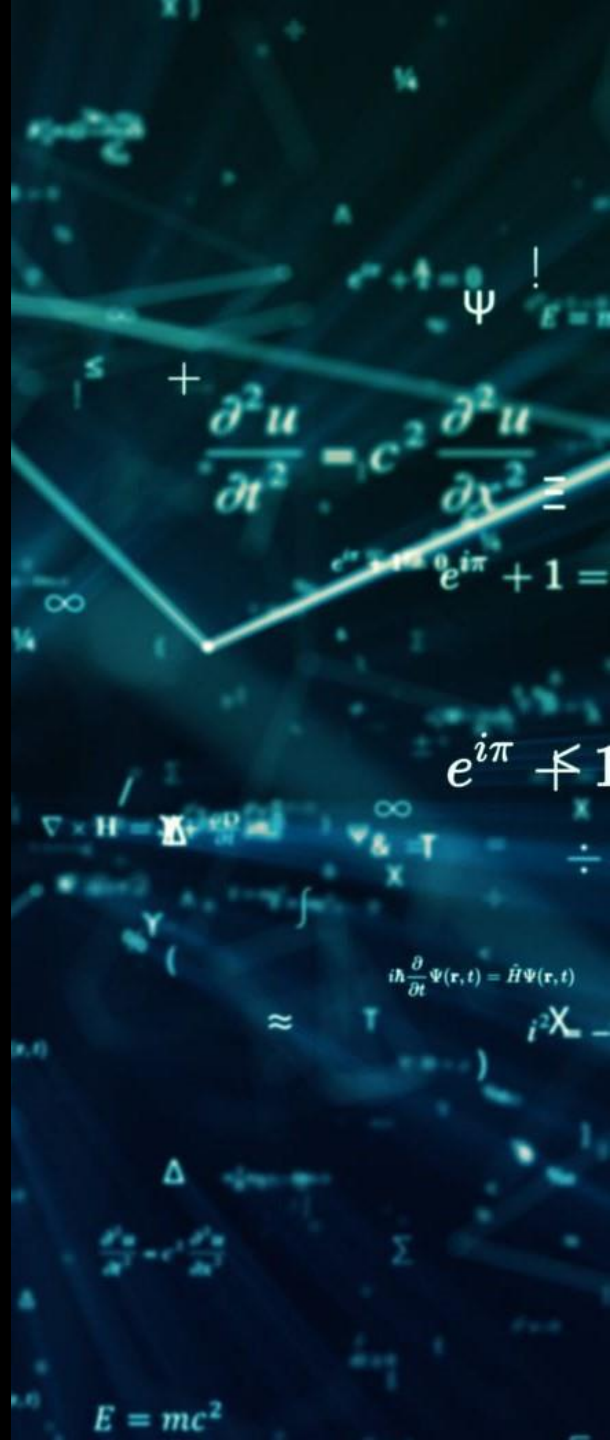
# Artificial Intelligence Algorithms and Mathematics

CSCN 8000

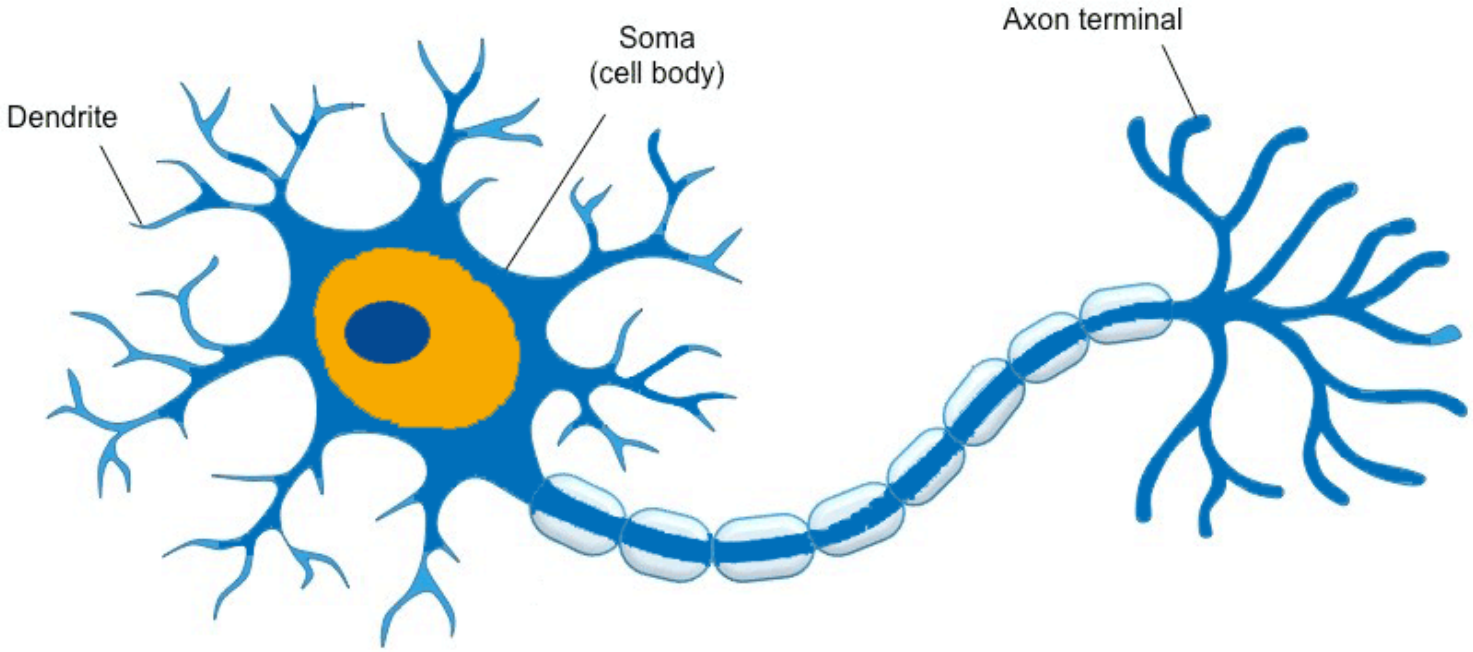


# Classification

- Perceptron
- Multi-Layered Perceptrons
- Activation Functions
- Backpropagation
- Bias vs Variance
- Solutions for Overfitting

# Biological Neuron

- A human brain has billions of neurons. Neurons are interconnected nerve cells in the human brain that are involved in processing and transmitting chemical and electrical signals.
- Dendrites receive input signals, soma cells processes them, and axon terminal produce the output signals.



# Artificial Neurons

- An artificial neuron is a mathematical function based on a model of biological neurons, where each neuron takes **inputs**, **weighs** them separately, **sums them up** and passes this sum through a **nonlinear function** to produce output.

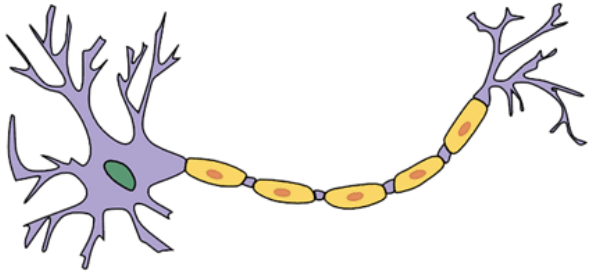


Fig: Biological Neuron

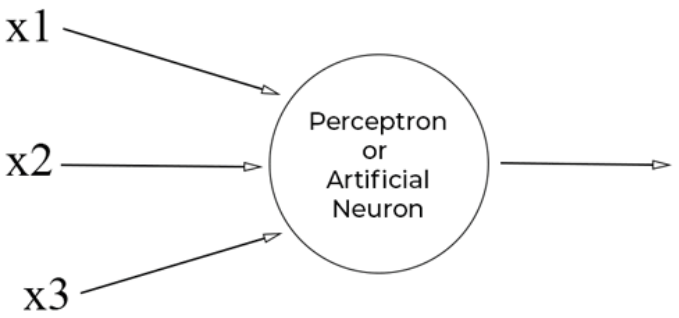


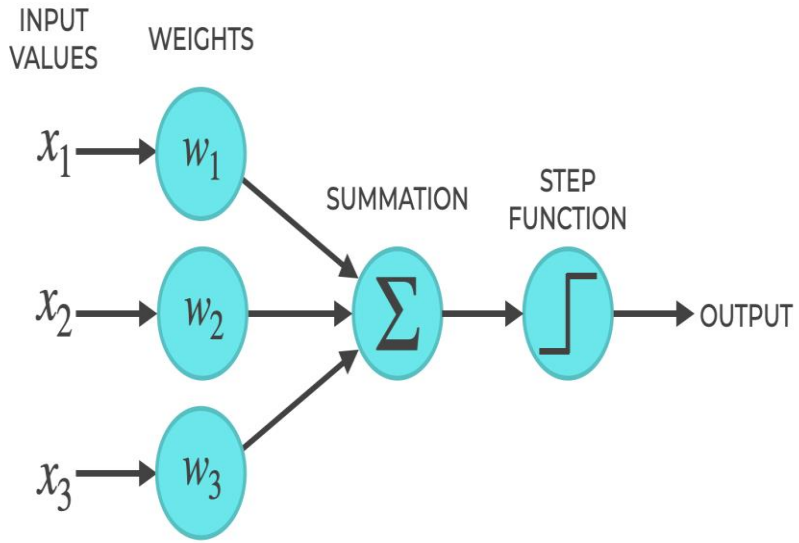

Fig: Artificial Neuron

# Perceptron

- The perceptron is an artificial neuron that carries out binary classification with supervised learning.
- The components of a perceptron are Input signals $\vec{\boldsymbol{x}}$, Weights $\vec{\boldsymbol{w}}$, Bias term $\boldsymbol{b}$, Step function $\delta$ , and Output values. Mathematically:

$$Output = \delta(\vec{\boldsymbol{w}}\,\vec{\boldsymbol{x}} + \boldsymbol{b})$$

- Does this look similar to a formulation we've seen before?



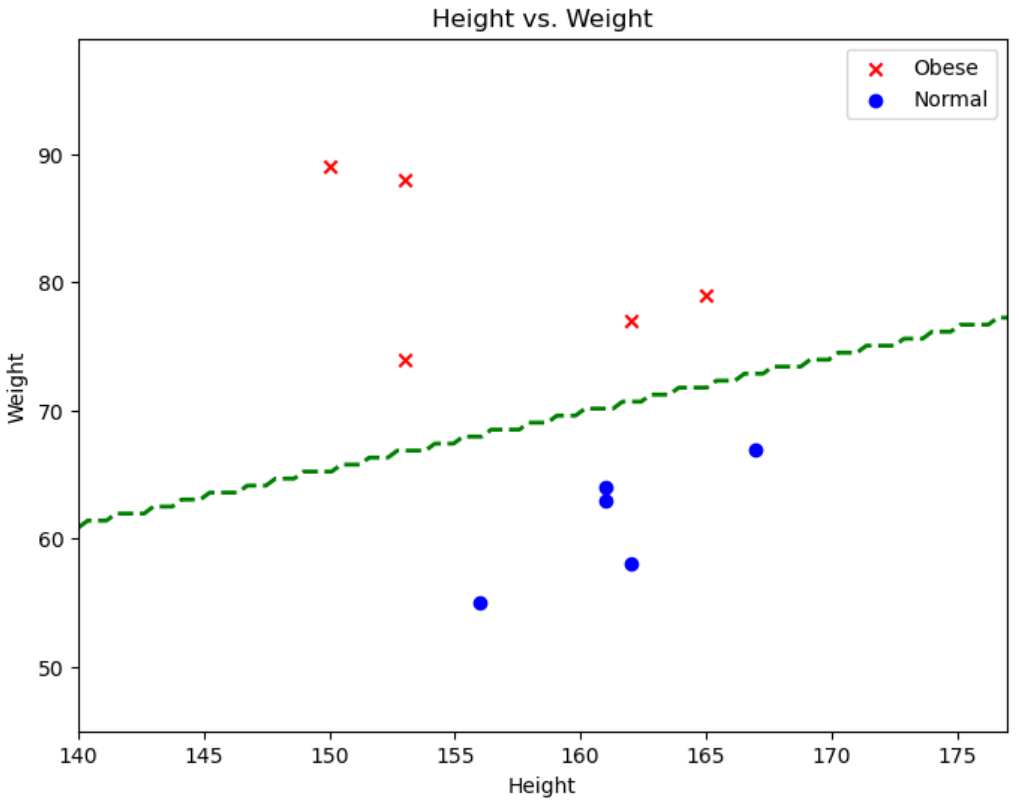
# Recall: Decision Boundary Definition

- The decision boundary in a binary classification problem is a hypersurface that separates the feature space into regions corresponding to different classes.
- The decision boundary only exists as a line or hyperplane if the classes are ***linearly separable***
- The equation of the linear decision boundary is as follows:

$$\hat{y} = \vec{w}\,\vec{x} + b$$

  - $\vec{x}$ represents all input features.
  - $\hat{y}$ represents estimated class a point belongs to.
  - $y$ represents the label of the point $\{-1,1\}$

- $\vec{w}, b$ are learnable parameters to be estimated



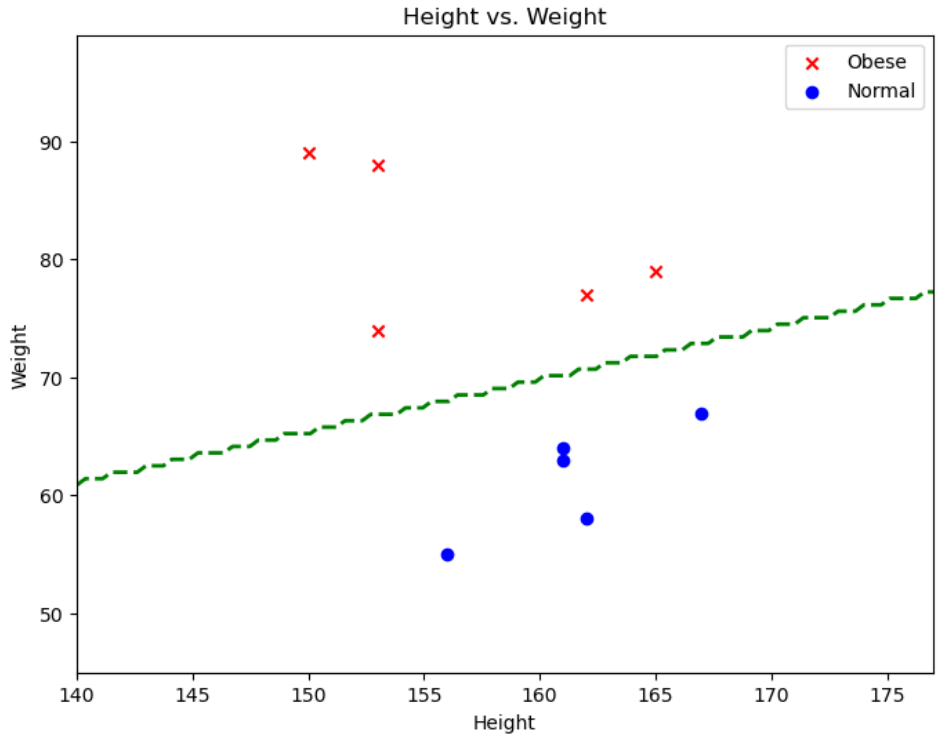

$$\hat{y} = \vec{w}\,\vec{x} + b$$

# Perceptron

- The perceptron learns a ***linear*** decision boundary to separate between classes.

$$\hat{\boldsymbol{y}} = \vec{\boldsymbol{w}}\,\vec{\boldsymbol{x}} + \boldsymbol{b}$$

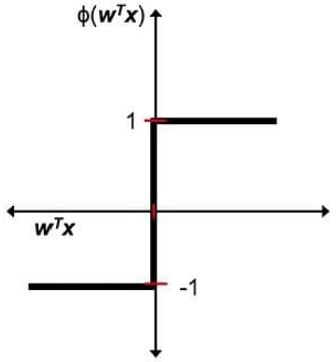
- The perceptron assumes that each point in the classes have labels $y \in \{1, -1\}$
- As we know, $\hat{\boldsymbol{y}}$ is a continuous value where $\hat{\boldsymbol{y}} > 0$ if above boundary and $\hat{\boldsymbol{y}} < 0$ if below boundary.
- We need a way to transform $\hat{\boldsymbol{y}}$ to binary values $\in \{1, -1\}$.

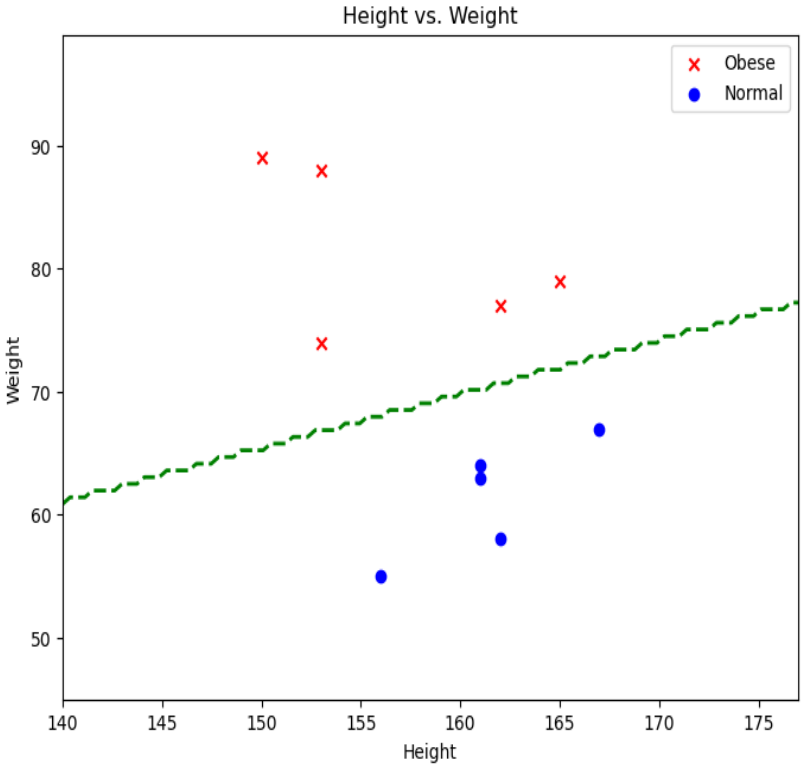


# Perceptron

- To achieve this, we need a way to transform $\hat{y}$ to binary values $\in \{1, -1\}$.
- We utilize a simple *sign function* formulated as follows:

$$\phi(\hat{y}) = \begin{cases} 1, & if\ \hat{y} \geq 0 \\ -1, & if\ \hat{y} < 0 \end{cases}$$




- For all correctly classified points $\rightarrow y_i \phi(\hat{y}_i) = 1 \rightarrow y_i \hat{y}_i \geq 0$
- For all incorrectly classified points $\rightarrow y_i \phi(\hat{y}_i) = -1 \rightarrow y_i \hat{y}_i < 0$

# Perceptron

- For all incorrectly classified points $\rightarrow \boldsymbol{y_i \phi(\hat{y}_i) = -1} \rightarrow \boldsymbol{y_i \hat{y}_i < 0}$
- *Target:* Classify ***all*** points ***correctly***.
- To achieve this, we will formulate an error function *only* for incorrectly classified points $\boldsymbol{\vec{x}_i}$ , such as:

$$L(\vec{\boldsymbol{w}}, \boldsymbol{b}) = -\boldsymbol{y_i \hat{y}_i} = -\boldsymbol{y_i}(\vec{\boldsymbol{w}}\,\vec{\boldsymbol{x}_i} + \boldsymbol{b})$$

- This error function quantifies how far is the misclassified point from the decision boundary.
  - The negative sign ensures that the Loss function is always positive.
- This loss function can be minimized using Gradient Descent to reach an optimal decision boundary.

# Perceptron

$$L(\vec{\boldsymbol{w}}, \boldsymbol{b}) = -\, y_i \hat{\boldsymbol{y}}_{\boldsymbol{i}} = -y_i(\vec{\boldsymbol{w}}\, \vec{\boldsymbol{x_i}} + \boldsymbol{b}\,)$$

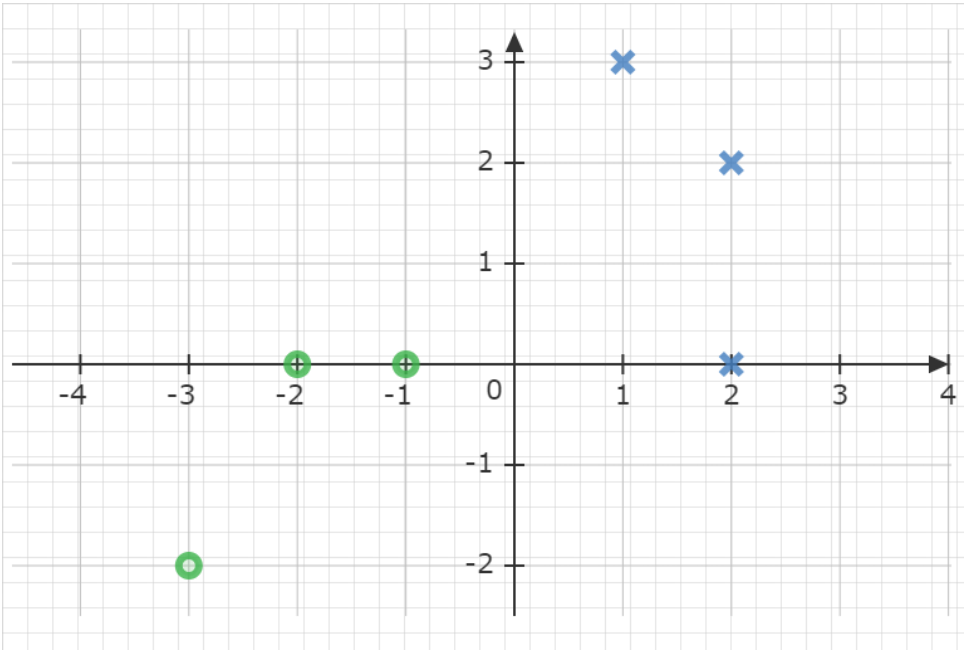
- To get the values of $\vec{\boldsymbol{w}}, \boldsymbol{b}$ at which $L(\vec{\boldsymbol{w}}, \boldsymbol{b})$ is minimum using Gradient Descent, we need to compute $\frac{d(L)}{d\vec{\boldsymbol{w}}}$ and $\frac{d(L)}{d\boldsymbol{b}}$.
- $\frac{d(L)}{d\vec{\boldsymbol{w}}} = -y_i \boldsymbol{x_i}\,, \quad \frac{d(L)}{d\boldsymbol{b}} = -y_i$
- Gradient Descent formulation for step size $\boldsymbol{\delta}$ and one misclassified point $\boldsymbol{x_i}$ :
  - $\vec{\boldsymbol{w}}_{\boldsymbol{t+1}} = \vec{\boldsymbol{w}}_{\boldsymbol{t}} - \boldsymbol{\delta}\frac{d\boldsymbol{L}}{d\vec{\boldsymbol{w}}} = \vec{\boldsymbol{w}}_{\boldsymbol{t}} + \boldsymbol{\delta}\, y_i \boldsymbol{x_i}$
  - $\boldsymbol{b_{t+1}} = \boldsymbol{b_t} - \boldsymbol{\delta}\frac{d\boldsymbol{L}}{d\boldsymbol{b}} = \boldsymbol{b_t} + \boldsymbol{\delta}\, y_i$

# Perceptron

- The steps of the perceptron learning algorithm are as follows:
  - Initialize $\overrightarrow{\boldsymbol{w_0}}, \boldsymbol{b_0}$ to random initial values
  - Use the rule $\boldsymbol{y_i \hat{y}_i < 0}$ to identify points that are misclassified given the current weight and bias.
  - Choose one misclassified point $\boldsymbol{x_i}$.
  - Update the weight and bias using the following update rules:
    - $\boldsymbol{\vec{w}_{t+1} = \vec{w}_t - \delta \frac{dL}{d\vec{w}} = \vec{w}_t + \delta\, y_i \vec{x}_i}$
    - $\boldsymbol{b_{t+1} = b_t - \delta \frac{dL}{db} = b_t + \delta\, y_i}$
  - Repeat Steps 2-4 until all points are correctly classified.
- The previous algorithm is guaranteed to converge to a solution if all the points are ***strictly linearly separable***.

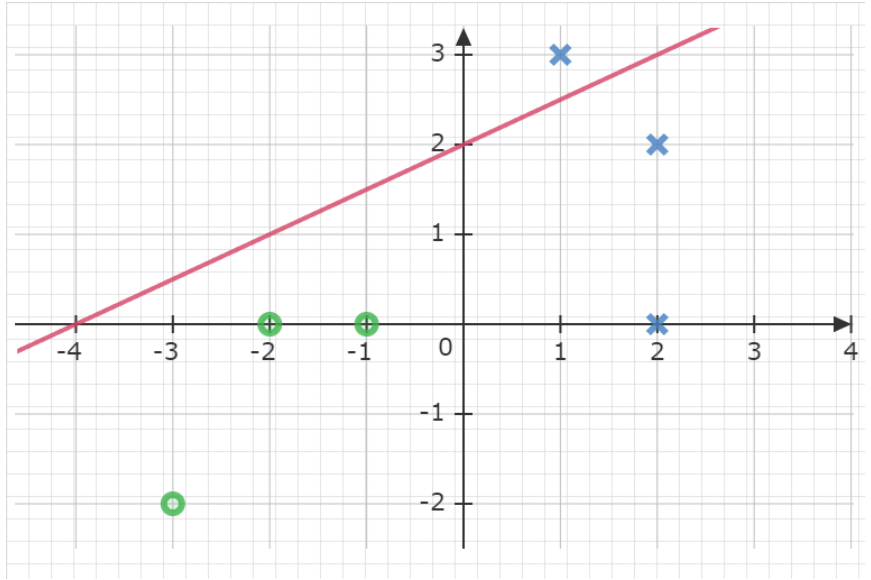
# Perceptron: Example

- Given the following dataset, we want to use the Perceptron algorithm to find the decision boundary.
- Blue points have $y_i = 1$.
- Green points have $y_i = -1$.



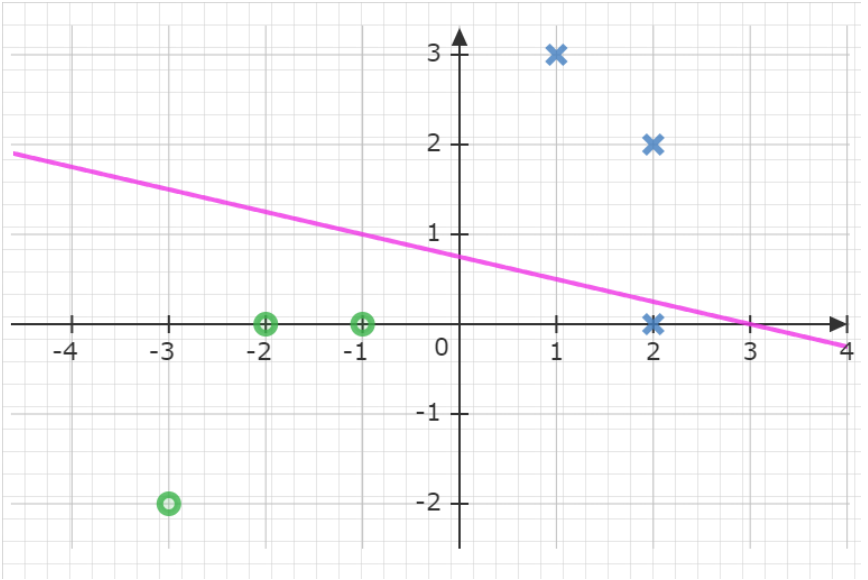
# Perceptron: Example

- Start by initializing the weight and bias in step $0$:
  - $\overrightarrow{\boldsymbol{w_0}} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}, \boldsymbol{b_0} = \boldsymbol{-4}$
- Check which points are misclassified:
  - $\begin{bmatrix} 1 \\ 3 \end{bmatrix} \rightarrow +1((-1 * 1) + (3 * 2) - 4) = 1 > 0$
  - $\begin{bmatrix} 2 \\ 2 \end{bmatrix} \rightarrow +1((-1 * 2) + (2 * 2) - 4) = -2 < 0$
- Update weights using point $\begin{bmatrix} 2 \\ 2 \end{bmatrix}$ using $\boldsymbol{\delta = 1}$:
  - $\overrightarrow{\boldsymbol{w_1}} = \begin{bmatrix} -1 \\ 2 \end{bmatrix} + \boldsymbol{1} \begin{bmatrix} 2 \\ 2 \end{bmatrix} = \begin{bmatrix} 1 \\ 4 \end{bmatrix}$
  - $\boldsymbol{b_1} = -4 + 1 = -3$



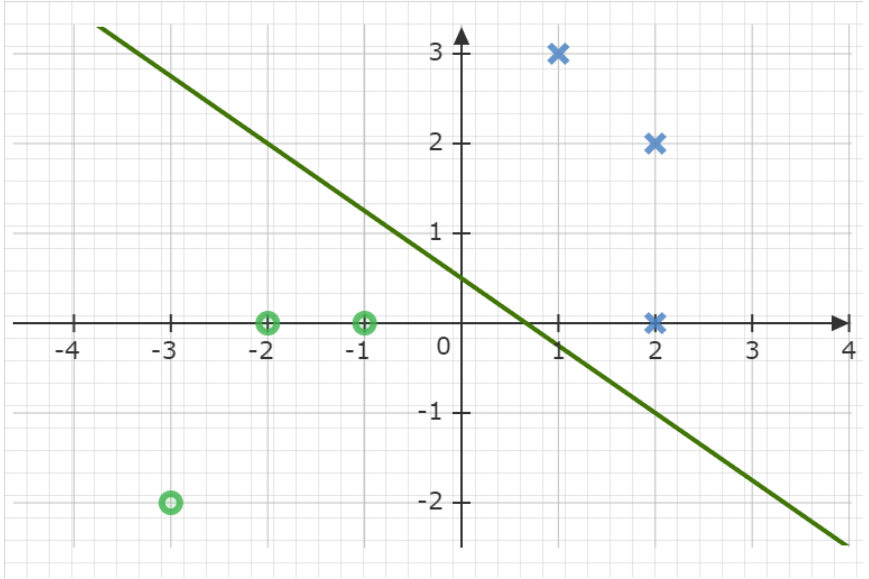
# Perceptron: Example

- $\overrightarrow{\boldsymbol{w_1}} = \begin{bmatrix}1\\4\end{bmatrix}, \boldsymbol{b_1} = -3$
- Check which points are misclassified:
  - $\begin{bmatrix}2\\0\end{bmatrix} \rightarrow +1((2*1)+(0*4)-3) = -1 < 0$
- Update weights using point $\begin{bmatrix}2\\0\end{bmatrix}$ using $\boldsymbol{\delta = 1}$:
  - $\overrightarrow{\boldsymbol{w_2}} = \begin{bmatrix}1\\4\end{bmatrix} + \mathbf{1}\begin{bmatrix}2\\0\end{bmatrix} = \begin{bmatrix}3\\4\end{bmatrix}$
  - $\boldsymbol{b_2} = -3 + 1 = -2$



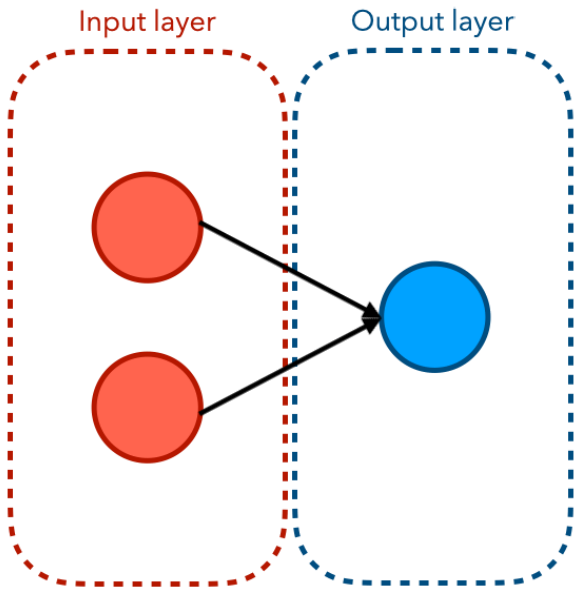
# Perceptron: Example

- $\overrightarrow{w_2} = \begin{bmatrix} 3 \\ 4 \end{bmatrix}, b_2 = -2$
- Check which points are misclassified:
  - All points are correctly classified with $y_i \hat{y}_i \geq 0$

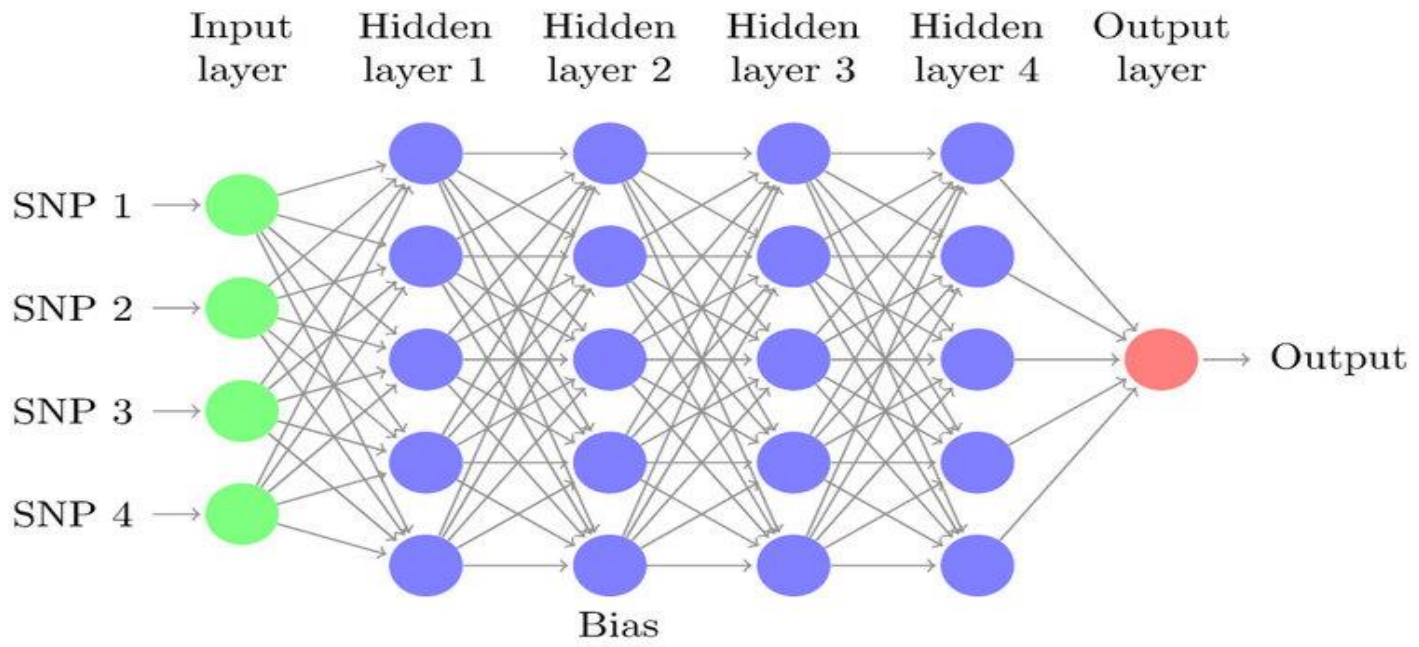


# Multi-Layer Perceptron

- The perceptron is only capable of generating a linear decision boundary and cannot be used to solve complex classification problems.
- To have a more powerful classification algorithm, we can stack a series of perceptrons on each other creating a **multi-layer perceptron.**
- Each hidden layer consists of a series of perceptrons (neurons) receiving input from previous layer and delivering output to the next layer.



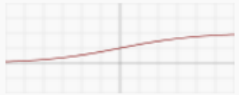
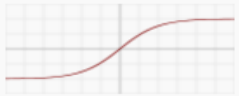
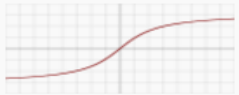
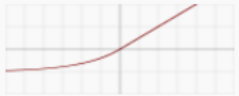
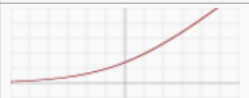
# Multi-layer Perceptron



# Multi-layer Perceptron

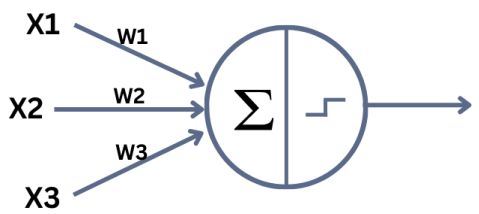
- Each neuron (perceptron) $i$ in the hidden layer $l$ with $N^l$ neurons must do the following operations:
  - Multiply the outputs from the previous layer by their corresponding weights → $\boldsymbol{w_i^l * o_i^{l-1}}$
  - Sum the multiplications between the weights and the outputs and add the bias term → $\boldsymbol{\left[\sum_{i=1}^{N^l} w_i^l * o_i^{l-1}\right] + b_i^l}$
  - Pass the summation to an *activation function f* to produce the output of the current neuron → $\boldsymbol{o_i^l = f\left(\left[\sum_{i=1}^{N^l} w_i^l * o_i^{l-1}\right] + b_i^l\right)}$
- The activation function is the most important component of the MLP. Without an activation function, the MLP will still only be able to learn a linear decision boundary similar to a single perceptron.
- There are different types of activation functions that could be used in an MLP.

# Types of Activation Functions

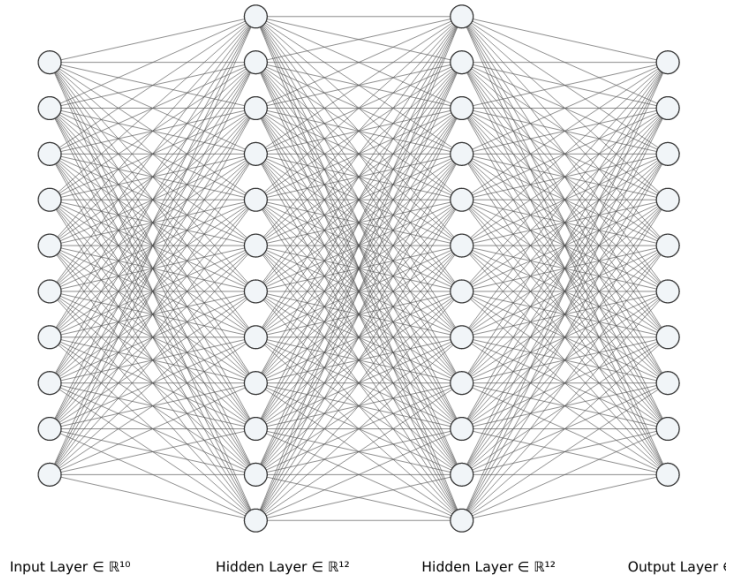
| Name | Plot | Equation | Derivative |
|---|---|---|---|
| Identity |  | $f(x) = x$ | $f'(x) = 1$ |
| Binary step |  | $f(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ 1 & \text{for} \quad x \geq 0 \end{cases}$ | $f'(x) = \begin{cases} 0 & \text{for} \quad x \neq 0 \\ ? & \text{for} \quad x = 0 \end{cases}$ |
| Logistic (a.k.a Soft step) |  | $f(x) = \frac{1}{1 + e^{-x}}$ | $f'(x) = f(x)(1 - f(x))$ |
| TanH |  | $f(x) = \tanh(x) = \frac{2}{1 + e^{-2x}} - 1$ | $f'(x) = 1 - f(x)^2$ |
| ArcTan |  | $f(x) = \tan^{-1}(x)$ | $f'(x) = \frac{1}{x^2 + 1}$ |
| Rectified Linear Unit (ReLU) |  | $f(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases}$ | $f'(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ 1 & \text{for} \quad x \geq 0 \end{cases}$ |
| Parameteric Rectified Linear Unit (PReLU) [2] |  | $f(x) = \begin{cases} \alpha x & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases}$ | $f'(x) = \begin{cases} \alpha & \text{for} \quad x < 0 \\ 1 & \text{for} \quad x \geq 0 \end{cases}$ |
| Exponential Linear Unit (ELU) [3] |  | $f(x) = \begin{cases} \alpha(e^x - 1) & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases}$ | $f'(x) = \begin{cases} f(x) + \alpha & \text{for} \quad x < 0 \\ 1 & \text{for} \quad x \geq 0 \end{cases}$ |
| SoftPlus |  | $f(x) = \log_e(1 + e^x)$ | $f'(x) = \frac{1}{1 + e^{-x}}$ |

# Multi-layer Perceptron: Learning Approach

- In a single-layer perceptron, we were able to learn the optimal $\vec{\boldsymbol{w}}, \boldsymbol{b}$ parameters by formulating $L(\vec{\boldsymbol{w}}, \boldsymbol{b})$, computing $\frac{d(L)}{d\vec{\boldsymbol{w}}}$ and $\frac{d(L)}{d\boldsymbol{b}}$ and utilizing Gradient Descent.
- In an MLP, could the same exact approach be used to reach the optimal parameters?
  - It's more complicated since we have different parameters (weight and bias) for each neuron and each neuron passes its output to the following layer.
  - $L(\vec{\boldsymbol{w}}, \boldsymbol{b})$ is calculated after the output layer, we need a way to calculate $\frac{d(L)}{d\vec{\boldsymbol{w}}}$ and $\frac{d(L)}{d\boldsymbol{b}}$ for each neuron in the previous layers.



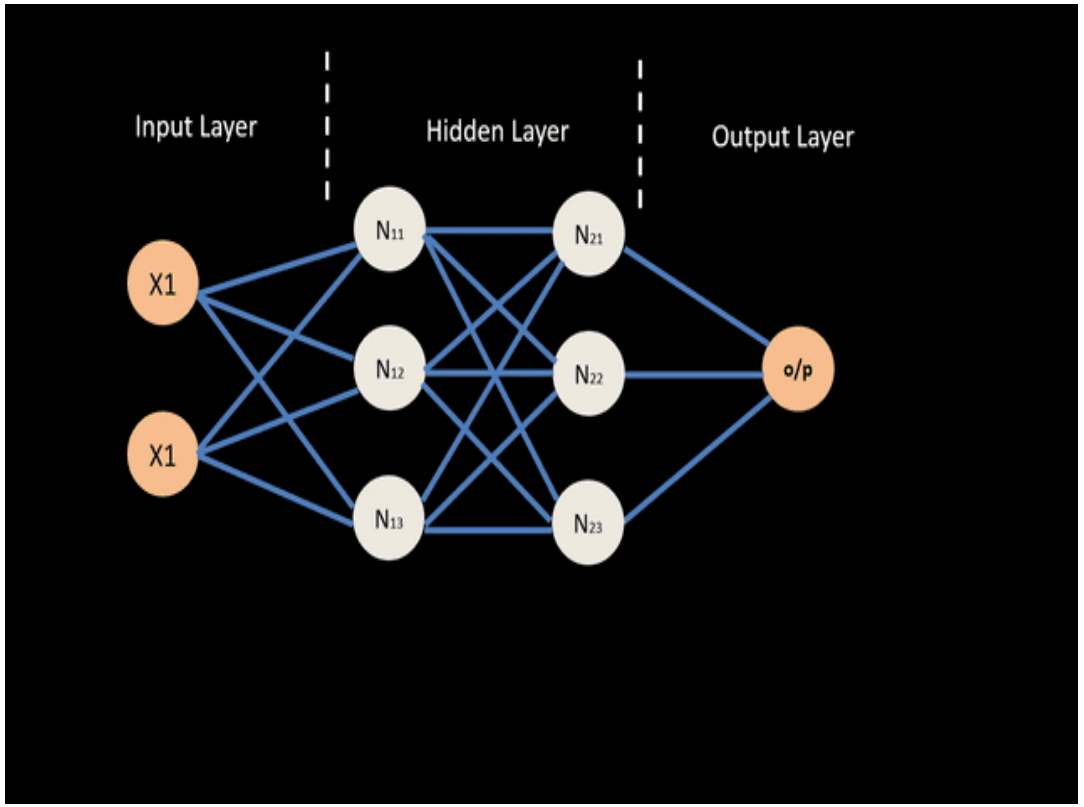

Single-layer perceptron




Multi-layer perceptron

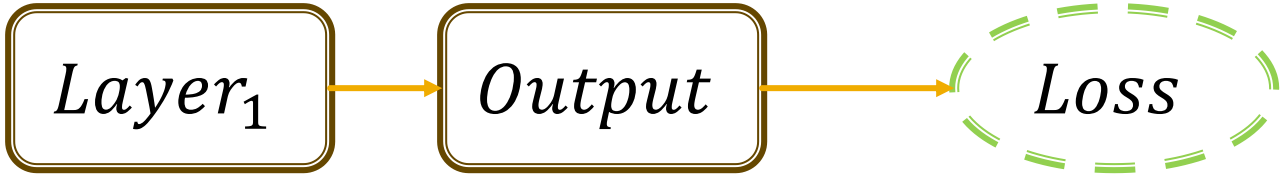
# Backpropagation

- Backpropagation is a gradient-based optimization algorithm that adjusts the weights of the MLP to minimize the difference between the predicted and actual outputs.
  - Forward Pass:
    - Input data is passed through the network layer by layer
    - The final output is compared to the target output.
    - The error (the difference between the predicted and actual outputs) is calculated.
  - Backward Pass:
    - The gradient of the error with respect to the weights is computed layer by layer starting from the output layer and moving backward towards the input layer.
    - The computed gradients are then used to update the weights in the direction that minimizes the error.
    - *But how are we able to propagate the gradient of the error to all the previous layers? → Chain Rule*



# Backpropagation: Chain Rule

- Assume we have an MLP with 1 hidden layer and 1 output layer:




- To calculate derivative of Loss with respect to the first layer:

$$\frac{d(Loss)}{d(Layer_1)} = \frac{d(Loss)}{d(Output)} * \frac{d(Output)}{d(Layer_1)} \rightarrow \boldsymbol{Chain\ Rule}$$

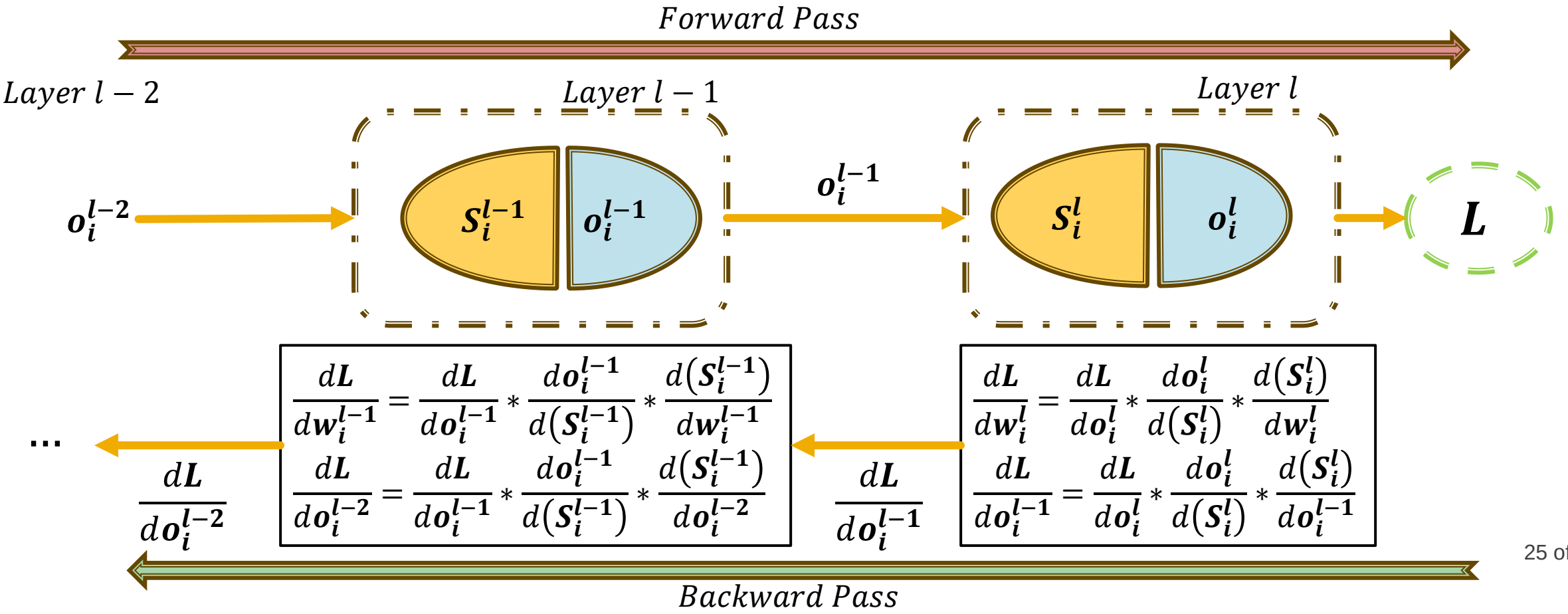
- The chain rule is the core rule used in backpropagation. We use it to recursively calculate the derivative of the loss starting from the output layer and going backwards until the input layer.

# Recall: MLP Steps

- Each neuron (perceptron) $i$ in the hidden layer $l$ with $N^l$ neurons must do the following operations:
  - Multiply the outputs from the previous layer by their corresponding weights → $\boldsymbol{w_i^l * o_i^{l-1}}$
  - Sum the multiplications between the weights and the outputs and add the bias term → $\left[\sum_{i=1}^{N^l} \boldsymbol{w_i^l * o_i^{l-1}}\right] + \boldsymbol{b_i^l}$
  - Pass the summation to an *activation function f* to produce the output of the current neuron → $\boldsymbol{o_i^l = f\left(\left[\sum_{i=1}^{N^l} w_i^l * o_i^{l-1}\right] + b_i^l\right)}$

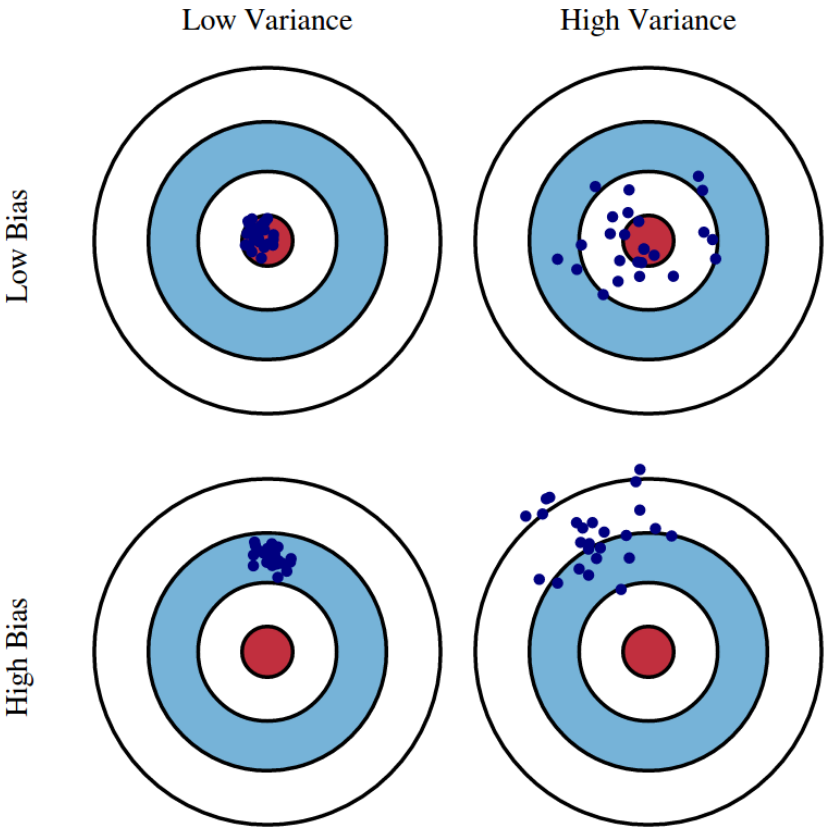
# Backpropagation

- $S_i^l = \left[\sum_{i=1}^{N^l} w_i^l * o_i^{l-1}\right] + b_i^l \quad \rightarrow \quad o_i^l = f(S_i^l)$

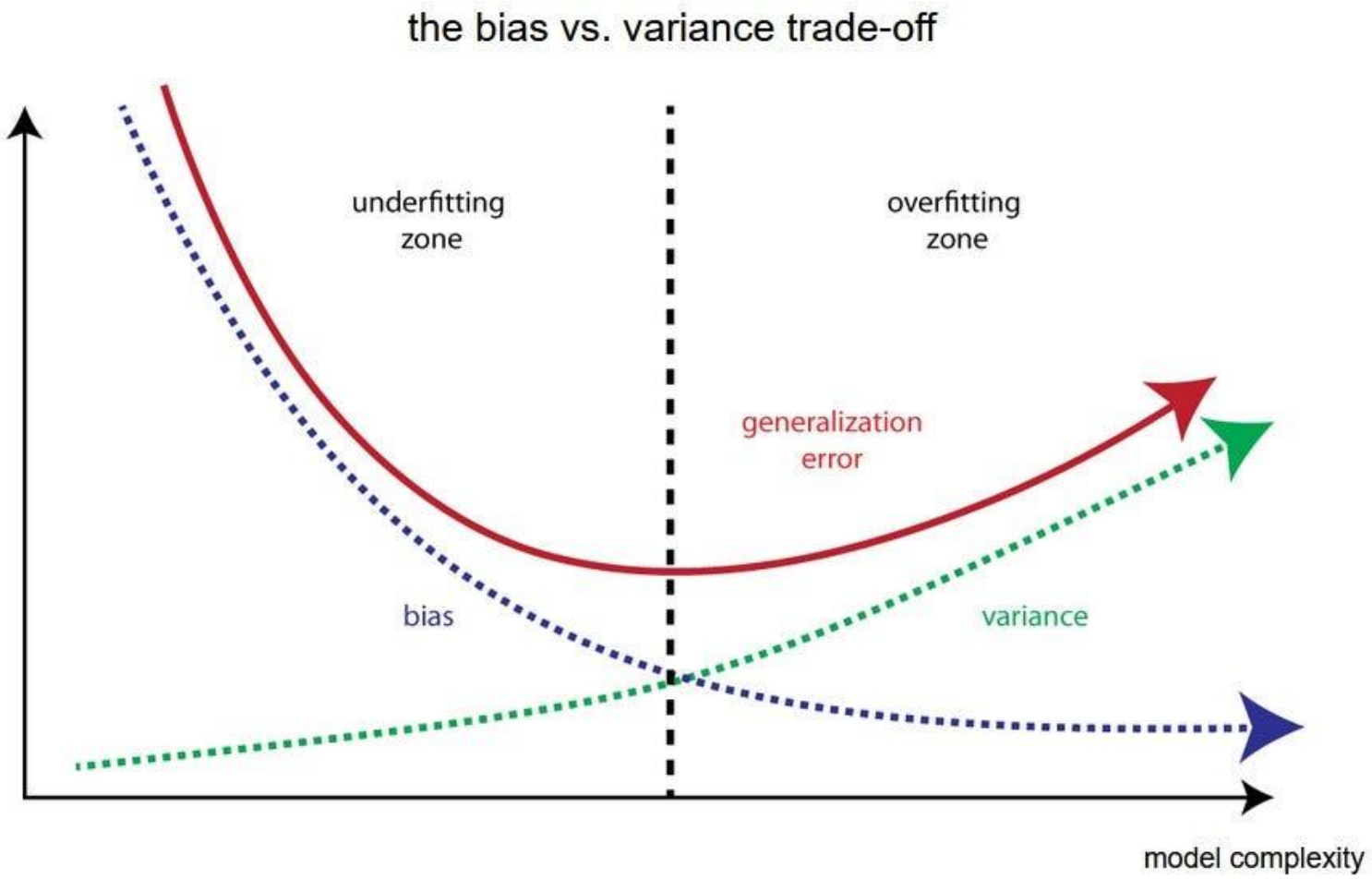


# Bias vs Variance Tradeoff

- The bias-variance tradeoff is a fundamental concept in machine learning that helps to understand the tradeoff between the *simplicity* and *flexibility* of a model.
- **Bias:** Bias is the difference between the average prediction of our model and the correct value which we are trying to predict.
  - High Bias: Occurs when the model is too simple for the problem leading to incorrect predictions → Underfitting
  - Low Bias: Occurs when the model is sufficiently complex to generate correct predictions
- **Variance**: Refers to the model's sensitivity to the specific training data on which it was trained.
  - High Variance: Occurs when the model is too complex where it captures noise in the training data → Overfitting
  - Low Variance: Occurs when the model correctly captures the underlying patterns in the training set and generalizes well to new test data.
- An optimal model will have low bias and low variance.



# Bias vs Variance Tradeoff

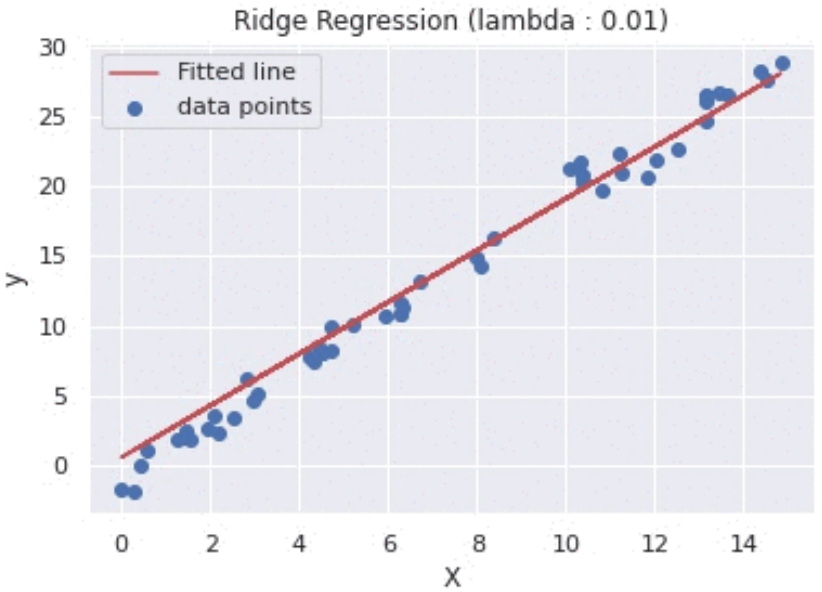


# Ways to reduce overfitting

- Overfitting occurs when a model learns not only the underlying patterns in the training data but also the noise and random fluctuations
- To reduce the overfitting, a general rule of thumb is to try to decrease the model complexity, but not too much to avoid underfitting.
- Specifically, techniques that can be used include:
  - **Cross-Validation:** Assess the model's performance on multiple subsets of the data to ensure good generalization to different portions of the dataset.
  - **Increase Dataset size:** A larger dataset can provide more diverse examples and help the model generalize better.
  - **Early Stopping:** Stop training once the performance on the validation set starts to degrade, preventing the model from overfitting the training data.
  - **Dropout:** random neurons are "dropped out" (ignored) during each training iteration. This helps prevent co-adaptation of neurons and reduces overfitting.
  - **Data Augmentation:** Increase the size of the training dataset by applying various transformations to the existing data
  - ***Regularization?***

# Regularization

- The intuition is that one way to decrease the overall model complexity is to reduce the magnitude of the learned weights $\vec{\boldsymbol{w}}$. This way we won't be assigning very high weights to certain features over the others → Simpler model
- This could be achieved by adding a *regularization term* to the loss function of any machine learning algorithm, such that,
  $L_{reg}(\vec{\boldsymbol{w}}, \boldsymbol{b}) = L(\vec{\boldsymbol{w}}, \boldsymbol{b}) + [\boldsymbol{\beta} * Regularization\ Term]$



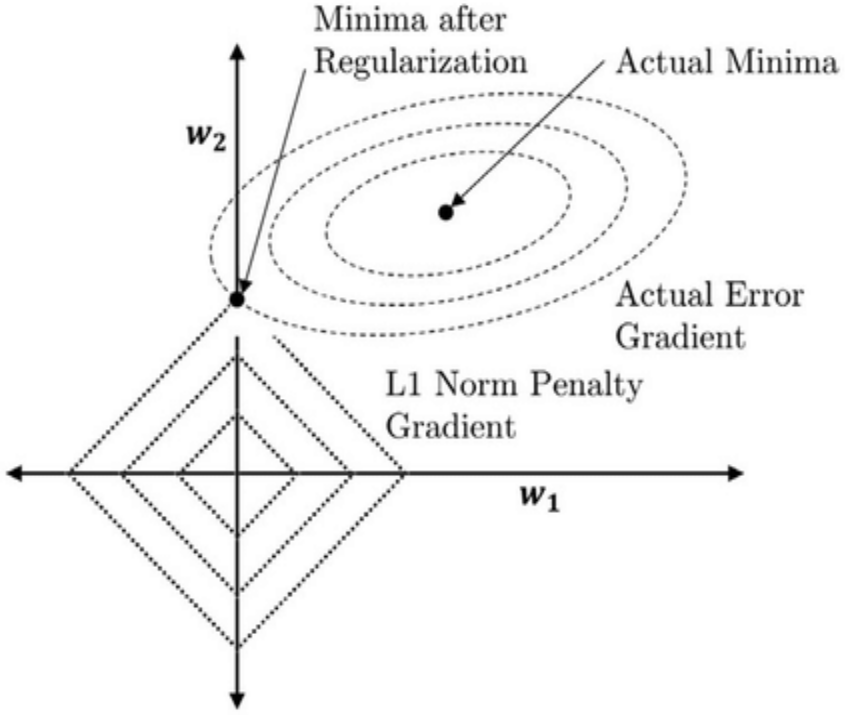

- This will allow the model to not only prioritize minimizing the cost term, but also minimizing the magnitude of the weights (controlled by the hyperparameter $\boldsymbol{\beta}$)
- There are different types of regularization terms that could be used.

# L1 Regularization (Lasso)

- L1 regularization introduces a penalty term proportional to the absolute values of the model parameters.
- Mathematically,

$$L_{reg}(\vec{\boldsymbol{w}}, \boldsymbol{b}) = L(\vec{\boldsymbol{w}}, \boldsymbol{b}) + \left[\boldsymbol{\beta} * \sum_{p} |\overrightarrow{\boldsymbol{w_p}}|\right]$$

- This encourages the model to prefer a sparse set of features, effectively driving some of them to exactly zero.
- Not only does it help prevent overfitting but also performs feature selection by excluding less relevant features.

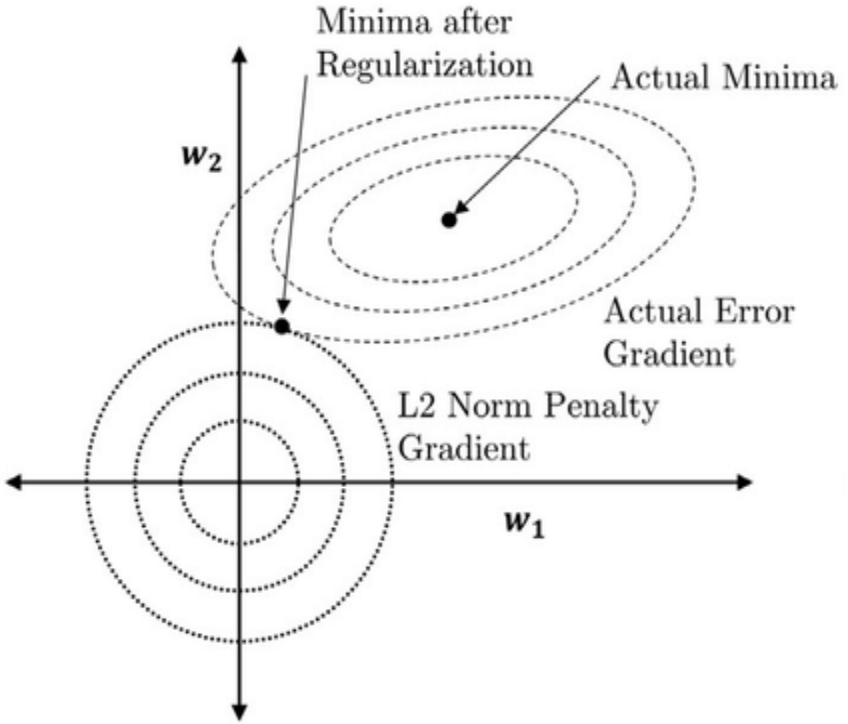


# L2 Regularization (Ridge)

- L2 regularization introduces a penalty term proportional to the square of the model parameters.
- Mathematically,

$$L_{reg}(\vec{w}, b) = L(\vec{w}, b) + \left[\beta * \sum_{p} (\vec{w_p})^2\right]$$

- This discourages the weights from becoming too large, preventing individual features from dominating the model.
- It distributes the importance of features more evenly, leading to a smoother model.



# Extra Resources

- More information regarding the Bias and Variance:
  - https://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote12.html

# Thank you!

- Any questions?

# Disclaimer

Due to nature of the course, various materials have compiled from different open source resources with some moderation. I sincerely acknowledge their hard work and contribution

CONESTOGA

**Thank You**

**Youssef Abdelkareem**

**yabdelkareem@conestogac.on.ca**

**** Credits for Dr. Anasuya Bhima for assisting in the slides structure***