6.1

Assignment 6.1

Explain in your own words how a perceptron looks like, how it is used for classification, and how it can be learned.

A perceptron is a type of artificial neuron that mimics the basic functionality of a biological neuron. Visually, it consists of:

1. Inputs (x1,x2,..,xn): These are numerical features of the data you want to classify.
2. Weights (w1,w2,...,wn): Each input has an associated weight that represents its importance in the classification.
3. Bias (b): A constant added to the weighted sum of inputs to adjust the decision boundary.
4. Summation Function: Calculates the weighted sum of inputs and the bias
5. Activation Function: Applies a threshold (e.g., step function) to the summation to produce an output (0 or 1).

The perceptron essentially performs a linear transformation followed by a decision rule.

How a Perceptron is Used for Classification:

A perceptron is used for binary classification, where the goal is to separate data into two categories (e.g., 0 or 1, +1 or −1).

1. Decision Boundary: The perceptron defines a linear boundary (hyperplane) in the feature space.
2. Prediction:
    - Compute the weighted sum of the inputs plus the bias.
    - Apply the activation function:
        - If the sum is greater than or equal to a threshold (often 0), the output is 1 (one class).
        - Otherwise, the output is 0 (the other class).
3. The perceptron assumes the data is linearly separable; that is, a single straight line (or hyperplane) can separate the classes

How a Perceptron Learns:

The perceptron learns through an iterative process called the perceptron learning algorithm, which adjusts the weights and biases based on prediction errors. Here's how it works:

1. Initialize:
    - Start with random weights and bias values.
2. For Each Training Example:
    - Compute the perceptron's output:

$$y_{\text{predicted}} = \text{step}(\sum_{i=1}^{n} w_i x_i + b)$$

- Compare y predicted to the actual label y true.

3. Update Rule: If there is an error (y predicted ≠ y true):
   - Adjust weights:

$$w_i = w_i + \eta(y_{\text{true}} - y_{\text{predicted}})x_i$$

   - Adjust bias:

$$b = b + \eta(y_{\text{true}} - y_{\text{predicted}})$$

     Here, η is the learning rate, which controls the step size of the updates.

4. Repeat:
   - Go through the dataset multiple times until all examples are classified correctly (or for a fixed number of iterations).

6.2

Assignment 6.2

Consider one perceptron defined by the threshold expression $w_0 + w_1 x_1 + w_2 x_2 > 0$. All weights are initialized with 0, i.e. $w_0 = 0, w_1 = 0, w_2 = 0$. Given is the following data:

x1	**x2**	**t (target)**
1	0	0
3	-1	1
2	2	0
4	4	0
1	-2	1

Apply the perceptron rule (in batch mode) with $\eta = 0.1$ in order to perform two updates (two iterations) on the weights. What is your final perceptron? Draw its classification boundary in a diagram! How does it classify the instances?

Reminder of the perceptron rule:

$$w_i \leftarrow w_i + \eta \sum_d (t_d - o_d) x_{i,d}$$

6.2 Iteration 1

x_0	x_1	x_2	t	pred(o)	error t
1	1	0	0	0	0
1	3	-1	1	0	1
1	2	2	0	0	0
1	4	4	0	0	0
1	1	-2	1	0	1

$w_i = w_i + \eta \sum_{i=1}^{n} \text{error}\, x_i$

$w_0 = 0 + 0.1[(1\times0)+(1\times1)+(1\times0)+(1\times0)+(1\times1)]$

$= 0.2$

$w_1 = 0 + 0.1[0+3+0+0+1] = 0.4$

$w_2 = 0 + 0.1[-1+0+0+0+(-2)] = -0.3$

Thrushold $0.2 + 0.4x_1 - 0.3x_2 > 0$ — ①

x_1	x_2	eq①	classification
1	0	0.6	1
3	-1	1.7	1
2	2	0.4	1
4	4	0.6	1
1	-2	1.2	1

Iteration 2

x_0	x_1	x_2	t	pred(o)	error
1	1	0	0	1	-1
1	3	-1	1	1	0
1	2	2	0	1	-1
1	4	4	0	1	-1
1	1	-2	1	1	0

$w_0 = 0.2$ $w_1 = 0.4$ $w_2 = -0.3$ $\eta = 0.1$

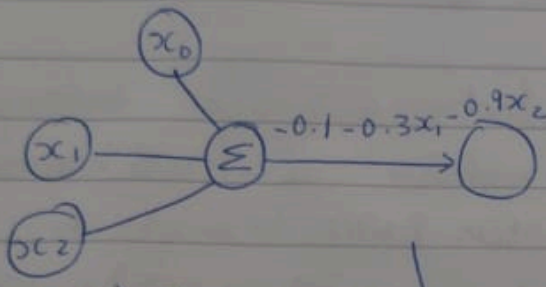
$w_0 = 0.2 + 0.1[(-1)+0+(-1)+(-1)+0] = -0.1$

$w_1 = 0.4 + 0.1[-1+0-2-4+0] = -0.3$

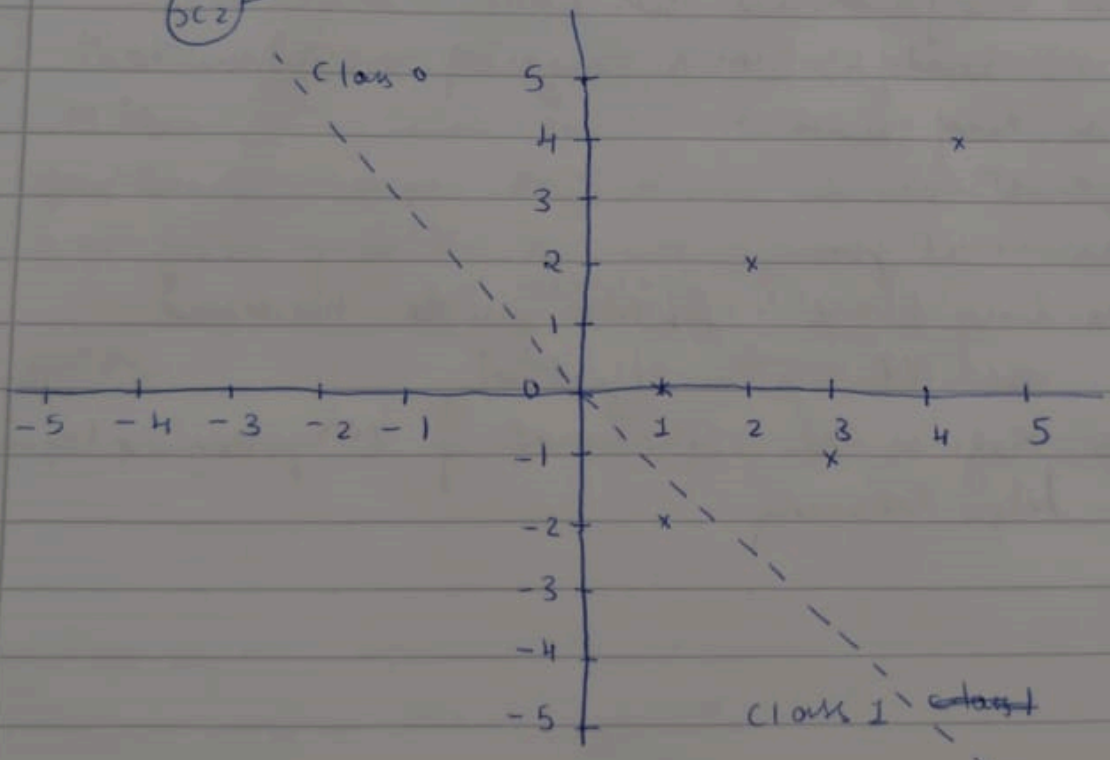
$w_2 = -0.3 + 0.1[0+0-2-4+0] = -0.9$

Threshold, $-0.1 - 0.3x_1 - 0.9x_2 > 0$ — ②

x_1	x_2	eq ②	Classification
1	0	-0.4	0
3	-1	-0.1	0
2	2	-2.5	0
4	4	-4.9	0
1	-2	1.4	1

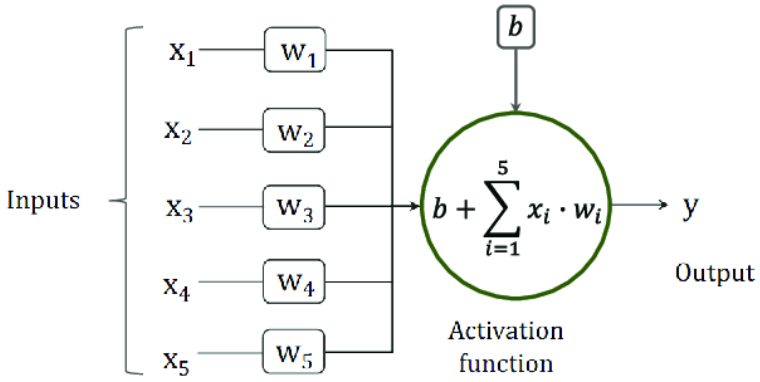


$$O(\vec{x}) = \begin{cases} 1, \text{ if eq ② } > 0 \\ 0, \text{ Otherwise} \end{cases}$$

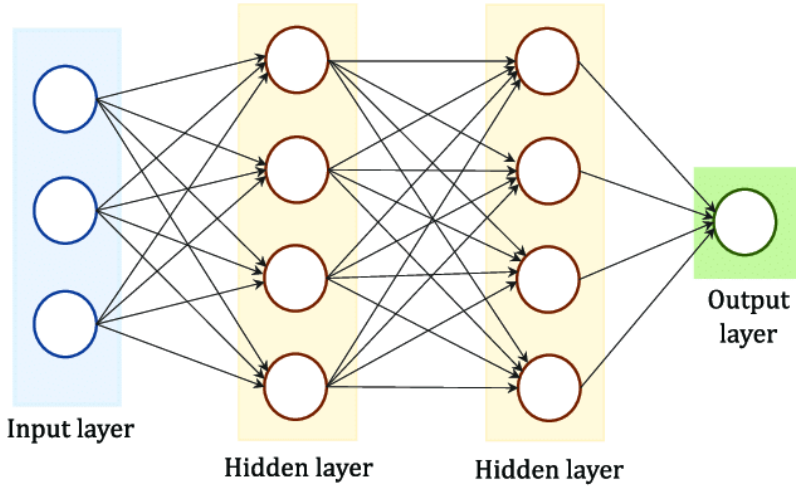


6.3

a) how a neural net looks like,



(a) A perceptron with $N = 5$ inputs x_i and one output y.



1. Neurons (Nodes):
    - Each node represents a unit of computation.
    - Neurons are organized into layers: input layer, hidden layers, and output layer.
2. Layers:
    - Input Layer: Accepts the data (features of the dataset). Each node in this layer represents one feature.
    - Hidden Layers: Perform computations and extract patterns. These layers consist of neurons interconnected with weights.
    - Output Layer: Produces the final result, such as a classification label or regression value.
3. Connections (Edges):
    - Nodes in one layer are connected to nodes in the next layer.
    - Each connection has a weight that determines the influence of one neuron on another.
4. Weights and Biases:
    - Weights: Multipliers applied to the input signals.
    - Bias: Added to the weighted sum to shift the activation function and adjust the decision boundary.

5. Activation Functions:
    - Apply a transformation to the weighted sum of inputs and bias, introducing non-linearities (e.g., sigmoid, ReLU, tanh).
- Input Layer: Nodes aligned in a vertical column, each representing an input feature (e.g., pixel values in an image).
- Hidden Layers: One or more vertical columns of nodes, with each node connected to all nodes in the previous and next layers (fully connected layers).
- Output Layer: A single column with one or more nodes depending on the task (e.g., 1 node for binary classification, multiple nodes for multi-class classification).

b) how a neural net is used for classification

- As explained earlier, Neural Network has both linear as well as Non-linear part. Therefore, the linear part helps in calculating the linear combination of weights and inputs which is then passed through the non-linear part(activation function), in return it helps in the classification process.
- Example of activation functions can be the sigmoid function or step function.

c) how a neural net is learned and what the problem of vanishing gradients is,

- Once the output is predicted by the Neural Network, then the error is calculated.
- Error = Ground truth - predicted output.
- Hence, the main aim of the neural network is to minimize errors. Hence, Neural network converges towards the minimum error hypothesis.
- After the calculation of the error, the gradient descent algorithm is used to update The weight parameters should fit best for all the training examples.

Vanishing Gradient Problem:

- With the gradient descent strategy, it aims to reach the minimum the error function.
- In the process, due to the presence of many hidden layers, the gradient of these layers will be very low for eg. 0.000,1, and when multiplying these small numbers would result in further low values which are near 0.
- Hence, the gradient will be lost w=before it reaches the minimum. This problem is known as the Vanishing Gradient problem and it can be solved by using the RELU activation function.

d) as well as why neurons should not all be initialized for all weights to the same value (e.g. 0).

- If all the weights are initialized with the same value, then the neural network cannot learn, because neurons will learn in the same way and all the hidden layers will get the same error, which in return updates the weights symmetrically.
- This results in an unstable model, as the model is not learning anything new.

Additionally argue, why we need the nonlinearity of such neural networks. Try to restrict to the most important points and be as clear as possible.

- If only linear functions are used in neural networks, then the combination of all these

will result in a linear activation function.
- This will probably lower the ability of the model to be fit for non-linear boundaries.

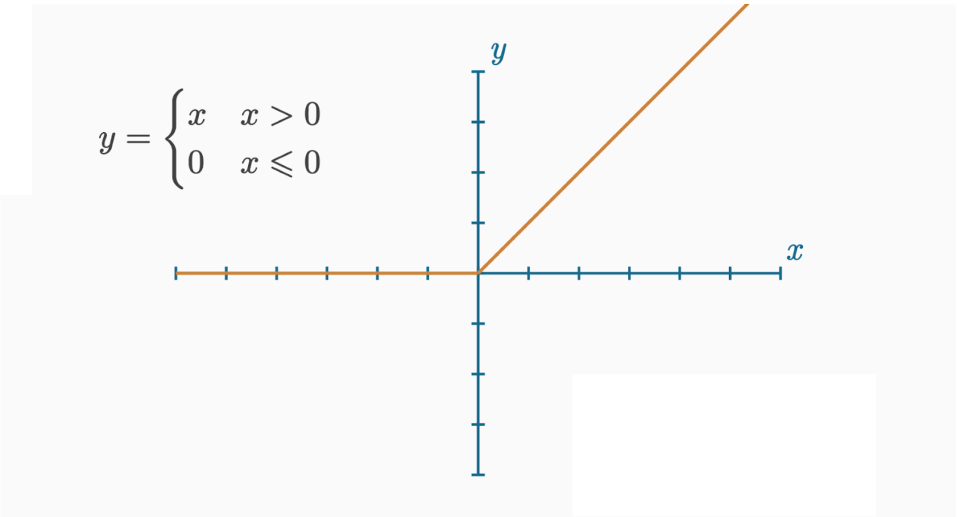
6.4

Assignment 6.4

Discuss **three** other activation functions besides the sign (binary step) function and the sigmoid function. Especially consider the variants of Rectified Linear Units (ReLU) and Leaky ReLUs! Describe the formulas and their advantages and disadvantages!

1. Rectified Linear Unit (ReLU)

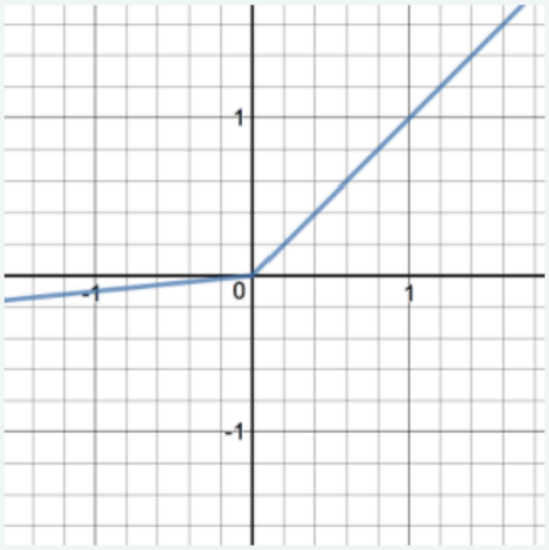
f(x)= max(0,x)



- **Advantages**:
  - **Computational Efficiency**: Simple and fast to compute.
  - **Sparsity**: Activates only a portion of the network at any given time (outputs are 0 for negative inputs).
  - **Mitigates Vanishing Gradient**: ReLU does not saturate for positive inputs, so gradients remain large.
- **Disadvantages**:
  - **Dying Neurons**: Neurons can "die" if the weights update such that their inputs are always negative, leading to zero gradients and no updates during training.
  - **Not differentiable at zero**: Although this is rarely an issue in practice.

2. Leaky ReLU

$$R(z) = \begin{cases} z & z > 0 \\ \alpha z & z <= 0 \end{cases}$$

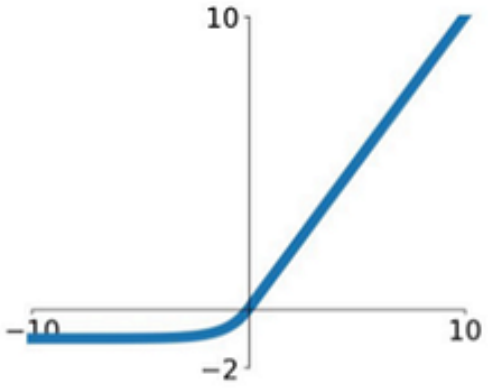


- **Advantages**:
  - **Solves Dying ReLU Problem**: By allowing a small slope (α\alphaα) for negative inputs, it ensures no neuron is completely inactive.
  - **Better Gradient Flow**: Ensures gradients flow even for negative inputs.
- **Disadvantages**:
  - **Hyperparameter Tuning**: The choice of α\alphaα can affect performance and may require tuning.
  - **Non-zero Output for Negative Inputs**: This can sometimes lead to less sparsity compared to standard ReLU.

3. Exponential Linear Unit (ELU)

$$\text{ELU}$$

$$\begin{cases} x & x \geq 0 \\ \alpha(e^x - 1) & x < 0 \end{cases}$$



- **Advantages**:
  - **Improves Gradient Flow**: Provides smoother gradients for negative inputs compared to ReLU and Leaky ReLU.
  - **Non-Zero Mean Activations**: Helps reduce bias shifts in the network, leading to faster convergence during training.
  - **No Dying Neurons**: Similar to Leaky ReLU, negative inputs still contribute to gradient updates.
- **Disadvantages**:
  - **Computational Cost**: More expensive to compute due to the exponential function.
  - **Hyperparameter Tuning**: Requires choosing the parameter α\alphaα.

Comparison Table

Activation Function	Formula	Key Advantages	Key Disadvantages
ReLU	max(0,x)\max(0, x)max(0,x)	Simple, computationally efficient; mitigates vanishing gradients	Dying neurons for x≤0
Leaky ReLU	max(αx,x)	Solves dying ReLU; better gradient flow for negative inputs	Requires tuning of α
ELU	x if x>0, α(exp(x)−1) otherwise	Smooth gradients, faster convergence	Computationally expensive

Each of these activation functions is chosen depending on the task, architecture, and the specific challenges faced during training. Variants like Leaky ReLU and ELU address key limitations of ReLU while introducing additional computational complexity or hyperparameters.