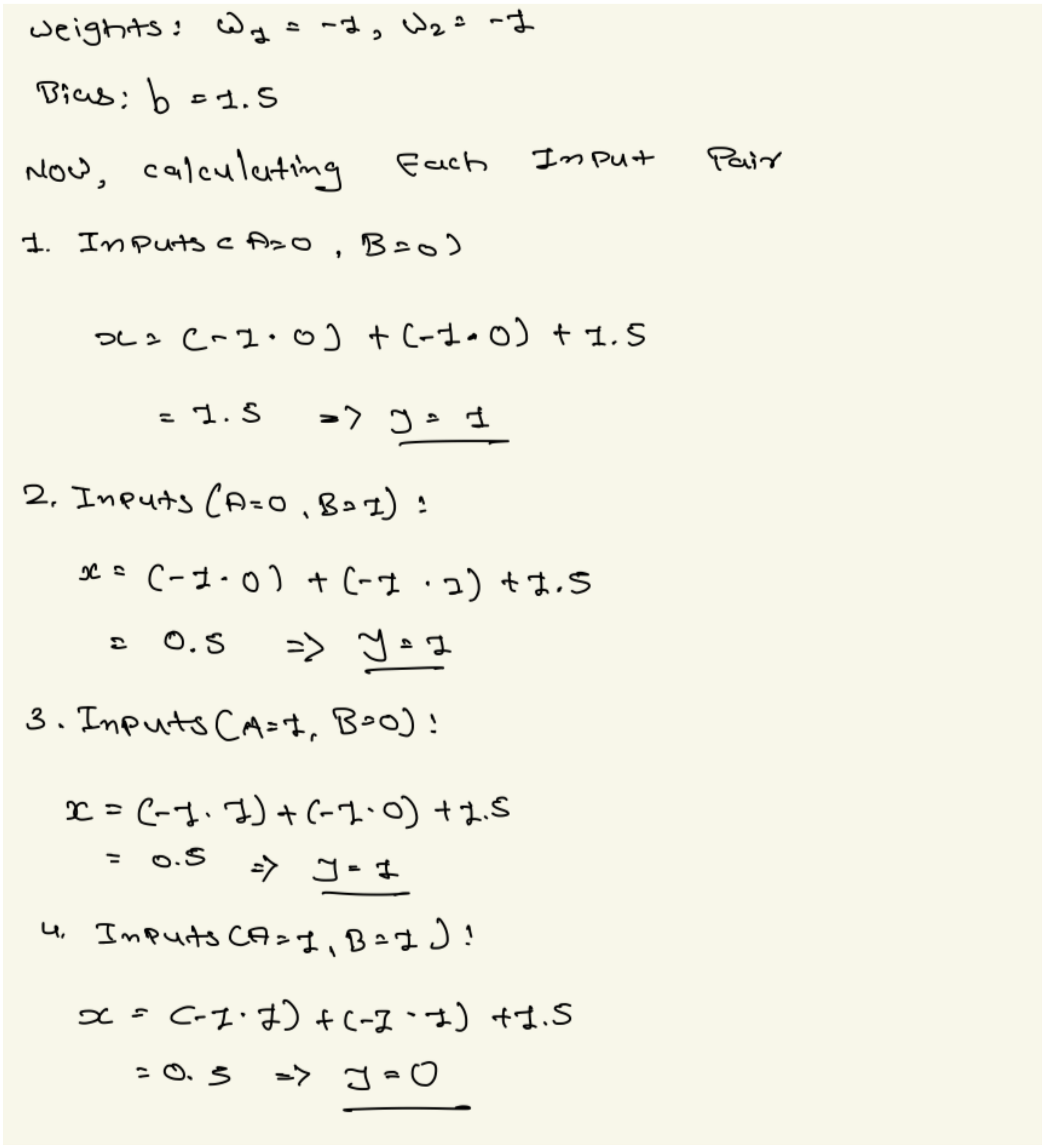
$$f(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

For a NAND gate, we need the output to be 1 in all cases except when both inputs are 1. To achieve this, we can choose the following values:



weights: $\omega_1 = -1$, $\omega_2 = -1$

Bias: $b = 1.5$



weights: $\omega_1 = -1$, $\omega_2 = -1$

Bias: $b = 1.5$

Now, calculating Each Input Pair

1. Inputs (A=0 , B=0)

$$x = (-1 \cdot 0) + (-1 \cdot 0) + 1.5$$
$$= 1.5 \Rightarrow \underline{y = 1}$$

2. Inputs (A=0 , B=1) :

$$x = (-1 \cdot 0) + (-1 \cdot 1) + 1.5$$
$$= 0.5 \Rightarrow \underline{y = 1}$$

3. Inputs (A=1, B=0) :

$$x = (-1 \cdot 1) + (-1 \cdot 0) + 1.5$$
$$= 0.5 \Rightarrow \underline{y = 1}$$

4. Inputs (A=1 , B=1 ) :

$$x = (-1 \cdot 1) + (-1 \cdot 1) + 1.5$$
$$= 0.5 \Rightarrow \underline{y = 0}$$

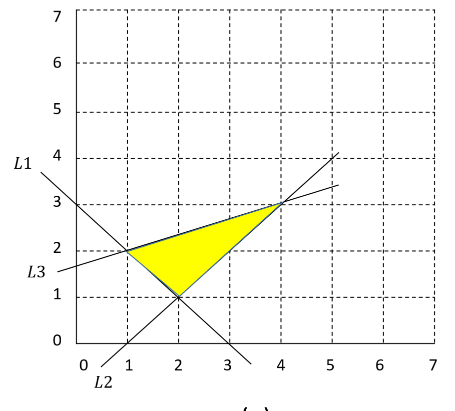
**4. [3 pts] Explain why deep layer has much smaller # of parameter than shallow one using the following formula.**
**# parameters ≈ $\mathbf{width}^2$ · depth**

Deep layers in a neural network have a smaller number of parameters compared to shallow layers because of the structural efficiency achieved by increasing depth rather than width, as calculated by the formula # parameters≈width2·depth.

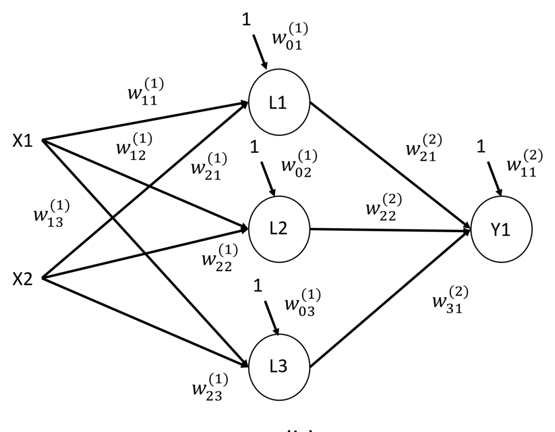
1. **Parameter Calculation**: For a layer of width w and depth d, the approximate number of parameters is proportional to w^2 * d. This formula suggests that the number of parameters increases exponentially with the square of the width but only linearly with the depth.
2. **Efficiency of Deep Networks**: In deep networks, information undergoes a progressive transformation across multiple layers, enabling the generation of intricate representations with fewer nodes per layer. Each layer extracts features from the preceding layer, accumulating feature transformations without necessitating a substantial number of neurons in each layer.
3. **Comparison to Shallow Networks**: Shallow networks rely on the width of their layers (i.e., the number of neurons within a single layer) to capture data complexity. This approach significantly increases the number of parameters required due to the relationship between the number of neurons and the number of weights (w2). Consequently, a shallow network with comparable capacity would necessitate an exceptionally wide architecture, resulting in an exponential increase in parameters. This computational burden makes it prone to overfitting.

Deeper architectures manage complexity more efficiently by stacking layers, keeping the width manageable, and reducing the total parameter count compared to an equally powerful shallow network.

**5. (Refer to p. 6-7 in the slides) Universal Approximation Theorem (for classification)**
**It is known that a neural network with one (or two hidden) layers can represent any classification boundary. Refer to the following classification boundary and a two layer neural network.**



(a)



(b)

The yellow region in figure (a) is a classification boundary, and we are going to represent the boundary using two layers neural network in figure (b). In other words, if a data point (x1, x2) falls within the yellow region, its target value is 1, 0 otherwise.

In figure (b), each (hidden/output) node is a perceptron using a step function. L1, L2 and L3 represents the linear boundary in figure (a), respectively. For example, if input data lies above L1 line, its output is 1, otherwise 0. If it lies above L2 line, output is 1, otherwise 0. if it lies below L3 line, output is 1, otherwise 0.

**1) [3 pts] Show the formula of lines L1, L2, and L3, respectively.**

To derive the equations for the lines L1L1, L2L2, and L3L3 in the classification boundary (yellow region) in Figure (a), we can analyze the orientation and position of these lines on the graph:

1. **Line L1**:
   L1 appears to have a negative slope and intersects the y-axis at approximately y=4.
   Assuming a slope of −1, the equation for L1 could be represented as:

   L1: Y = −X + 4

2. **Line L2**:
   L2 has a positive slope and passes through the origin (0,0).
   Assuming a slope of approximately 1/2, the equation for L2 would be:

   L2: Y = 0.5X

3. **Line L3**:
   L3L3 has a negative slope and intersects the y-axis at approximately y=2.
   Assuming a slope of −1/2, the equation for L3 would be:

   L3: y= −0.5x + 2

These equations for L1, L2, and L3 define the linear boundaries represented in Figure (a).

**2) [3 pts/ea] Define all parameter values (weights and activation function) of perceptron L1, L2 and L3, respectively. Explain how each perceptron works using these parameters. You have to explain using these parameters in detail.**

To implement the perceptron's L1, L2, and L3 based on the lines given in the classification boundary figure (a), we need to define the weights and bias for each perceptron such that the perceptrons produce output 11 when the input lies within the yellow region defined by the lines L1, L2, and L3.

Each perceptron L1, L2, and L3 will use a step activation function.

**From the derived equation for**,

L1:

L1: y = −x + 4

Rearrange to match the perceptron format w1·x1+w2·x2+b=:
X + y – 4 = 0
So, the weights and bias for L1 are:

- w1=1 (for x1)
- w2=1 (for x2)
- b=−4

**Explanation**

- This perceptron will output 1 if x + y ≥ 4 (i.e., the point lies above or on the line L1), otherwise it outputs 0.

L2:

L2: y = 0.5x
Rearrange to match the perceptron format:
−0.5x+y=0
So, the weights and bias for L2L2 are:

- w1=−0. (for x1)
- w2= (for x2)
- b=0

**Explanation**

- This perceptron will output 1 if −0.5x + y ≥ 0 (i.e., the point lies above or on the line L2), otherwise it outputs 0.

L3:

L3: y = −0.5x + 2
Rearrange to match the perceptron format:
0.5x + y – 2 = 0
So, the weights and bias for L3L3 are:

- w1 = 0.5 (for x1)
- w2=1 (for x2)
- b = −2

**Explanation**

- This perceptron will output 1 if 0.5x + y ≥ 2 (i.e., the point lies below or on the line L3), otherwise it outputs 0.

**3) [5 pts] Define all parameter values (weights and activation function) of perceptron Y1. Explain how Y1 is able to find the boundary region.**

To implement the output perceptron Y1 in the neural network, we need to define its weights and bias such that it produces an output of 1 when the input point lies within the yellow region (defined by the boundaries created by L1, L2, and L3) and outputs 0 otherwise.

**Logic of Perceptron Y1**

The perceptrons L1, L2, and L3 create intermediate outputs based on whether a point lies above or below each line:

- L1=1: Point lies above or on line L1.
- L2=1: Point lies above or on line L2.
- L3=1: Point lies below or on line L3.

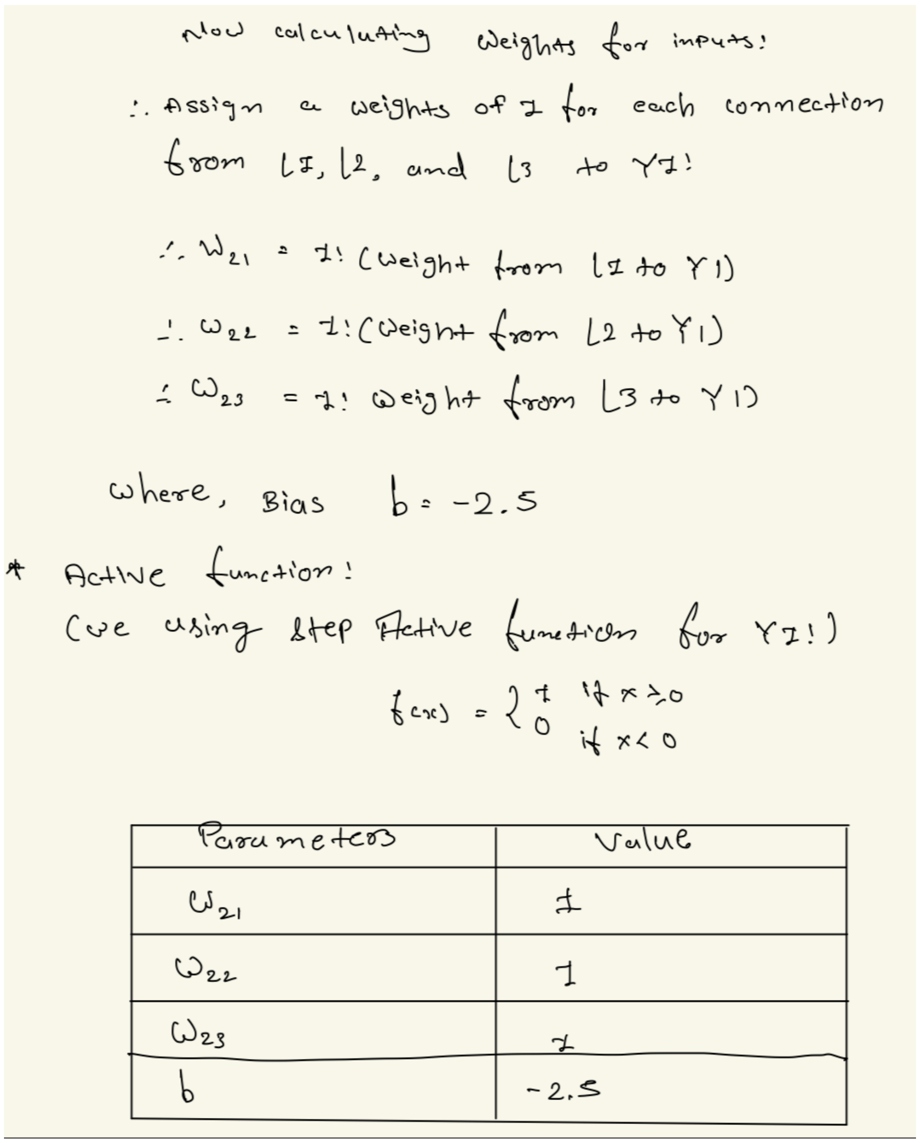
To define the yellow region (the triangular boundary in Figure (a)), the output perceptron Y1 should activate (output 11) when all of the following are true:

- The point satisfies L1=1,
- The point satisfies L2=1,
- The point satisfies L3=1.

Thus, Y1 should output 1 only when all three conditions are met, which resembles an **AND operation**.

**Setting Parameters for Perceptron Y1**

We can represent an AND operation by assigning positive weights to the outputs of L1, L2, and L3, and setting a threshold for the perceptron to activate only when all inputs are 1.



now calculating weights for inputs!

∴ Assign a weights of 1 for each connection from L1, L2, and L3 to Y1!

∴ $w_{21}$ = 1! (weight from L1 to Y1)

∴ $w_{22}$ = 1! (weight from L2 to Y1)

∴ $w_{23}$ = 1! weight from L3 to Y1)

where, Bias $b = -2.5$

* Active function!

(we using step Active function for Y1!)

$$f(x) = \begin{cases} 1 & \text{if } x \geq 0 \\ 0 & \text{if } x < 0 \end{cases}$$

| Parameters | Value |
|---|---|
| $w_{21}$ | 1 |
| $w_{22}$ | 1 |
| $w_{23}$ | 1 |
| b | -2.5 |

**Explanation**

The perceptron Y1 effectively combines the outputs of L1, L2, and L3 to identify the triangular region. By using an AND logic through weighted summation and a threshold, Y1 outputs 1 only when the point lies within the boundary region defined by the intersections of L1, L2, and L3. This approach demonstrates how multiple linear boundaries can be combined in a two-layer network to approximate complex decision boundaries, as per the Universal Approximation Theorem.

**4) [5 pts] Show the classification values of input (x1=2, x2=2) and (x1=4, x2=1), respectively. Show the derivations.**

To classify the input points (x1=2,x2=2) and (x1=4,x2=1), we need to evaluate them through each perceptron L1, L2, L3, and finally Y1 to determine if they fall within the yellow region.

Perceptron Equations and Conditions Recap

Each perceptron L1, L2 and L3 is associated with a line in figure(a) and has the following eqⁿ.

L1: $x+y-4=0 \geq 2$, output = 1 otherwise 0;

L2: $-0.5x+y=0 \geq 0$, output = 1 otherwise 0;

L3: $0.5x+y-2=0 \geq 2$, output = 1 Otherwise 0;

So,

$Y1 = 1\cdot L1 + 1\cdot L2 + 1\cdot L3 - 2.5$

output 1 if L1 = 1, L2 = 1, and L3 = 1;

otherwise 0

* Classification for Input ($x_1 = 2$, $x_2 = 2$)

1. L1 Output:

$L1 = 2+2-4 = 0$

$\therefore$ output = 1

Likewise, L2 = 1, output 1

L3 = 1, output 1

Now calculate Y1 output!

$Y1 = 1 \cdot 1 + 1 \cdot 1 + 1 \cdot 1 - 2.5$

$= 3 - 2.5$

$= \underline{0.5}$

$\therefore$ output $= 1$

So, for $(x_1 = 2, x_2 = 2)$ the classification output $= 1$

* classification for Input $(x1 = 4, x2 = 1)$

L1: output 1

L2: output 0

L3: output 1

Here, $L1 = 1$, $L2 = 0$, $L3 = 1$ since L2 outputs 0, the point does not meet all conditions to be in the yellow region

* Calculating Y1 output!

$Y1 = 1 \cdot 1 + 1 \cdot 0 + 1 \cdot 1 - 2.5$

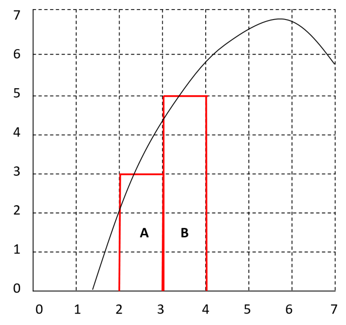
$= 1 + 0 + 1 - 2.5$

$= -0.5$ $\therefore$ output $= 0$

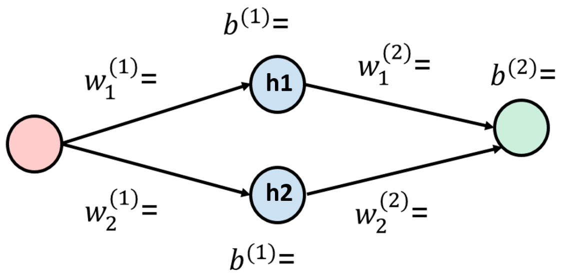
$\therefore$ for $(x_1 = 4, x_2 = 1)$, the classification O/P = 0

**6. (Refer to p. 11-13 in the slides) Universal Approximation Theorem (Regression)**
**It is also known that a neural network with one layer can represent any continuous (regression) functions.**

Refer to the following function and a one layer neural network. Activation function of hidden nodes is sigmoid and output node uses linear activation function.
We want to approximate the function in figure (a) using red bars (e.g., A and B).



(a)



(b)

**1) [4 pts/ea] Now we want to implement the red bar "A" using a neural network in figure (b). Show all the parameters of hidden nodes h1 & h2 (e.g., the weights, bias, slope of curve), respectively. Show the exact shape of output of each hidden node.**

To approximate the function depicted in Figure (a) using the red bar "A" with the neural network shown in Figure (b), we need to establish two hidden nodes, h1 and h2, that activate in the suitable region, thereby creating the desired rectangular shape. Since the activation function for the hidden nodes is sigmoid and the output node employs a linear activation function, we will configure the parameters (weights and biases) of h1 and h2 to achieve this shape.

**Setting Parameters for h1 and h2**

We will configure h1 to activate around x=2 (left edge of bar "A") and h2 to deactivate around x=3 (right edge of bar "A"). This setup will help create a rectangular activation in the region defined by bar "A."

1. **Hidden Node h1** (activates near x=2):

↳ Weight $\omega_1^{(1)}$ : Positive value to control the slope, for e.g, $\omega_1^{(1)} = 10$

↳ Bias $b_1^{(1)}$ : set the bias to shift the activation to around $x=2$!

$$b_1^{(1)} = -2 \times \omega_1^{(1)} = -20$$

↳ o/p shape of h1: The Sigmoid function for h1 will have a steep rise centered near $x=2$, transitioning from 0 to 1

2. **Hidden Node h2** (deactivates near x=3x=3):

Weight $\omega_2^{(1)}$ : set as a negative value to create a descending slope, so $\omega_2^{(1)} = -10$

Bias $b_2^{(1)}$ : set the bias to shift the activation down around $x=3$!

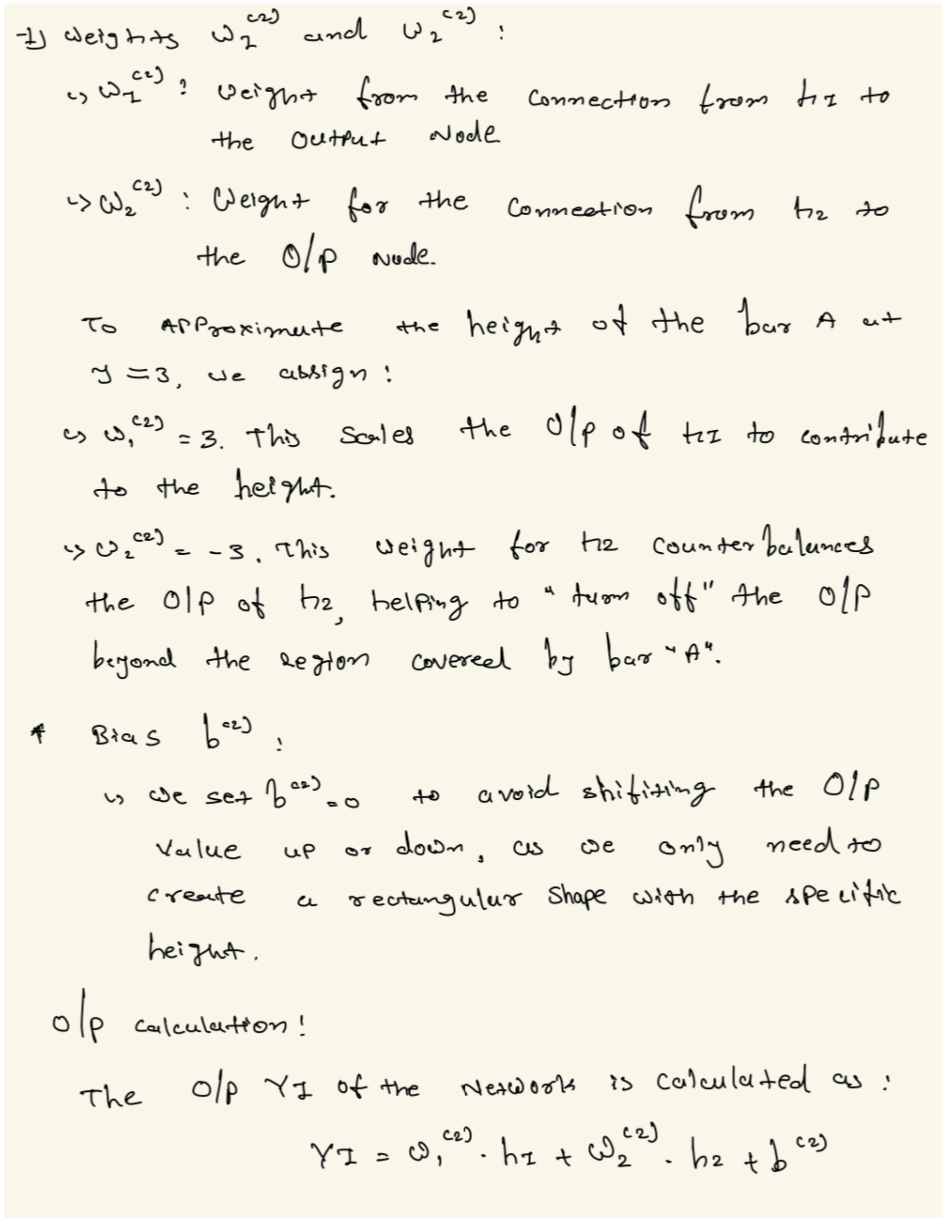
$$b_2^{(1)} = 3 \times \omega_2^{(1)} = 30$$

O/P shape of h2 : The sigmoid function for h2 will decrease from 1 to 0 around $x=3$, effectually creating the right edge of bar "A."

**2) [4 pts] Show all the parameters of output node. Explain in detail how the function is implemented using this network.**

**Output Node Parameters**

The output node receives two inputs from the hidden nodes, h1 and h2. Since the output node employs a linear activation function, the output is a weighted sum of h1 and h2, along with a bias. Let's define the parameters accordingly.



1) weights $\omega_1^{(2)}$ and $\omega_2^{(2)}$:

↳ $\omega_1^{(2)}$: weight from the connection from $h_1$ to the output node

↳ $\omega_2^{(2)}$: weight for the connection from $h_2$ to the O/P node.

To approximate the height of the bar A at $y = 3$, we assign:

↳ $\omega_1^{(2)} = 3$. This scales the O/P of $h_1$ to contribute to the height.

↳ $\omega_2^{(2)} = -3$. This weight for h2 counterbalances the O/P of $h_2$, helping to "turn off" the O/P beyond the region covered by bar "A".

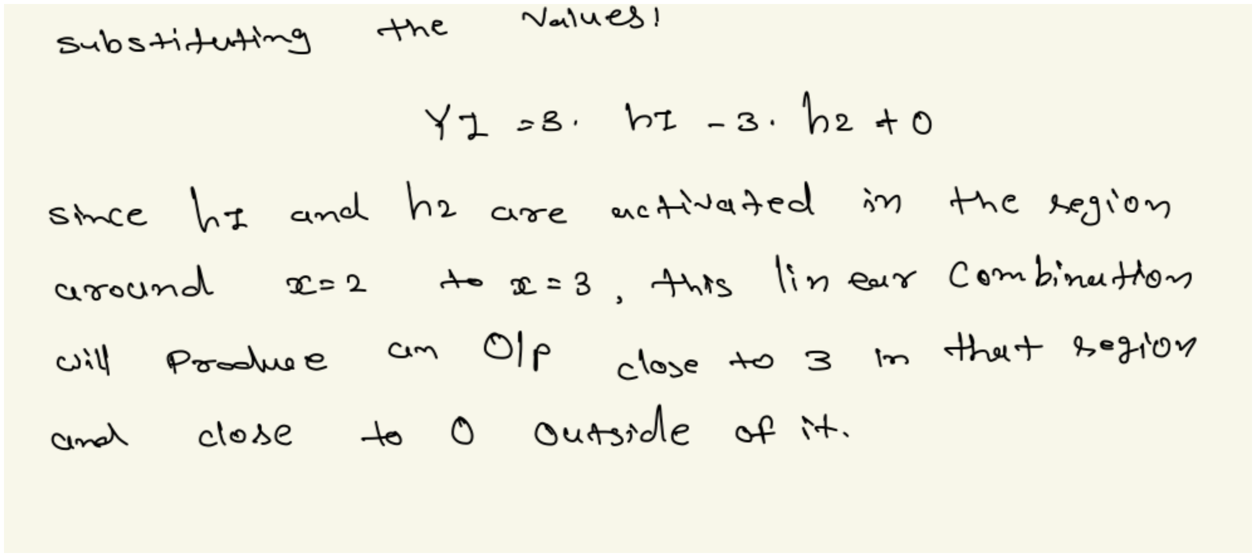
* Bias $b^{(2)}$:

↳ we set $b^{(2)} = 0$ to avoid shifiting the O/P value up or down, as we only need to create a rectangular shape with the specific height.

O/P calculation:

The O/P $Y_1$ of the Network is calculated as:

$$Y_1 = \omega_1^{(2)} \cdot h_1 + \omega_2^{(2)} \cdot h_2 + b^{(2)}$$


substituting the values!

$$Y1 = 3 \cdot h1 - 3 \cdot h2 + 0$$

since h1 and h2 are activated in the region around $x=2$ to $x=3$, this linear combination will produce an O/p close to 3 in that region and close to 0 outside of it.

**3) [4 pts] Show the output value of network given an input 2.5.**

To determine the output value of the network for an input x=2.5, we first calculate the activations of the hidden nodes h1 and h2. Subsequently, we use these values to compute the final output Y1.

1) Hidden Node h1

for h1, the activation function is sigmoid, and the weights and bias for h1 are set to make it activate near $x = 2$.

The I/p $h_1 = \omega_1^{(1)} \times x + b_1^{(1)}$

Now, $\omega_1^{(1)} = 10$ & $b_1^{(1)} = -20$!

I/p $h_1 = 10 \times 25 - 20 = 25 - 20 = 5$

The o/p of h1 using the Sigmoid activation function is

$$h_1 = \sigma(5) = \frac{1}{1+e^{-5}} \approx 0.9933$$

2) Node h2

The I/p h2 is ! $\omega_2^{(1)} \cdot x + b_2^{(1)}$

substituting $\omega_2^{(1)} = -10$ and $b_2^{(1)} = 30$!

I/p $h_2 = -10 \cdot 2.5 + 30$

$= -25 + 30$

$= 5$

$\therefore \quad h_2 = \sigma(5) = \frac{1}{1+e^{-5}} \approx 0.9933$

$Y_I = \omega_1^{(2)} \cdot h_I + \omega_2^{(2)} \cdot h_2 + b^{(2)}$

$\omega_1^{(2)} = 3, \ \omega_2^{(2)} = -3$, and $b^{(2)} = 0$!

$$Y_I = 3 \cdot 0.9933 - 3 \cdot 0.9933 + 0$$
$$= 2.9799 - 2.9799$$
$$= 0$$

the O/P value of network for an input $x = 2.5$

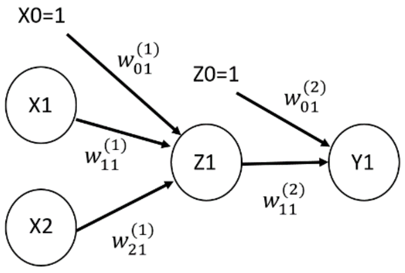
is $Y_I \simeq 0$

**7. Backpropagation algorithm**

**With the following neural network, we perform backpropagation using stochastic gradient descent.**

**Input data are x = [ [1 0] [0 1] [1 1] ] and its corresponding target y=[0 1 1]. All weight values are initialized to 0.1 and learning rate is 0.2.**

**We use sum of squared error (not mean of squared error) as the error function($E_d$). Each node in hidden/output has two functions $net_j$ and $o_j$ (refer to p. 22 in the slides) and $o_j$ (activation function) is the sigmoid function.**



**1) [6 pts] Given x = [1 0] and y = 0, Compute the output of Z1 and Y1. Show the derivations.**

To compute the output of Z1 and Y1 in the given neural network for the input $x = [1, 0]$ and target $y = 0$

∴ calculate the net Input to Z1

∴ calculate the net Input to $Y_1$

Now, find the $O_{Z1}$ & $O_{Y1}$

↳ The given data,

Input $x = [1, 0]$

Target $y = 0$

Initial weights:

* $W_{01}^{(1)} = 0.1$ (bias weight of Z1)
* $W_{11}^{(1)} = 0.1$ (Weight from X1 to Z1)
* $W_{21}^{(1)} = 0.1$ (Weight from X2 to Z1)
* $W_{01}^{(2)} = 0.1$ (bias weight for Y1)
* $W_{11}^{(2)} = 0.1$ (Weight from Z1 to Y1)

$\eta = 0.2$

function = $\sigma(x) = \dfrac{1}{1+e^{-x}}$

Now, compute the output Z1

$\therefore net_{z1} = W_{01}^{(1)} \cdot x0 + \omega_{11}^{(1)} \cdot x1 + W_{21}^{(1)} \cdot x2$

substitutes the value.

$net_{z1} = (0.1 \cdot 1) + (0.1 \cdot 1) + (0.1 \cdot 0)$

$= 0.1 + 0.1 + 0$

$= 0.2$

$\therefore O_{z1} = \sigma(net_{z1})$

$= \dfrac{1}{1+e^{-0.2}} \approx 0.5498$

Now, Y1

$\therefore net_{y1} = \omega_{01}^{(2)} \cdot 20 + \omega_{11}^{(2)} \cdot O_{z1}$

Put the value,

$net_{y1} = (0.1 \cdot 1) + (0.1 \cdot 0.5498)$

$= 0.1 + 0.05498$

$= 0.15498$

$$O_{Y_1} = 6(net_{Y_1}) = \frac{1}{1 + e^{-0.15498}}$$

$$\approxeq 0.5387$$

$\therefore Z_1(O_{Z_1}) : \approxeq 0.5498$

$\therefore Y_1(O_{Y_1})! \approxeq 0.5387$

**2) [4 pts] Show the chain rule formula of $\frac{\partial E_d}{\partial w_{11}^{(2)}}$**

chain Rule for $\frac{\partial E_d}{\partial \omega_{11}^{(2)}}$

we use the chain rule to break down this partial derivative:

$$\frac{\partial E_d}{\partial \omega_{11}^{(2)}} = \frac{\partial E_d}{\partial o_{Y_1}} \cdot \frac{\partial o_{Y_1}}{\partial net_{Y_1}} \cdot \frac{\partial net_{Y_1}}{\partial \omega_{11}^{(2)}}$$

Now compute $\frac{\partial E_d}{\partial o_{Y_1}}$

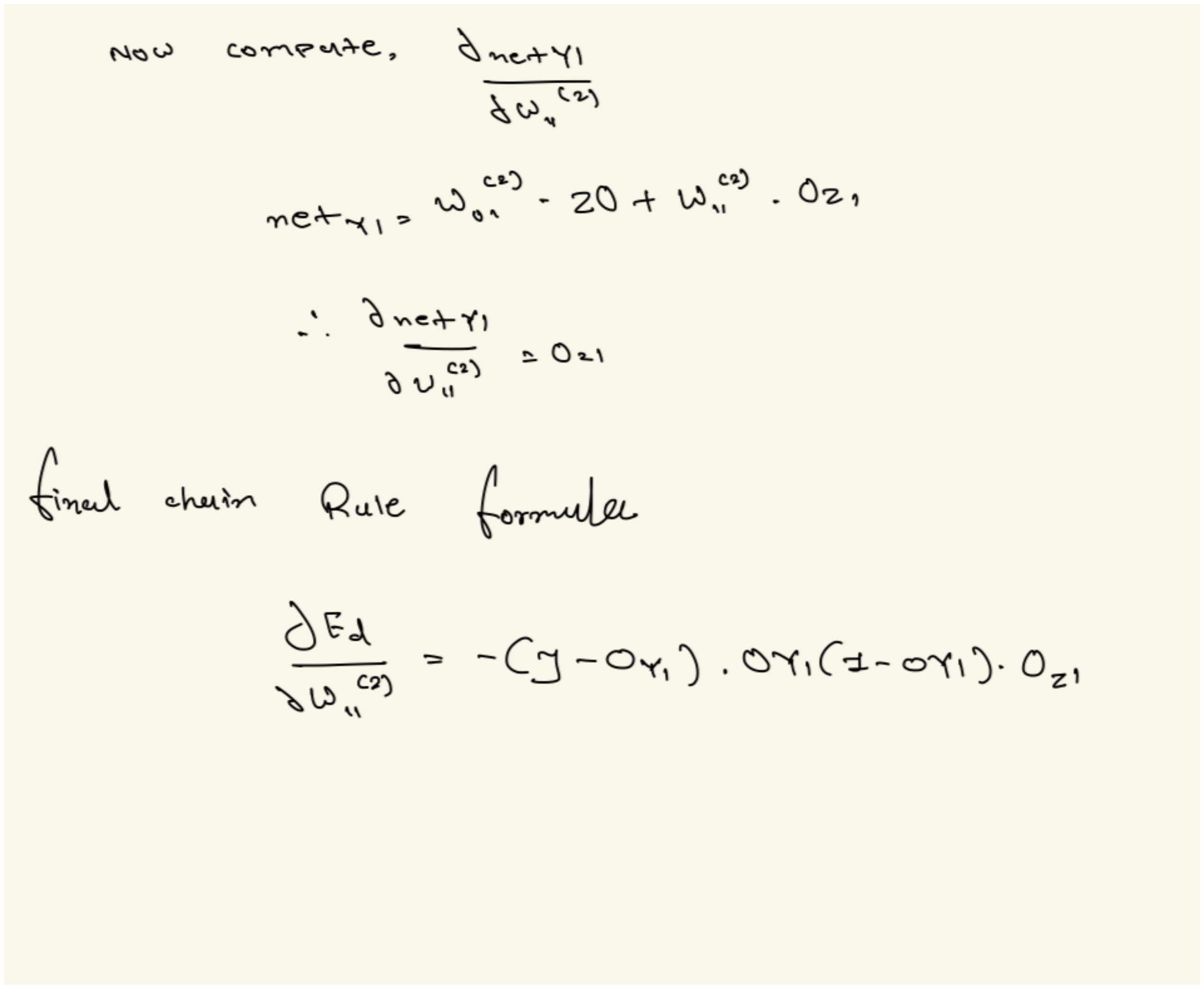
$$\frac{\partial E_d}{\partial o_{Y_1}} = -(y - o_{Y_1}), \quad E_d = \frac{1}{2}(y - o_{Y_1})^2$$

$\therefore$ Now, $\frac{\partial o_{Y_1}}{\partial net_{Y_1}}$

$$o_{Y_1} = \sigma(net_{Y_1}) = \frac{1}{1+e^{-net_{Y_1}}}$$

so,

$$\frac{\partial o_{Y_1}}{\partial net_{Y_1}} = o_{Y_1}(1 - o_{Y_1})$$



Now compute, $\frac{\partial net_{Y1}}{\partial w_{11}^{(2)}}$

$$net_{Y1} = w_{01}^{(2)} \cdot Z0 + w_{11}^{(2)} \cdot O_{Z1}$$

$$\therefore \frac{\partial net_{Y1}}{\partial v_{11}^{(2)}} = O_{Z1}$$

final chain Rule formula

$$\frac{\partial E_d}{\partial w_{11}^{(2)}} = -(y - O_{Y_1}) \cdot O_{Y1}(1 - O_{Y1}) \cdot O_{Z1}$$

**3) [6 pts] Using the value of Z1 and Y1, compute $\delta_{y1}$ (refer to p. 26)**

We use the given formula

$$\delta_{Y_1} = (y - OY_1) \cdot OY_1 \cdot (1 - OY_1)$$

Where,

↳ $y$ is the target output for current Input

↳ $OY_1$ is the actual output from the neuron $Y1$

↳ $OY_1 \cdot (1 - OY_1)$ is the derivative of the sigmoid activation function used in $Y1$

* Given values.

for input $x = [1, 0]$ and target $y = 0$

so, $OY_1 \approx 0.5387$

Now, $y - OY_1$ :

$$y - OY_1 = 0 - 0.5387$$
$$= -0.5387$$

Now, $OY_1 \cdot (1 - OY_1)$

$$\therefore OY_1 \cdot (1 - OY_1) = 0.5387 \cdot (1 - 0.5387)$$
$$\approx 0.5387 \cdot 0.4613 \approx 0.2489$$

Now,

$$\delta Y_1 = (y - OY_1) \cdot OY_1 \cdot (1 - OY_1)$$

$$= -0.5387 \cdot 0.2489$$

$$\cong -0.134$$

So, $\delta Y_1 \cong -0.134$

**4) [4 pts] Update $w_{11}^{(2)}$**

using gradient descent rule:

$$W_{11}^{(2)} \leftarrow W_{11}^{(2)} + \eta \cdot \delta_{Y_I} \cdot OZ_I$$

where,

↳ $W_{11}^{(2)}$ is the weight connecting the hidden node $Z_I$ to the O/P Node $Y_I$

↳ $\eta$ is learning rate which is given 0.2

↳ $\delta_{Y_1}$ is the error term for the O/P node $Y_I$

↳ $OZ_1$ o/p node of the hidden node $Z_I$

Now, $W_{11}^{(2)} \leftarrow W_{11}^{(2)} + 0.2 \cdot (-0.134) \cdot (0.5498)$

$0.2\ (0.134) \cdot 0.5498 \approx -0.0147$

initial weight was 0.1

$W_{11}^{(2)} \leftarrow 0.1 - 0.0147 = 0.0853$

So, $W_{11}^{(2)} \simeq 0.0853$

**5) [Bonus  8 pts] Repeat above process for updating $w_{21}^{(1)}$**

Now, compute $\delta_{21}$

$$\delta_{21} = O21 \cdot (1 - O21) \cdot \sum \delta_{Y_1} \cdot W_{11}^{(2)}$$

where,

$O_{21}$ is the output of 21, (we alredy calculated)

$\delta_{Y_1} = -0.134$

$W_{11}^{(2)} = 0.0853$

Now, Substitute the value

$$\delta_{21} = 0.5498 \cdot (1 - 5498) \cdot (-0.134) \cdot 0.0853$$

Now, calculate $O2_1 \cdot (1 - O2_1)$ :

$\therefore O2_1 \cdot (1 - O2_1)$

$\therefore 0.5498 \cdot 0.4502 \simeq 0.2474$

Now, $\delta_{21}$ :

$$\delta_{21} = 0.2474 \cdot (-0.134) \cdot 0.0853 \simeq -0.0028$$

now, $\omega_{21}^{(1)}$

using the gradient descent update rule:

$$w_{21}^{(1)} \leftarrow \omega_{21}^{(1)} + \eta \cdot \delta_{21} \cdot x_2$$

where $\eta = 0.2$

$\delta_{21} \approx -0.0028$

$x_2 = 0$

$$w_{21}^{(1)} \leftarrow w_{21}^{(1)} + (0.2) \cdot (-0.0028) \cdot 0$$

$$= w_{21}^{(1)}$$

$\therefore$ The weight $w_{21}^{(1)}$ remains unchanged in the update step because the input $x2 = 0$

$\therefore$ $w_{21}^{(1)}$ remain $0.1$

**8. Given the image and filter**

| | | | | |
|---|---|---|---|---|
| **2** | **4** | **5** | 9 | 7 |
| **6** | **3** | **1** | 4 | 6 |
| **4** | **2** | **7** | 7 | 2 |
| 6 | 5 | 9 | 9 | 8 |
| 8 | 4 | 3 | 6 | 5 |

| | | |
|---|---|---|
| 2 | 3 | 1 |
| 1 | 0 | 4 |
| 4 | 8 | 4 |

**1) [4 pts] Show the output of feature map node using red region and filter (bias=0) using ReLU activation function**

Given data!

→ Red region of the image!

$$\begin{bmatrix} 2 & 4 & 5 \\ 6 & 3 & 1 \\ 4 & 2 & 7 \end{bmatrix}$$

Filter !

$$\begin{bmatrix} 2 & 3 & 1 \\ 1 & 0 & 4 \\ 4 & 8 & 4 \end{bmatrix}$$

RelU Activation Function

$$\text{Relu}(x) = \max(0, x)$$

Now, we calculating each term

$2 \times 2 = 4$

$4 \times 3 = 12$

$\vdots$

$7 \times 4 = 28$

Adding this result

$$4+12+5+6+0+4+16+16+28 = 91$$

**2) [3 pts] Compute the size of feature map. (stride=2, no padding).**

Now, Applying ReLU Activation

Bias = 0

$$\text{ReLU}(91) = \max(0, 91) = \underline{91}$$

Input size (Image) = 5 × 5

Filter size: 3 × 3

Stride: 2

Padding: 0

∴ Formula for O/P size:

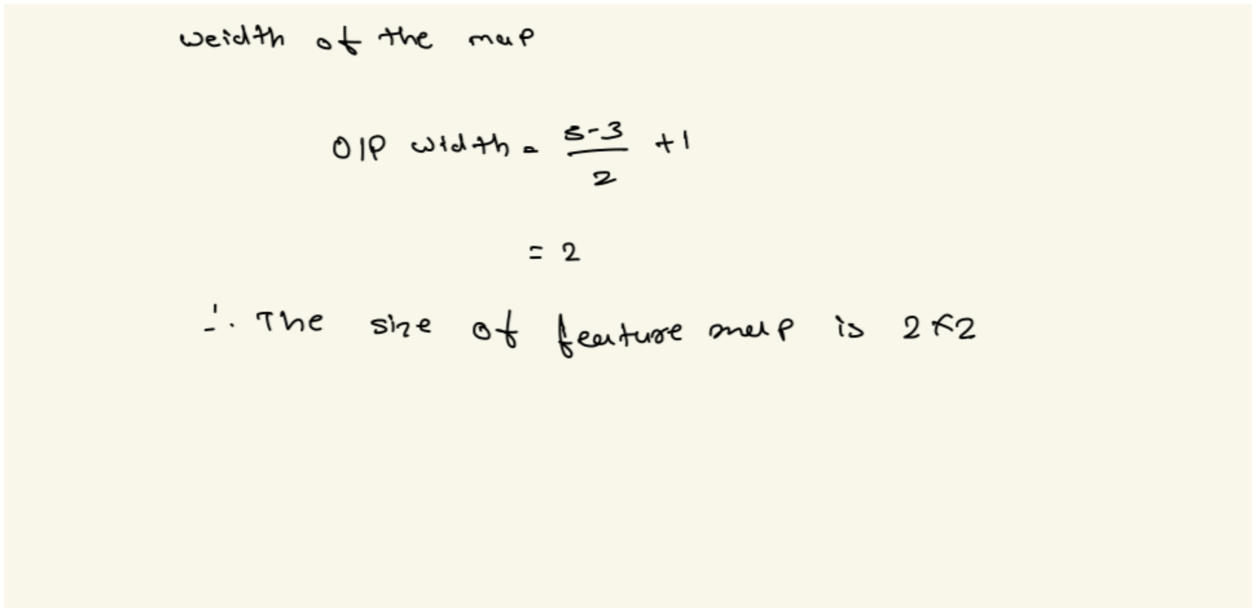
$$\text{O/P size} = \frac{\text{I/P size} - \text{Filtersize}}{\text{Stride}}$$

Now calculate:

Height of Feature map:

$$\text{O/P height} = \frac{5-3}{2} + 1$$

$$= 2$$



weidth of the map

$$O/P \text{ width} = \frac{5-3}{2} + 1$$

$$= 2$$

$\therefore$ The size of feature map is $2 \times 2$

**3) [3 pts] When there are 10 feature maps, what is the total number of parameters?**

To determine the total number of parameters, we must account for the parameters associated with each filter (or kernel) and any associated biases, considering the presence of 10 feature maps.

- Given the information provided, each feature map is generated by a filter. The filter size is 3x3, and there is a bias term for each feature map. The total number of feature maps is 10.

**Calculating Parameters for One Feature Map**

1. **Parameters in the Filter**: Each filter has 3×3=9 weights.
2. **Bias**: Each feature map has one bias term, so there is 1 additional parameter per feature map.

Thus, the total parameters for one feature map are: 9+1=10

**Total Parameters for 10 Feature Maps**

Since there are 10 feature maps, the total number of parameters is: 10×10=100

The total number of parameters when there are 10 feature maps is: 100

**9. We have the following multi-channel data.**

| channel 1 | | | channel 2 | | | channel 3 | | | channel 4 | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 3 | 1 | 1 | 9 | 1 | 5 | 3 | 1 | 4 | 5 | 1 |
| 1 | 0 | 4 | 6 | 1 | 2 | 1 | 0 | 4 | 1 | 7 | 1 |
| 4 | 1 | 4 | 2 | 8 | 3 | 4 | 2 | 4 | 4 | 8 | 4 |

**1) [3 pts] Show an example of 1X1 convolution filter.**

Given the multichannel data.

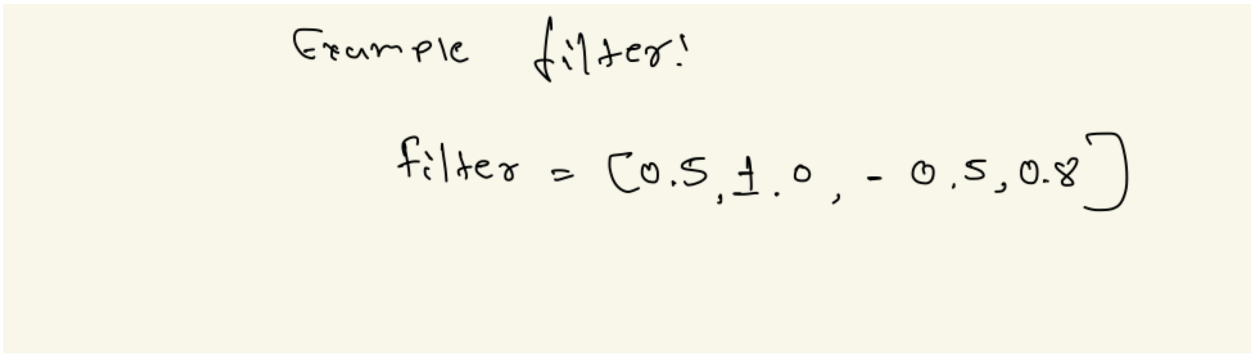
Each channel has 3×3 matrix

channel 1:
$$\begin{bmatrix} 2 & 3 & 1 \\ 1 & 0 & 4 \\ 4 & 1 & 4 \end{bmatrix}$$

channel 2 :
$$\begin{bmatrix} 1 & 9 & 1 \\ 6 & 1 & 2 \\ 2 & 8 & 3 \end{bmatrix}$$

channel 3:
$$\begin{bmatrix} 5 & 3 & 1 \\ 1 & 6 & 4 \\ 4 & 2 & 4 \end{bmatrix}$$

channel 4:
$$\begin{bmatrix} 4 & 5 & 1 \\ 1 & 7 & 1 \\ 4 & 8 & 4 \end{bmatrix}$$



Example filter:

filter = [0.5, 1.0, - 0.5, 0.8]

**2) [3 pts] compute the output of upper left corner (red) position using above 1X1 filter.**

To illustrate a 1×1 convolution filter, we require a filter that covers a single element in each channel. Since this involves multi-channel data, a 1×1 filter would possess a weight for each channel it applies to.

The data has 4 channels, so a 1×1 filter would have 4 weights (one for each channel) and potentially a bias term.

The values in the top-left corner for each channel are:

- **Channel 1**: 2
- **Channel 2**: 1
- **Channel 3**: 5
- **Channel 4**: 4

The output for this position would be:

Output = (2*0.5)+(1*1.0)+(5*(−0.5))+(4*0.8)

Calculating each term:

Output= 1.0 + 1.0 − 2.5 + 3.2 = 2.7

The output value at this position using the 1×1 convolution filter is: 2.7

**3) [3 pts] What is the advantage of using 1X1 convolution. Compare it with traditional Convolutional filter.**

**Advantages of Using 1×1 Convolution**:

1. **Dimensionality Reduction**:
    - A 1×1 convolution can reduce the number of channels in the feature map. For example, applying multiple 1×1 filters can reduce a high-dimensional input to a lower-dimensional output, which helps control model complexity and reduce computational load.

2. **Inter-Channel Mixing**:
    - The 1×1 convolution allows learning a weighted combination of channels, which can effectively capture relationships across channels (for example, mixing color channels or feature channels in CNNs). This can be useful for creating new features that capture cross-channel information.
3. **Increased Network Depth**:
    - By adding 1×1 convolutions, we can increase the depth of the network without significantly increasing the computational cost. This allows the network to learn more complex representations and can enhance model performance.

**Comparison with Traditional Convolutional Filters**:

- **Traditional Convolutional Filters** (e.g., 3×3, 5×5) capture spatial features, detecting patterns like edges and textures by looking at neighboring pixels. These are essential for capturing spatial relationships in images.
- **1×1 Convolutions**, in contrast, do not capture spatial relationships (since they operate on individual pixels across channels) but are useful for channel-wise transformations and dimensionality reduction.

In essence, 1×1 convolutions are advantageous for dimensionality reduction, inter-channel mixing, and increasing network depth. On the other hand, traditional convolutions are crucial for capturing spatial patterns.

**10. [5 pts] In convolutional layer, one issue is to determine the proper filter size. Can we find the proper filter size automatically in CNN? Explain one possible way of computing filter size automatically.**

In Convolutional Neural Networks (CNNs), choosing the appropriate filter size is paramount because it influences the network's capacity to extract pertinent spatial features from the input. Historically, filter sizes were manually selected (e.g., 3x3 or 5x5), but there are methods to automatically determine the optimal filter size. One such approach involves Neural Architecture Search (NAS).

**Neural Architecture Search (NAS)**
**Explanation**: Neural Architecture Search (NAS) is an automated approach that systematically explores various network configurations to discover optimal neural network architectures, particularly in Convolutional Neural Networks (CNNs). NAS algorithms systematically investigate different aspects of network design, including the number, size, and shape of filters, as well as other hyperparameters, to identify the architecture that performs the best on a specific task.

**How it Works**:

1. Define a range of possible filter sizes (e.g., 1×1, 3×3, 5×5, etc.) along with other parameters, such as the number of layers and neurons.
2. **Search Algorithm**: Use search strategies such as **Reinforcement Learning (RL)**, **Evolutionary Algorithms**, or **Gradient-Based Optimization** to explore different combinations of filter sizes and network structures.
3. **Evaluation**: Each candidate architecture (including its filter size) is trained and evaluated on a validation dataset. Performance metrics like accuracy or loss are used to determine the effectiveness of each configuration.

4. **Optimization**: The search algorithm iteratively adjusts the filter sizes and other parameters to optimize the network's performance, ultimately identifying the optimal filter size automatically.

**Example**: Suppose a NAS algorithm is applied to a CNN for image classification. Initially, it experiments with different filter sizes and combinations. After several iterations, the algorithm discovers that 3×3 filters in the first layer and 5×5 filters in the second layer provide the highest accuracy on the validation set.

**Advantages of NAS for Filter Size Selection**

- **Efficiency**: NAS can automatically optimize filter sizes and other hyperparameters, potentially saving time and reducing the need for manual experimentation.
- **Performance**: By identifying optimal filter sizes, NAS can enhance the network's ability to learn useful features, improving accuracy and generalization.

**11. Suppose you developed a CNN network that successfully identifies the images of 'dog' and 'cat'. Now we want to make the neural network classify the images of 'duck' as well.**
**1) [3 pts] Explain the approach of building the CNN from scratch.**

To construct a Convolutional Neural Network (CNN) from the ground up for the purpose of identifying images of 'dog,' 'cat,' and now 'duck,' the following steps would generally be followed:

1. **Data Collection**: Gather a labeled dataset comprising images of 'dog,' 'cat,' and 'duck.' Ensure the dataset is substantial and varied to facilitate the model's ability to generalize effectively to novel images.
2. **Network Architecture Design**: Design a CNN architecture specifically designed for image classification. This architecture should incorporate multiple convolutional layers, pooling layers, and fully connected layers. The depth of the architecture should be sufficient to extract intricate features such as edges, shapes, and textures that distinguish between 'dog,' 'cat,' and 'duck.'
3. **Training the Network**: Initialize the CNN weights and train the model from scratch using the collected dataset. This process involves forward propagation to calculate predictions and backpropagation to update the weights based on the error between the predicted values and the actual labels.
4. **Hyperparameter Tuning**: Optimize hyperparameters like the learning rate, batch size, and number of epochs to enhance the model's performance.
5. **Evaluation**: Evaluate the model on a separate test set to ensure that it has learned to generalize well and can accurately classify new images of 'dog,' 'cat,' and 'duck.'

This process can be very time-consuming and computationally intensive, especially if starting with a complex CNN architecture.

**2) [6 pts] Now in transfer learning approach:**
2-1) Explain source domain and target domain in this approach.

- **Source Domain**: In transfer learning, the source domain refers to the initial training domain of the model. For instance, it could be a pre-trained Convolutional Neural Network (CNN) model that was originally trained on a vast dataset like ImageNet, which encompasses a diverse range of categories, such as ‘dog’ and ‘cat’ (and possibly other animals, although not specifically ‘duck’).

- **Target Domain**: The target domain is the domain where the model will be applied or fine-tuned. In this instance, the target domain encompasses images of ‘dog,’ ‘cat,’ and ‘duck.’ The objective is to adapt the source model to classify images within this target domain, including the newly introduced ‘duck’ category.

2-3) Explain the process of applying transfer learning in this task.

To apply transfer learning, we would first select a pre-trained Convolutional Neural Network (CNN) model, such as VGG or ResNet, that has already been trained on a vast dataset like ImageNet. This model has learned to recognize general features from the training data. Then, we would fine-tune this pre-trained model on our specific task by providing it with a smaller dataset that contains the relevant information for our problem. This process allows the model to leverage the knowledge it has acquired from the pre-training phase and adapt it to the new task.

1. **Load the Pre-trained Model**: Utilize a pre-trained CNN model, specifically one that has been trained on the ImageNet dataset or a comparable dataset. The initial layers of this model have been adept at recognizing fundamental features such as edges and textures, which are highly relevant and useful across a wide range of image classification tasks.
2. **Freeze Initial Layers**: Freeze the weights of the initial layers in the pre-trained model. These layers contain general feature detectors that don’t require retraining because they’re not specific to ‘dog,’ ‘cat,’ or ‘duck.’
3. **Replace and Fine-tune the Final Layers**: Replace the final layers of the model (typically the fully connected layers) with new layers specifically designed for our task. In this instance, replace the output layer with a new layer having three output units, one for each class (‘dog,’ ‘cat,’ ‘duck’).
4. **Train the Final Layers**: Train the new layers with the target dataset (images of 'dog,' 'cat,' and 'duck'). Optionally, fine-tune some of the later convolutional layers as well to adapt the model to the specific characteristics of the new dataset.
5. **Evaluation**: Evaluate the modified model on a test set containing 'dog,' 'cat,' and 'duck' images to ensure it can accurately classify all three categories.

3) [3 pts] Explain whether approach in 2) is better than approach 1).

**Yes, the transfer learning approach is generally superior to constructing a CNN from scratch, particularly for this task, for several reasons:**

1. **Reduced Computational Cost and Time**: Transfer learning capitalizes on the knowledge acquired from a substantial dataset, significantly reducing the need for extensive computational resources and time. Training a Convolutional Neural Network (CNN) from the ground up would necessitate substantial computational power, especially when dealing with limited data.
2. **Better Performance with Limited Data**: Pre-trained models have acquired generic features that transfer well to the target domain, even when the target dataset is limited. However, training from scratch with limited data may lead to poor generalization, as the model may not effectively learn the necessary features.
3. **Improved Generalization**: Since the pre-trained model has already been trained on a vast dataset, it possesses a superior capacity to generalize. The initial layers identify fundamental features that are applicable across numerous images, enabling transfer learning to effectively adapt these features to novel tasks.

**<u>* For coding assignments:</u>**

1) Don’t change the basic program code (unless you have my permission).
2) Never use any ready-made library to implement algorithms (except for bonus questions)
3) Don’t put entire program in one cell in Jupyter. Instead, for each sub questions, show the corresponding program code in .ipynb and explain it.
4) Don’t copy from other sources
5) HWs violating these guidelines will get zero.

12. (coding) perceptron

```
# For each subquestion, show the program code and explain it in .ipynb

import numpy as np
import matplotlib.pyplot as plt
import random
import pandas as pd

#import dataset
df = pd.read_csv('iris.csv', header=None)

# select the first 100 rows with only the first 2 features (petal length & width)
# X=[[first feature value of 1st data  &  second feature value of 1st data] [.....]]
# y=[target values]
#     y value is -1 if target=setosa, y=-1, otherwise 1
# e.g.,
# X=[[5.1 1.4] [5.1 3. ] … [5.7 4.1]]
# y=[-1 -1  … -1  1 … 1]
#
‘’’
[2 pts]
YOUR WORK HERE: 12-1)
# do the following:
# create X and y as above
# show the first 5 contents of X and y
 ‘’’

class Perceptron():
   # initialize learning rate and number of iterations
   def __init__(self, lrate=0.1, no_iter=50):
        self.lrate = lrate
        self.no_iter = no_iter

    # using X & y, update ww parameters
    def fit(self, X, y):
        # initialze weights ww to random value (-1,1)
        self.ww = [random.uniform(-1.0, 1.0) for _ in range(1+X.shape[1])] #randomly initialize weights
        # initilize list_error=[]
        # keeps track of the errors per iteration for graph plotting
        self.list_errors = []

        #iterate over labelled dataset updating weights for each features accordingly
        for _ in range(self.no_iter):
            cur_error = 0
            for xx, label in zip(X, y):
                '''
                [8 pts]
                YOUR WORK HERE: 12-2)
                # update weights using w_i ← w_i + Δw_i          where  Δw_i = η(t_i – o_i)x_i

                # Refer to p. 14-15
                # X: parameter array, y: label
                #
                # compute delta Δw_i
                # update self.ww
                # compute cur_error. cur_error is 0 if delta=0, 1 otherwise
                # append cur_error into list_errors
                '''
        return self

    #compute the net input i.e sum of X and the weights plus the bias value
    def net_input(self, X):
        return np.dot(X, self.ww[1:]) + self.ww[0]

    #predict a classification for a sample of features X
    def predict(self, X):
        '''
        [6 pts]
        YOUR WORK HERE: 12-3)
        # do the following
        # implement step function of perceptron

        # output =  1  if  ∑i=0 wixi >0 ,    = -1   otherwise

        # return 1 or -1 depending on ∑i=0 wixi
        '''

# create a new perceptron & initialize parameters
```

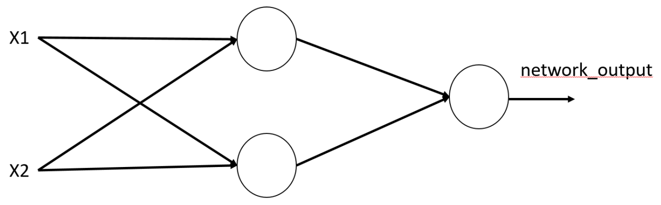
# you can change no_iter
# train perceptron using data X, y
model = Perceptron(no_iter=10)
model.fit(X, y)

‘’’
[4 pts]
YOUR WORK HERE: 12-3)
# plot errors against number of iterations using list_errors
‘’’

13. (coding) Backpropagation

# Don’t change the program code.
# For each subquestion, complete the code and explain it.

We have the following neural network.



Both hidden nodes and output node uses sigmoid function.

**(Program code)**

```
import numpy as np


def forward(X):
    # Feedforward
    # we assume there is a dummy input X0. Therefore, no need of bias notation.
    hidden_input = np.dot(X, hidden_weights)
    hidden_output = sigmoid(hidden_input)

    output_input = np.dot(hidden_output, output_weights)
    output_output = sigmoid(output_input)

    return hidden_output, output_output

# Activation function: Sigmoid
def sigmoid(x):
    ‘’’
    [2 pts]
    YOUR WORK HERE 13-1)
    # compute the value of sigmoid function and return it
    '''

# Derivative of Sigmoid
def sigmoid_derivative(x):
    '''
    [2 pts]
    YOUR WORK HERE 13-2)
    # compute the derivative of sigmoid function and return it
    '''

# XOR dataset is used as a training data
'''
[2 pts]
YOUR WORK HERE 13-3)
Instead of using bias, we assume there a dummy variable x0=1.
1) Change the following X data accordingly (in green)
2) change input_nodes values (in green)
'''
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])  # Input features (x1, x2)
Y = np.array([[0], [1], [1], [0]])  # Target values

#
# Construct a Neural Network
#
# number of nodes in each layer
input_nodes = 2  # Number of input nodes
hidden_nodes = 2  # Number of hidden nodes
output_nodes = 1  # Number of output nodes

# Randomly initialize weights and biases between (0,1)
# hidden_weights: weights connecting input layer nodes to hidden layer nodes
# output_weights: weights connecting hidden layer nodes to output node

hidden_weights = np.random.uniform(size=(input_nodes, hidden_nodes))
output_weights = np.random.uniform(size=(hidden_nodes, output_nodes))

# Learning rate. You can change this
lrate = 0.1

# Number of iteration. You can change this
no_iter = 10000

# Backpropagation training
for iter in range(no_iter):

    # forward step
    # hidden_output: outputs of hidden layer nodes
    # output_output: output of output layer node
    hidden_output, output_output = forward(X)

    # Compute error
    error = Y - output_output

    # Backpropagation

    '''
    [8 pts]
    YOUR WORK HERE 13-4)

    # Refer to p. 26
    # compute error
    # implement the following for output weight update
```

$$\Delta w_{ij} = \eta(t_j - o_j)o_j(1 - o_j)(output\ of\ hidden\ node\ i)$$
$$w_{ij} = w_{ij} + \Delta w_{ij}$$

```
    '''

    '''
    [10 pts]
    YOUR WORK HERE 13-5)
    # Refer to p. 30
    # since we have only one output node k=1 from the formula in p. 30
    # implement the following for hidden weight update
```

$$\Delta w_{ij} = -\eta o_j(1 - o_j).(-\delta_k\ w_{jk}).x_i$$
$$\delta_k = (t_j - o_j)o_j(1 - o_j)$$
$$w_{jk}: output\ weights$$

```
    '''

    # Print error every 100 iter for monitoring
    if iter % 100 == 0:
        print(f"Epoch {iter} Error is {np.mean(np.abs(error))}")


# predict using training data
_, nn_output = forward(X)
print(nn_output)
```

14. [Bonus 12 pts] Implement Question 13 with two output nodes (e.g., output_nodes = 2). Each output node represents the target value 0 and 1, respectively.

$$\Delta w_{ij} = -\eta o_j(1 - o_j).\left(\sum_{k \in parent(j)} -\delta_k\ w_{jk}\right)x_i$$