Ahmed Ibrahim

ECE 9039/9309 MACHINE LEARNING

Last Lecture

- Cross-validation
- Outliers Detection
- Decision Trees
  - Regression Tree
  - Function Approximation
- Bagging
- Random Forest

Outline

- Comments on the project proposals
- Boosting
  - AdaBoosting
  - Gradient Boosting (Gboost)
  - eXtreme Gradient Boosting (XGBoost)
- Quiz #2
- Neural Network
  - Single-Layer vs. Multi-layer
  - Activation Functions
  - Backpropagation Algorithm

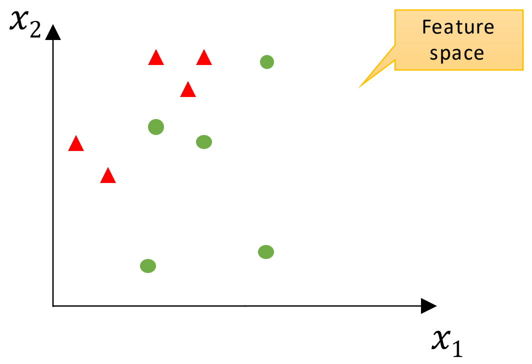
Boosting

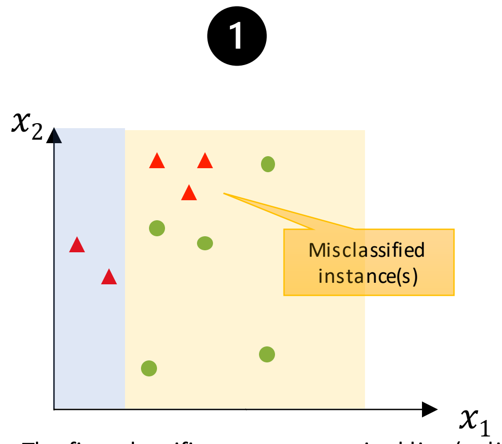
- Boosting is another ensemble technique that aims to create a strong classifier out of several weak classifiers.
- By 2016, boosting algorithms, particularly those implementing trees, had become very popular in Kaggle competitions due to their effectiveness in handling a wide range of data types and problems and their ability to produce highly accurate models.
- Boosting works great for small and medium-sized data compared to **Random Forests** and can be flexible and accurate with little work.

Boosting (cont.)

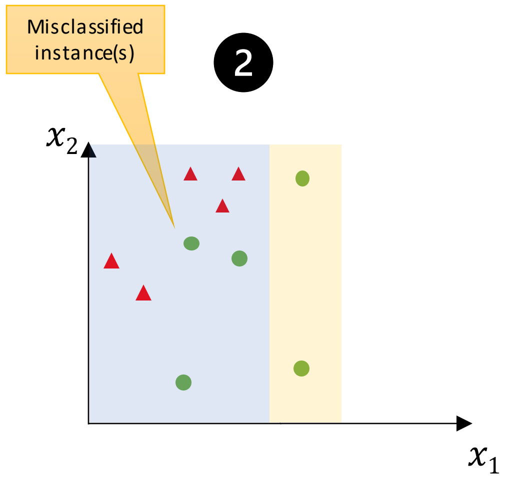
- Boosting is a **sequential** process. It slowly learns from data and tries to improve its prediction in subsequent iterations.
- Boosting starts with a weak model and **iteratively adds** new models that correct the errors of the combined existing models.
- Combine simple classifiers additively so that the final classifier is: $\operatorname{sign}\left(\sum_{i=1}^{n} \overset{\text{weights}}{\alpha_i} \cdot h_i(x)\right)$
- The "votes" in the final combination are weighted (α_i) to reflect the accuracy of each weak learner.
- Classifiers that are more accurate in predicting the training data get higher weights.

Boosting (cont.)

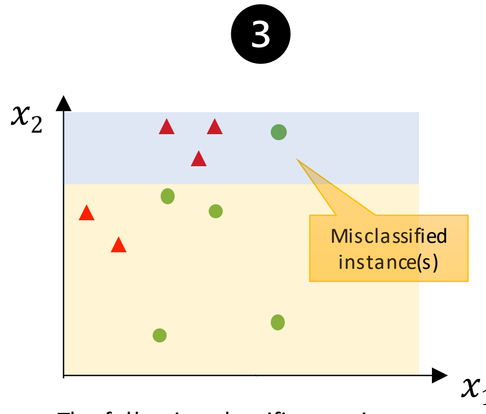




- The first classifier creates a vertical line (split).

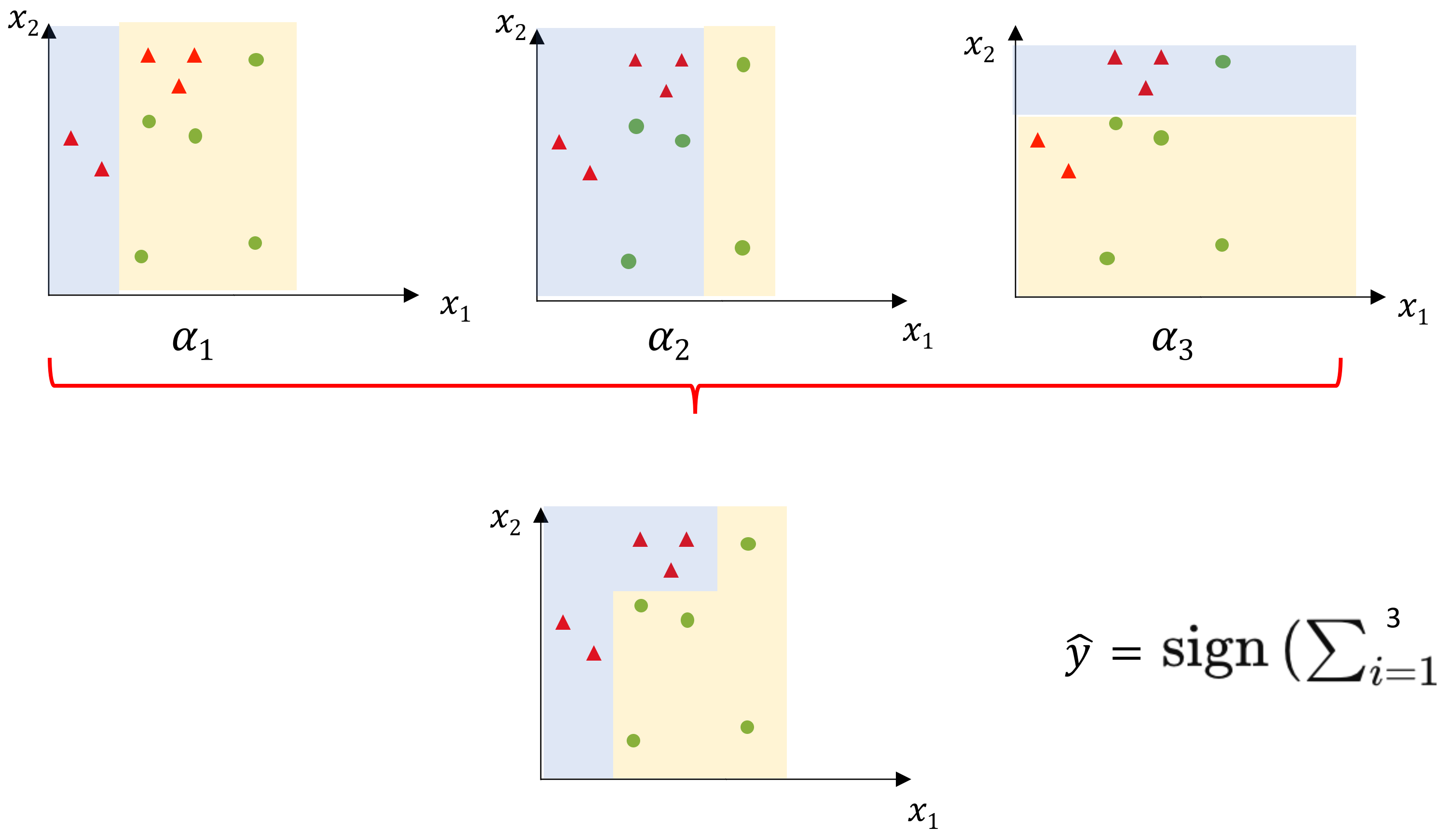


- The second classifier tries to correct the errors from the first classifier.
- It creates another vertical line (split).



- The following classifier continues to correct the errors.

Boosting (cont.)



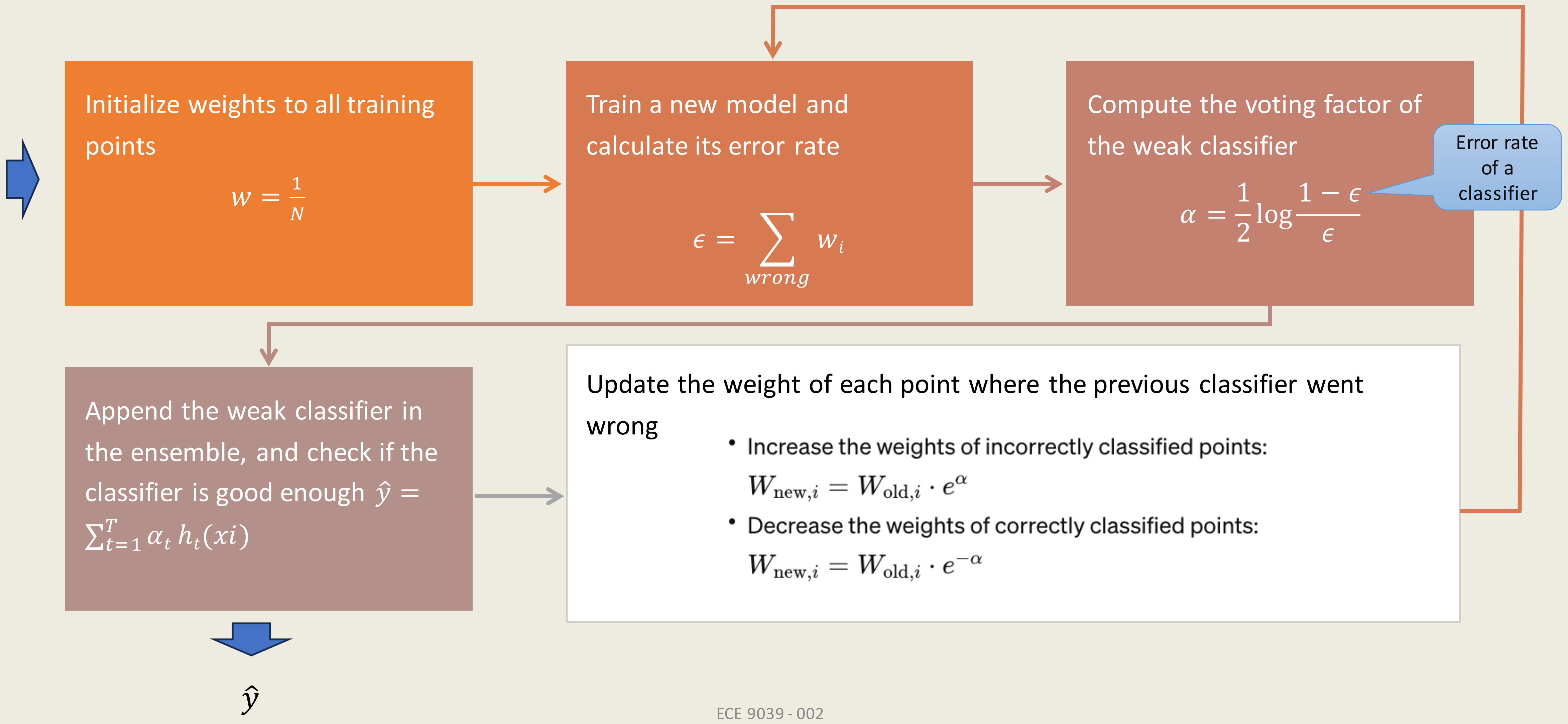
$$\hat{y} = \text{sign}\left(\sum_{i=1}^{3} \alpha_i \cdot h_i(x)\right)$$

[Bhavesh Bhatt]

AdaBoosting

- AdaBoost (**Ada**ptive **Boosting**) is one of the first practical boosting algorithms.
- AdaBoost's "adaptive" part refers to adjusting the weights of incorrectly classified instances, allowing subsequent weak learners to focus more on the difficult points.
- In AdaBoost, the weak learners (classifiers) are typically decision tree stumps (trees with a single split).
  - Each stump is made to correct the errors of its predecessors by paying more attention to the previously misclassified instances.
- This process focuses the learning on the **hardest-to-classify** instances.

AdaBoosting Algorithm



Initialize weights to all training points

$$w = \frac{1}{N}$$

Train a new model and calculate its error rate

$$\epsilon = \sum_{wrong} w_i$$

Compute the voting factor of the weak classifier

$$\alpha = \frac{1}{2}\log\frac{1-\epsilon}{\epsilon}$$

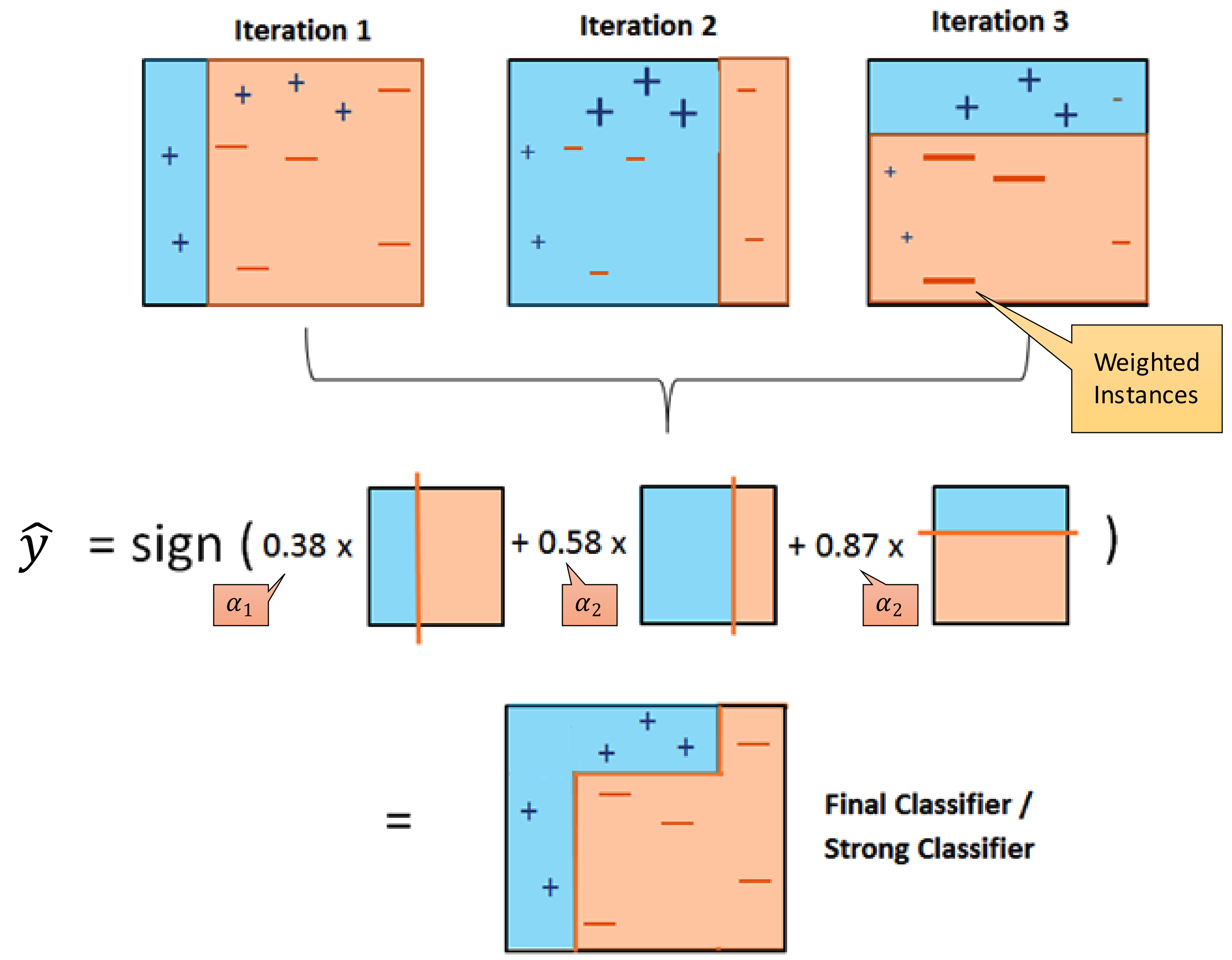
Append the weak classifier in the ensemble, and check if the classifier is good enough $\hat{y} = \sum_{t=1}^{T} \alpha_t \, h_t(xi)$

Update the weight of each point where the previous classifier went wrong

- Increase the weights of incorrectly classified points:
  $W_{\text{new},i} = W_{\text{old},i} \cdot e^{\alpha}$
- Decrease the weights of correctly classified points:
  $W_{\text{new},i} = W_{\text{old},i} \cdot e^{-\alpha}$

$\hat{y}$

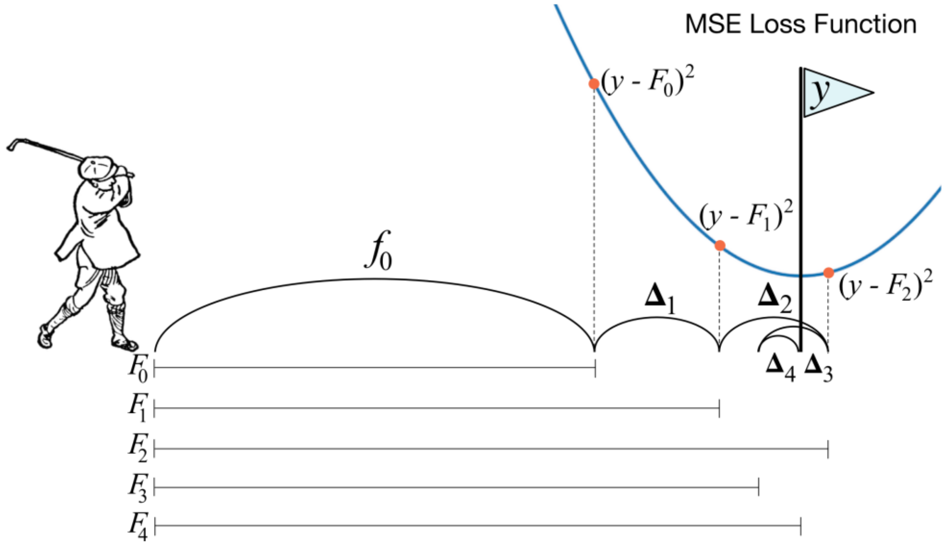
AdaBoost Classifier



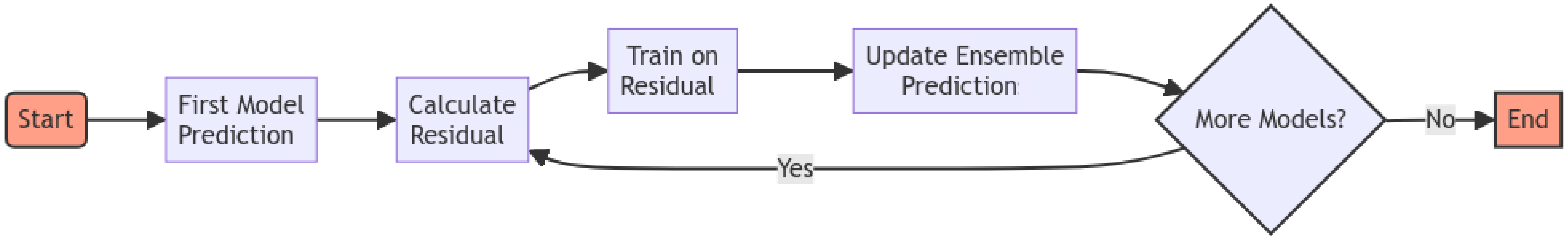
Source: Dangeti, P. (2017) *Statistics for Machine Learning*. Birmingham: Packt.

Gradient Boosting (Gboost)

- The “gradient” represents how much you adjust your prediction based on the errors made by previous models, making it a powerful ensemble method.
- The primary difference between **AdaBoosting** and **Gradient Boosting** lies in their update strategies. AdaBoosting focuses on adjusting data point weights, while Gradient Boosting optimizes the model’s predictions based on residual errors.



Source: https://explained.ai/gradient-boosting/



Example

Sample Data:

- House Size (sq ft): [600, 800, 1000, 1200, 1400]
- Actual Price ($): [300000, 350000, 500000, 600000, 650000]

Initial Model:

- Calculate the average price of the houses:

$$\text{Average Price} = \frac{300000 + 350000 + 500000 + 600000 + 650000}{5} = 480000$$

- So, the initial model h_0 predicts every house to cost $480,000, regardless of its size (***h*₀. = $480,000**).
- The residuals are the differences between the actual prices and the predicted prices from our initial model:

r_0=[300000,350000,500000,600000,650000]−480000
=[−180000,−130000,20000,120000,170000]

- These residuals represent the errors of our initial model.

Train a Decision Tree on Residuals:

- Now, train a decision tree to predict these residuals [−180000,−130000, 20000, 120000, 170000] from the house size (sq ft): [600, 800, 1000, 1200, 1400]

For simplicity, let's say our decision tree makes the following splits:

- For house sizes < 900 sq ft, predict residual = -155000 (average of -180000 and -130000)
- For house sizes ≥ 900 sq ft, predict residual ≈ 103333.3 (average of 20000, 120000, and 170000)

Update Predictions:

- We update our model's predictions by adding the predicted residuals to the initial predictions. So, for each house size category:
  - For house sizes < 900 sq ft:

New prediction ($\boldsymbol{h_1}$)= 480000 + η×(−155000) = 480000 − 0.1×155000 = 480000−15500 = 464500

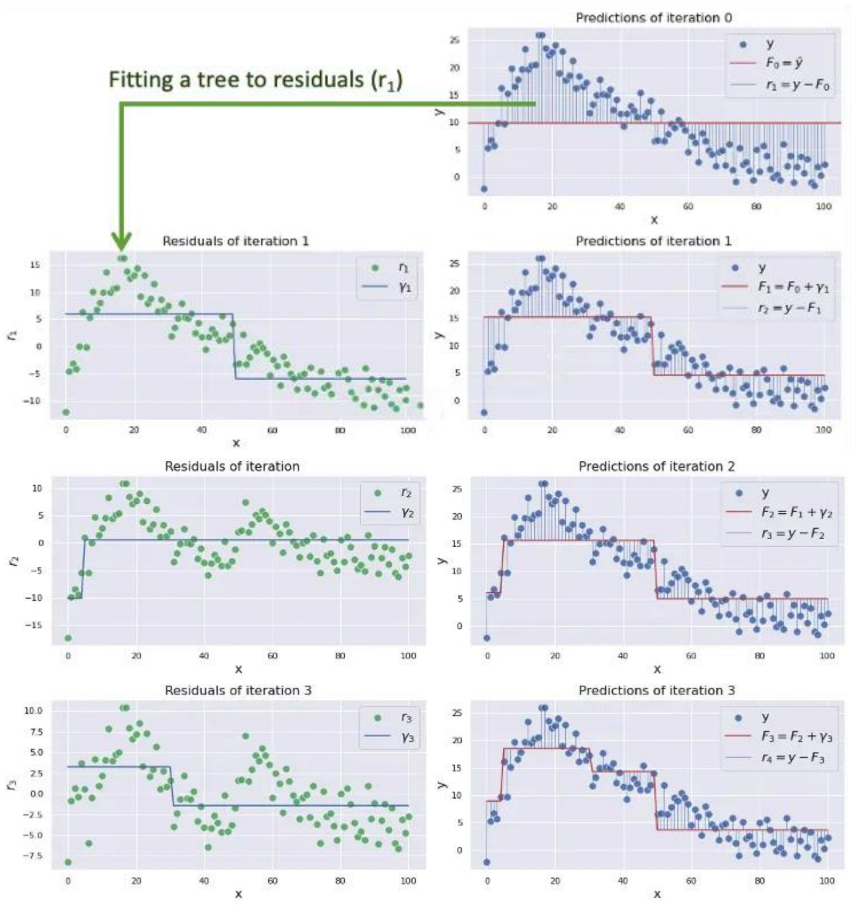
  - For house sizes ≥ 900 sq ft:

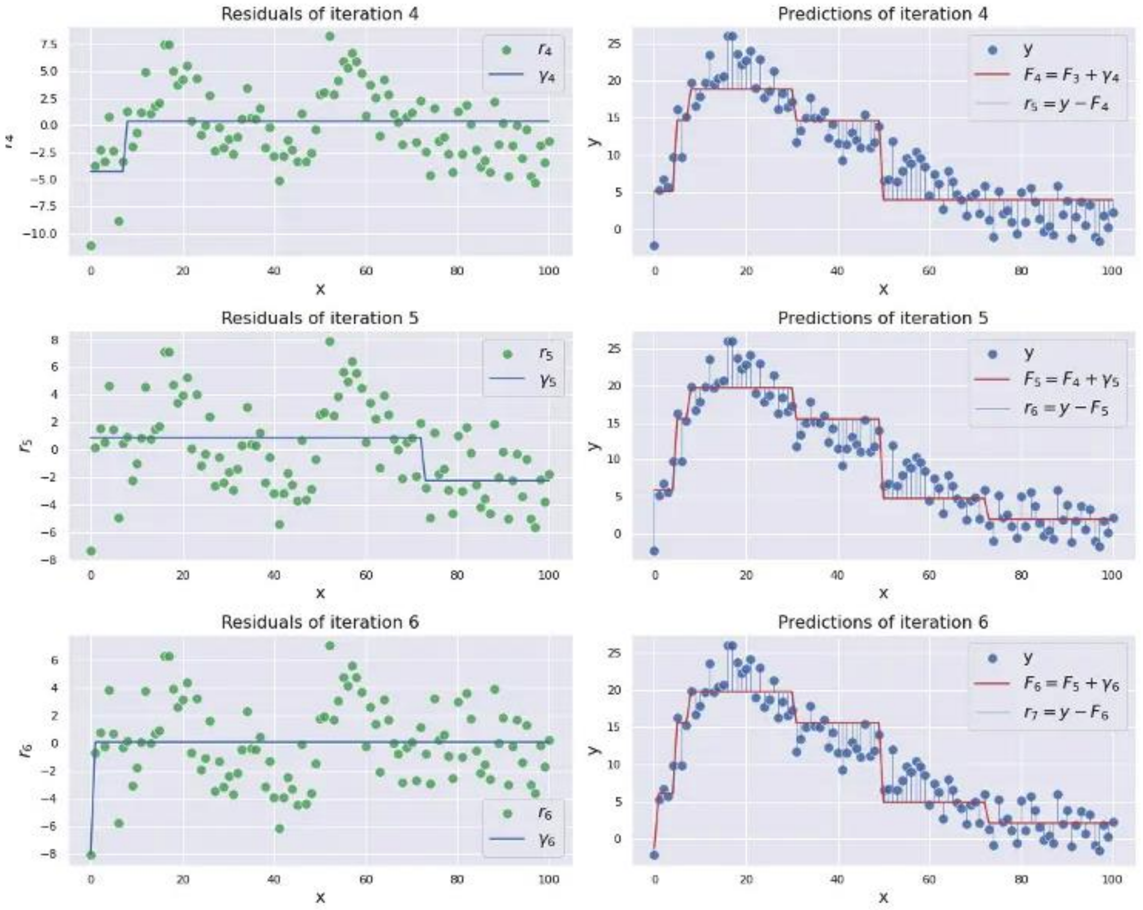
New prediction ($\boldsymbol{h_1}$)= 480000+ η×103333.3 = 480000+ 0.1×103333.3 = 480000 + 10333.33 = 490333.3

So, with a learning rate of 0.1, the updated predictions after applying the first decision tree (h_1) to correct the initial model (h_0) are:

- $464,500 for house sizes < 900 sq ft.
- $490,333 for house sizes ≥ 900 sq ft.

Gradient Boosting Steps





Source: https://towardsdatascience.com/all-you-need-to-know-about-gradient-boosting-algorithm-part-1-regression-2520a34a502

Gradient Boosting Algorithm

1. Initialize model h_0 with a constant value (*e.g.* average):

$$\underset{h}{argmin} \sum_i \mathcal{L}(y, h(X))$$

Prediction

2. Compute residuals:

loss function

$$r_0 = \frac{\partial \mathcal{L}(y,h(X))}{\partial h(x_i)} = \sum_i (h(x_i) - y)$$

$$r_0 = -\frac{\partial \mathcal{L}(y, h_0(X))}{\partial h_0(X)} \quad where \quad \mathcal{L}(y, h(X)) = \frac{1}{2}\sum_i (y - h(x_i))^2$$

3. Find nonlinear predictor (fit residuals to a tree h_1)

Learning rate

4. Update the model (fit next tree to updated outputs): $h_2 = h_{2-1} - \eta \frac{\partial \mathcal{L}}{\partial h_{2-1}}$

5. …. continue for m trees: $h_m = h_{m-1} - \eta \frac{\partial \mathcal{L}}{\partial h_{m-1}}$

Gradient Boosting vs. Regression Tree

- Regression Tree – A single model representing a series of decision rules based on the input features.
- Gradient Boosting – A combination of multiple regression trees where each tree corrects the errors of the ensemble to date, leading to a potentially more accurate and generalized model.

eXtreme Gradient Boosting (XGBoost)

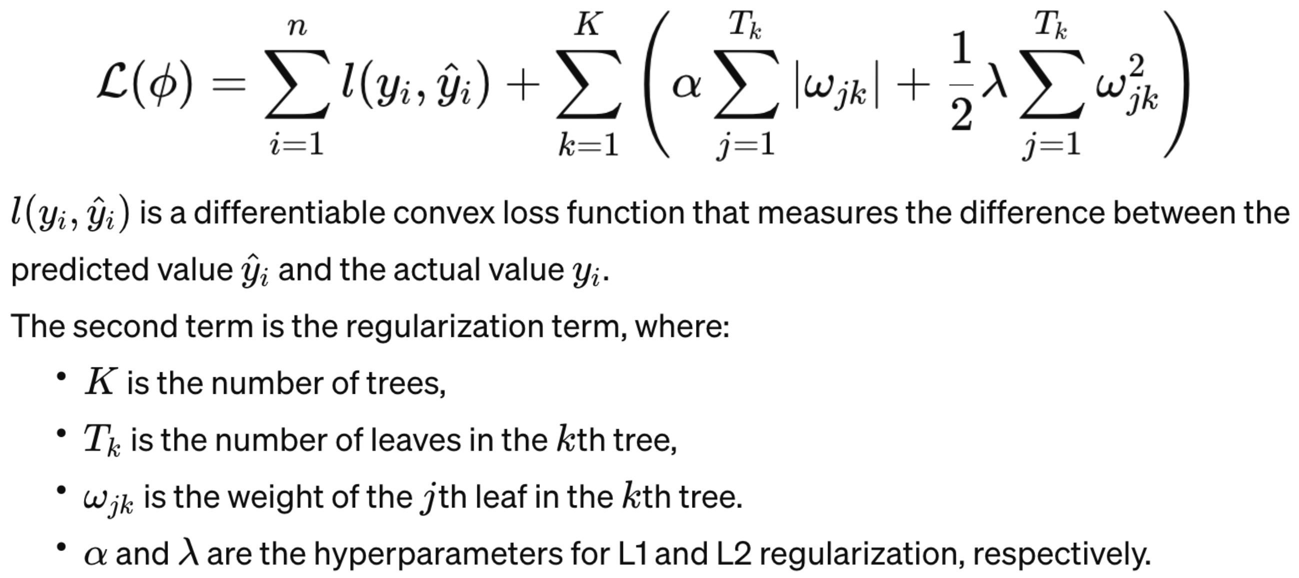
- XGBoost is often faster and can handle larger datasets better than standard Gradient Boosting; XGBoost **parallelizes** constructing a single tree??
  - It can simultaneously process multiple features across multiple cores to find the best split point for each feature. This is done during the construction of each tree (**Feature Parallelism**).
- XGBoost can distribute the dataset across different cores, where each holds a subset of the data. Each computes the gradients on its subset of the dataset. Then, the results are aggregated to decide on the best split. This approach allows for handling larger datasets that might not fit into the memory of a single machine (**Data Parallelism**).
- While traditional Gradient Boosting stops growing the tree when it encounters a **negative loss** in the split, XGBoost grows the tree up to a **specified max-depth** and then **prunes** back the tree using the **gained** information.
- XGBoost has a built-in routine to **handle missing values**, allowing the model to learn the best imputation value for missing data based on a reduction in loss.
- It has a built-in **cross-validation capability** at each iteration, which makes it easy to get the most optimized model.

Regularization in XGBoost

- XGBoost includes **regularization parameters** that can reduce overfitting, improving model performance on unseen data.
- Regularization penalizes more complex models, often by adding a penalty term to the loss function that the learning algorithm optimizes.
- In the context of linear models, XGBoost uses $L1$ and $L2$ regularization, also known as Lasso and Ridge Regression.
    - For a linear model, the cost function with L1 regularization (Lasso) is: $\mathcal{J}(\theta) = \mathrm{MSE}(\theta) + \alpha \sum_{j=1}^{n} |\theta_j|$
    - For a linear model, the cost function with L2 regularization (Ridge) is: $\mathcal{J}(\theta) = \mathrm{MSE}(\theta) + \lambda \sum_{j=1}^{n} \theta_j^2$
- In the context of XGBoost, these regularization techniques are applied to the trees (ensemble of decision trees) that make up the model.
- By including these regularization terms in the cost function that XGBoost optimizes, the algorithm is encouraged to keep the model simpler. Simpler models are less likely to overfit, meaning they perform better on unseen data.

Regularization in XGBoost

- For XGBoost, the regularization terms are added to the traditional cost function used in boosting, which results in the following regularized objective:



$$\mathcal{L}(\phi) = \sum_{i=1}^{n} l(y_i, \hat{y}_i) + \sum_{k=1}^{K} \left(\alpha \sum_{j=1}^{T_k} |\omega_{jk}| + \frac{1}{2}\lambda \sum_{j=1}^{T_k} \omega_{jk}^2 \right)$$

$l(y_i, \hat{y}_i)$ is a differentiable convex loss function that measures the difference between the predicted value $\hat{y}_i$ and the actual value y_i.

The second term is the regularization term, where:

- K is the number of trees,
- T_k is the number of leaves in the kth tree,
- ω_{jk} is the weight of the jth leaf in the kth tree.
- α and λ are the hyperparameters for L1 and L2 regularization, respectively.

- If you are interested in more details, see chapters 9, 10, and 15 of *Elements of Statistical Learning*

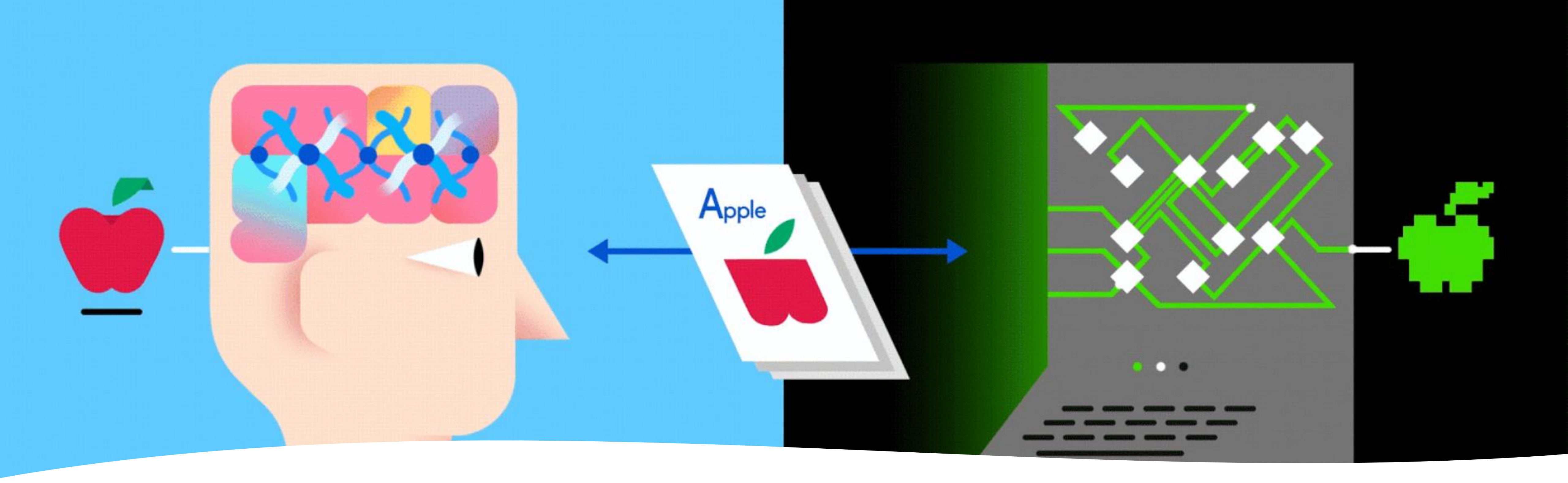
Common XGBoost parameters with sci-kit-learn:

- **General Parameters**:
    - **booster** – The type of model to run at each iteration. It can be gbtree (tree-based models), or gblinear (linear models).
- **Booster Parameters**:
    - **n_estimators**: Number of gradient-boosted trees.
    - **learning_rate** (or eta): Step size shrinkage used to prevent overfitting. Range is [0,1].
    - **max_depth**: Maximum depth of a tree.
    - **reg_alpha**: L1 regularization term on weights (equivalent to Lasso regression).
    - **reg_lambda**: L2 regularization term on weights (equivalent to Ridge regression).
- **Learning Task Parameters**:
    - **eval_metric**: Evaluation metrics for validation data, a default metric will be assigned according to the objective (rmse for regression, and error for classification, mean average precision for ranking).
    - **seed**: Random number seed. (Can be used for creating reproducible results).

Attendance

You can use the provided link if you don't have a mobile phone or if your phone lacks a QR-Code reader – https://forms.gle/WDbsq7ETC5BSNZ7w8

Artificial Neural Network (ANN)



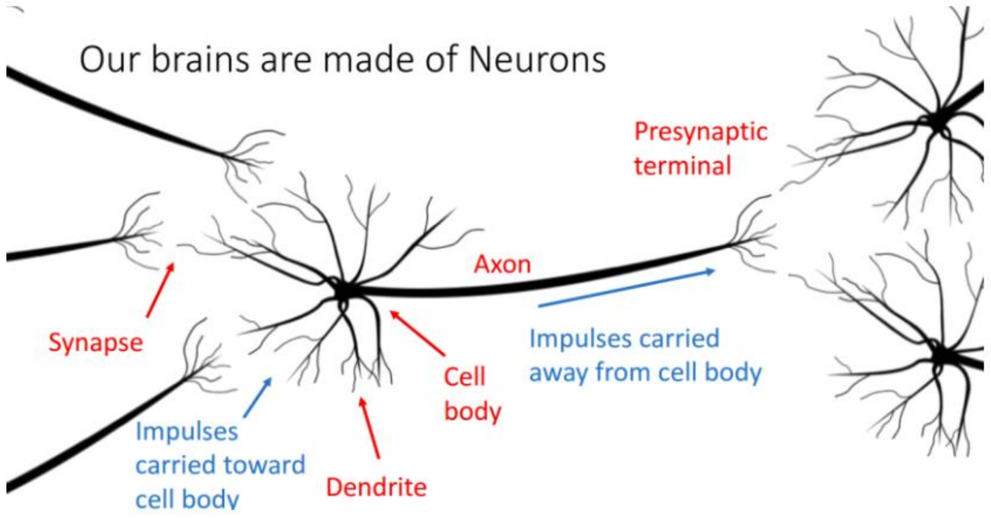
Neural Networks: The Biological Inspiration

- Machines can do tasks that typically require **human intelligence**
- **Learn by experience** and acquire skills without human involvement
- Inspired by the human brain, we can learn from experience by performing a task repeatedly

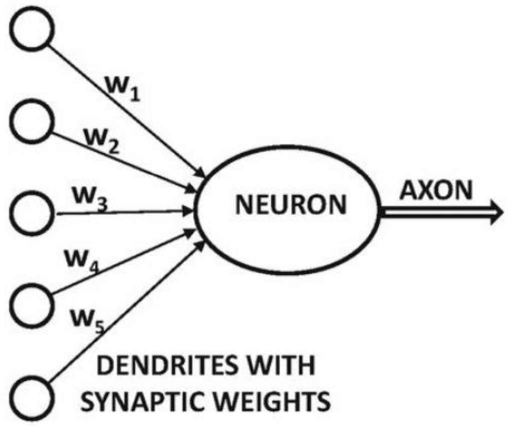
Neural Networks (NN): The Biological Inspiration

- Neural networks contain **computation** units ⇒ **Neurons**.
- The computational units are connected to one another through **weights** ⇒ Strengths of synaptic connections in biological organisms.
- Each input to a neuron is <u>scaled with a weight</u>, which affects the function computed at that unit.

NNDL: C. Aggarwal. Neural Networks and Deep Learning, Springer 2018 [Free through Western]. https://link-springer-com.proxy1.lib.uwo.ca/book/10.1007/978-3-319-94463-0

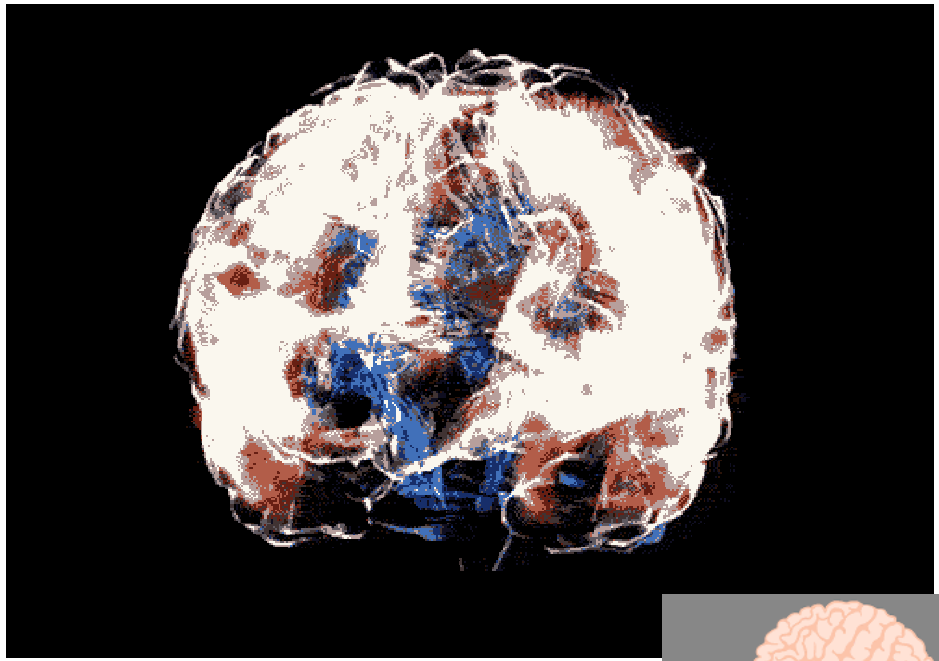

Our brains are made of Neurons
Presynaptic terminal
Axon
Synapse
Impulses carried away from cell body
Cell body
Impulses carried toward cell body
Dendrite

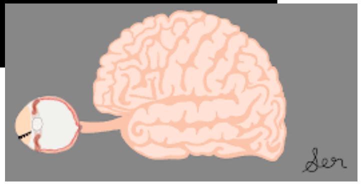
a) Biological neural network

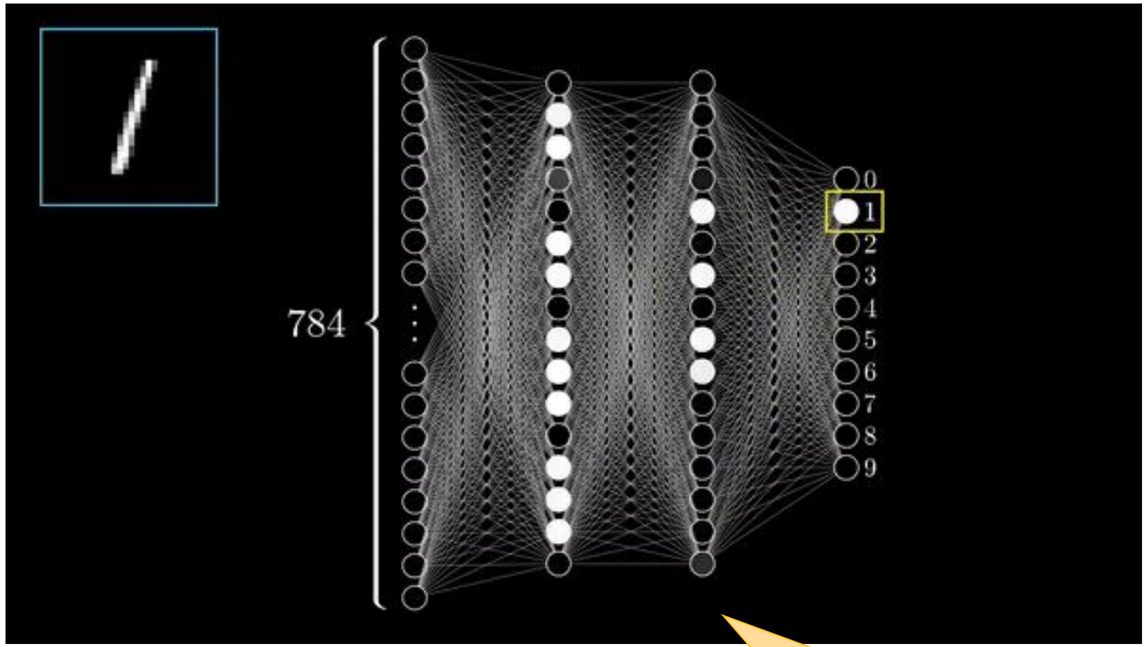


b) Artificial neural network

Biological vs. Artificial



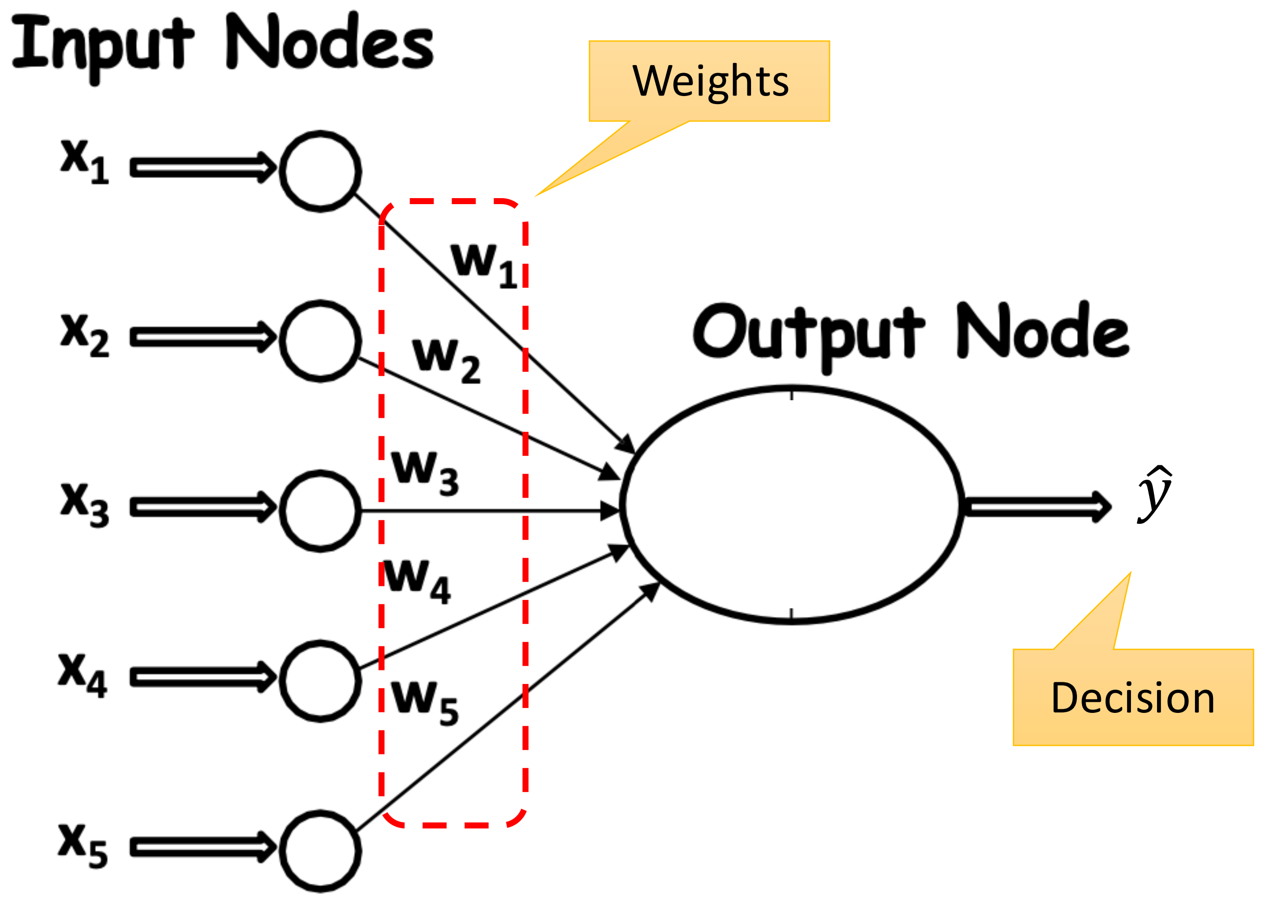




An artificial neural network tries to recognize handwritten numbers.

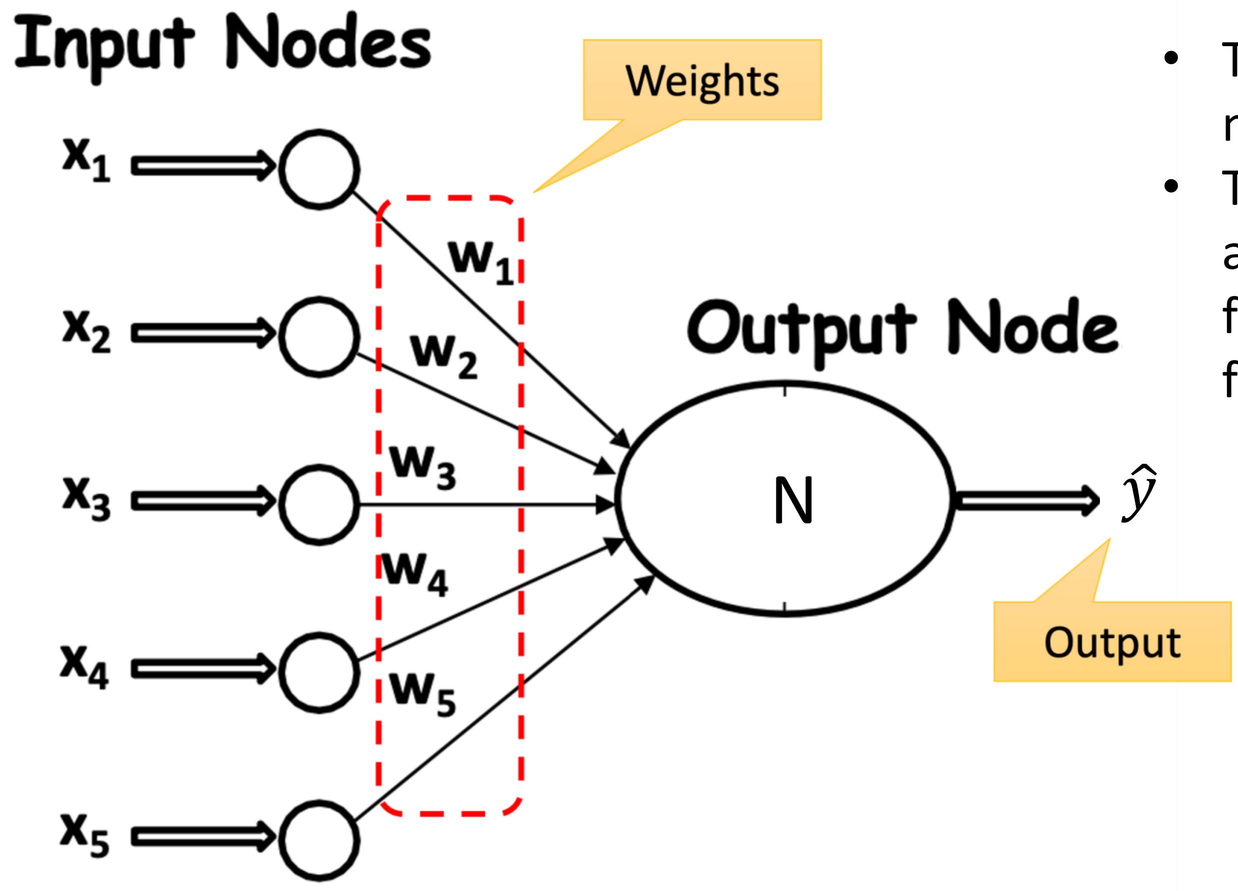
ANN Computational Unit!

ANN Computational Unit: The Perceptron



- It takes several inputs and gives a single binary output (**Decision**).
- The input nodes that transmits the k features. With edges of weights $\mathbf{w} = [w_1 \ldots. w_k]$ to an output node.

The Perceptron



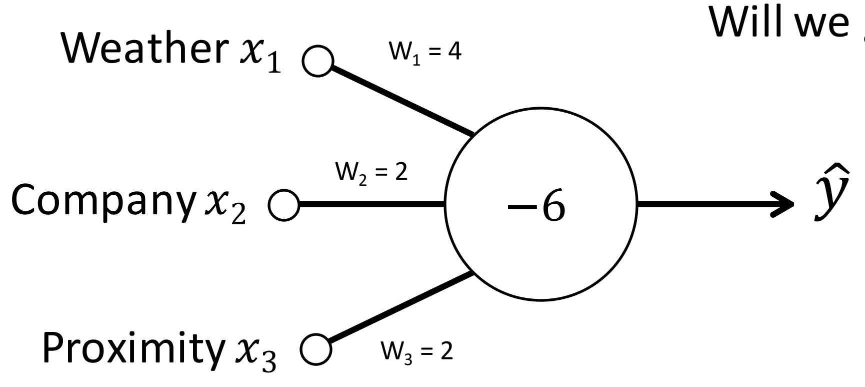
- Two functions are computed at the output nodes
- The value computed by the 1st function (f) is applied to a second function Φ (Phi). The Φ function is referred to as the activation function.

$$\hat{y} = \begin{cases} -1 & if\ \sum_j w_j x_j + b \le 0 \\ 1 & if\ \sum_j w_j x_j + b > 0 \end{cases}$$

bias

Where, w_j = weights
And b = bias (threshold)

Example



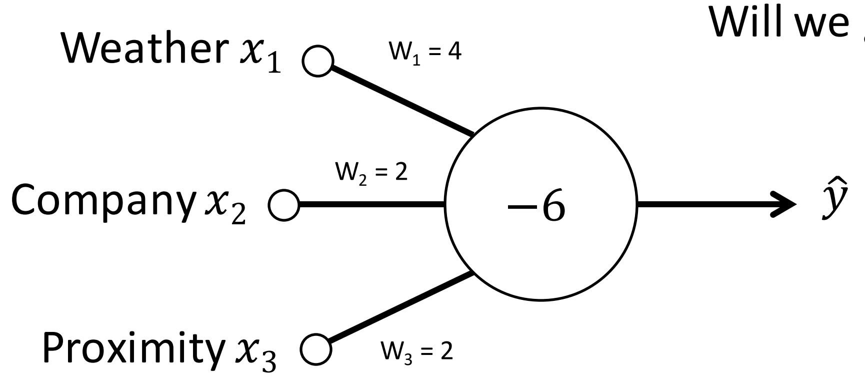
Will we go to a movie? No (-1)

- If the weather is bad (x_1=0), there is a company (x_2= 1) and the theater is nearby (x_3=1)
- Thus,
- $\hat{y} = sign(x_1 w_1 + x_2 w_2 + x_3 w_3 + b)$
- $\hat{y} = sign\big(0 \times 4 + 1 \times 2 + 1 \times 2 + (-6)\big) \Rightarrow -1$

x_1: Is the weather good?
x_2: Do we have company?
x_3:Is the theater nearby?

- <u>Assumptions</u>:
  - x_1 is double important compared to other features.
  - Visit the cinema only if the weather is good and at least one other condition is positive.
  - Bias (threshold) $b = -6$

Example (cont.)



Will we go to a movie? Yes (1)

- If the weather is bad (x_1=1), there is a company (x_2= , 1) and the theater is nearby (x_3=1)
- Thus,
- $\hat{y} = sign(\underbrace{x_1 w_1 + x_2 w_2 + x_3 w_3 + b}_{\text{pre-activation value}})$
- $\hat{y} = sign\big(1 \times 4 + 1 \times 2 + 1 \times 2 + (-6)\big) \Rightarrow 1$

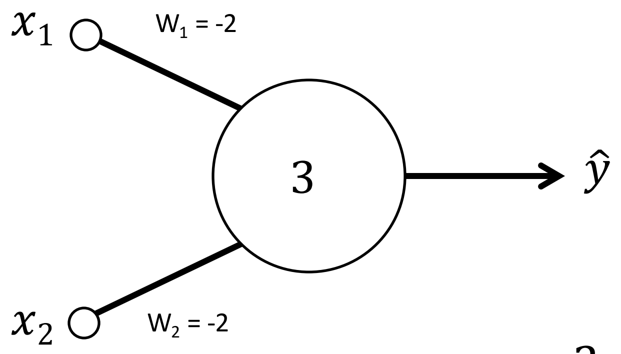
post-activation value

x_1: Is the weather good?
x_2: Do we have company?
x_3:Is the theater nearby?

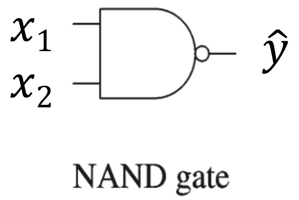
- Assumptions:
  - x_1 is double important compared to other features.
  - Attend the cinema only when the weather conditions are favorable, and all other relevant aspects are also positive.
  - Bias (threshold) $b = -6$

The value computed before applying the activation function ϕ will be referred to as the *pre-activation value*
The value computed after applying the activation function is referred to as the *post-activation value*

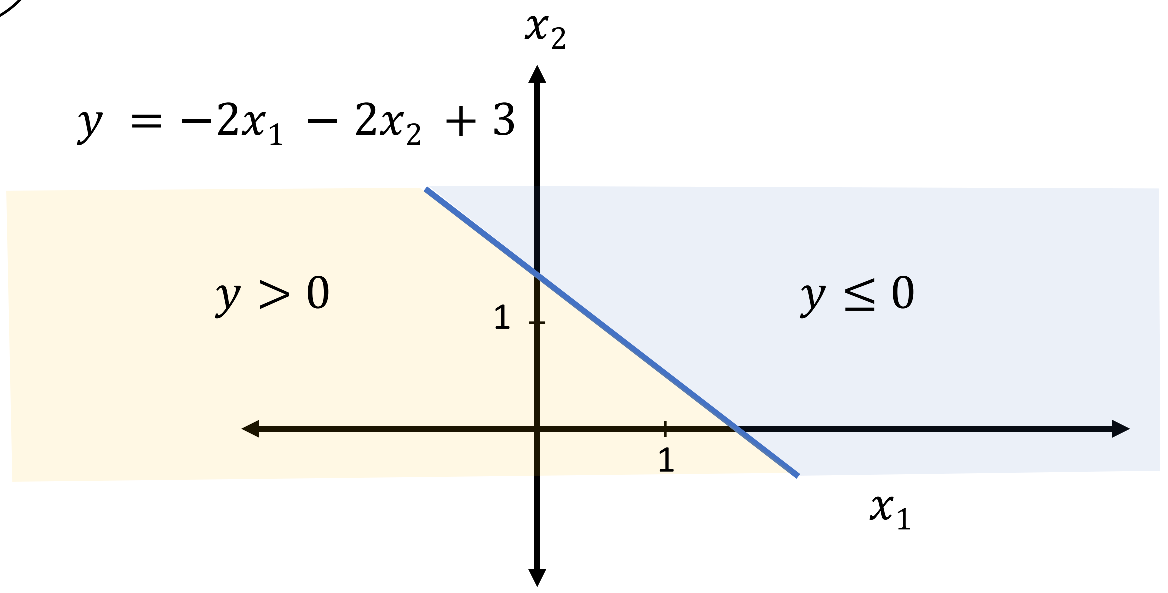
Perceptron as a Linear Classifier



$$\hat{y} = \begin{cases} -1 & if \sum_j w_j x_j + b \le 0 \\ 1 & if \sum_j w_j x_j + b > 0 \end{cases}$$



NAND gate

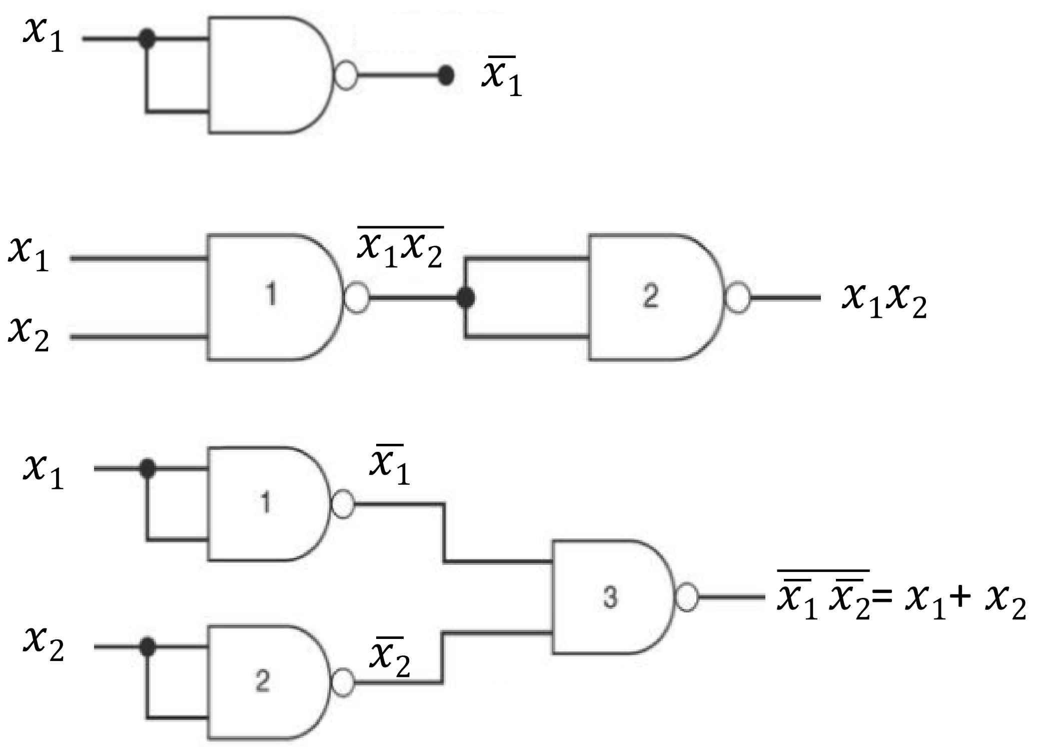


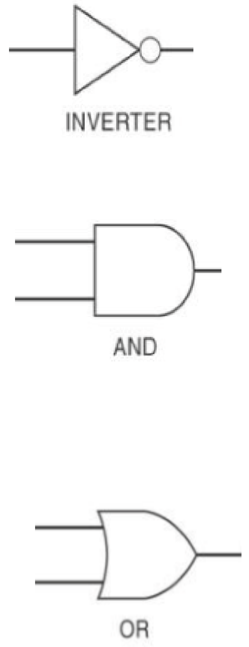
x_1	x_2	$\hat{y}$
0	0	1
0	1	1
1	0	1
1	1	0

Perceptron as a NAND gate

Shree K. Nayar, Dr
Columbia University

Universal Logic Gates



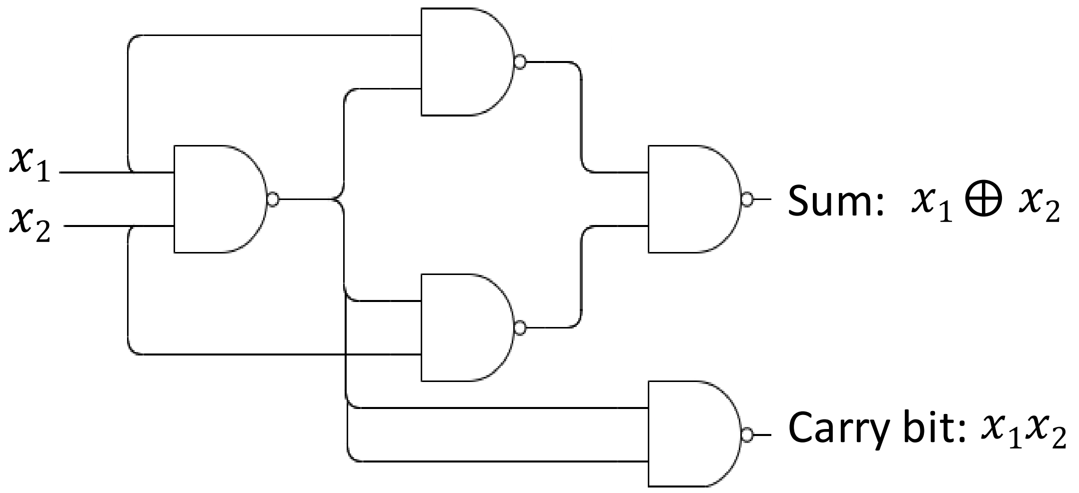


- Hence, perceptrons have the capacity for universal computation, allowing for the creation of any digital logic circuit by utilizing them.

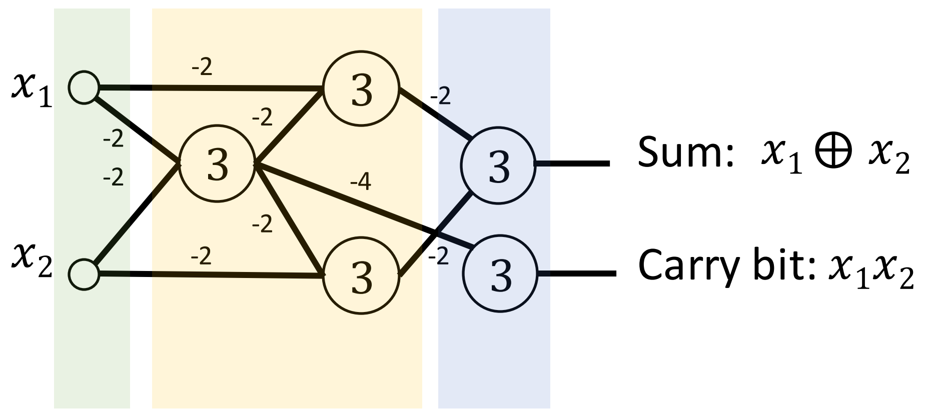
[Shree K. Nayar, Dr
Columbia University]

1–bit Adder

Digital Logic Circuit



Equivalent Perceptron Network



Multi-layer Network

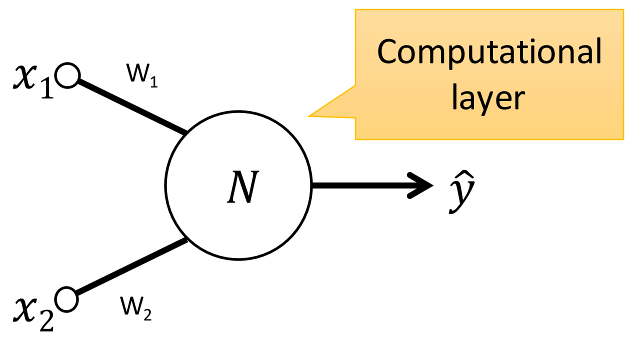
Legend

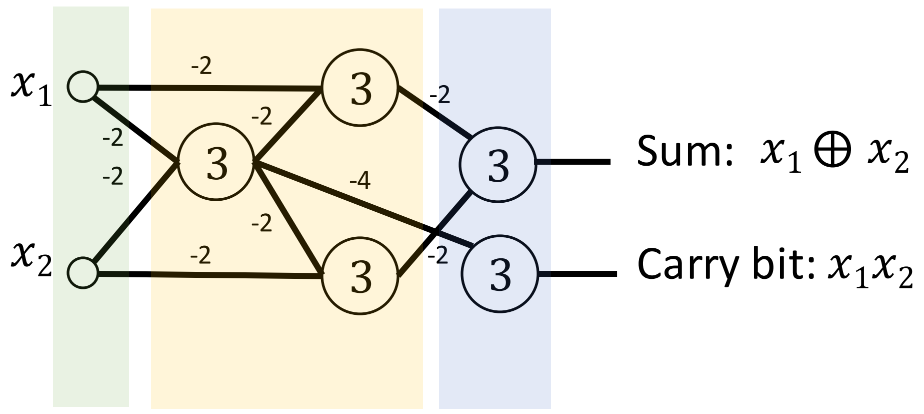
🟩	Input Layer
🟨	Hidden Layer
🟦	Output Layer

- Machines can do tasks that typically require **human intelligence**.

Shree K. Nayar, Dr
Columbia University

Single-layer Network vs. Multi-layer Network



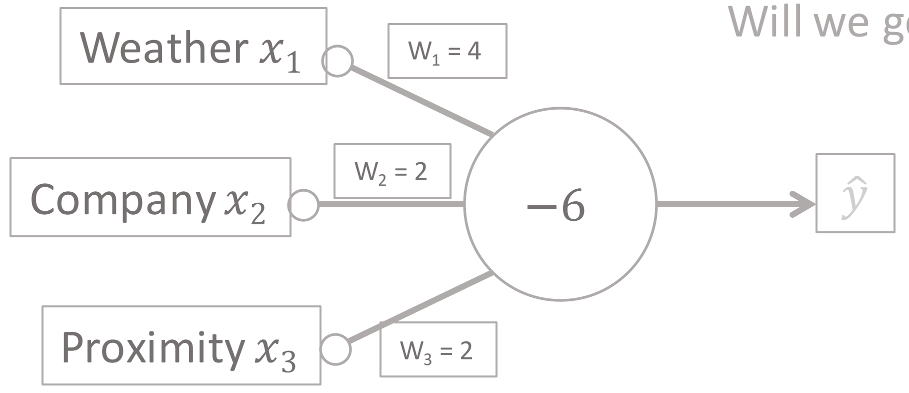


- The Perceptron is a single-layer network.
- The input layer is not included in the count of the number of layers in a neural network (it does not perform any computation)
- Since the perceptron contains a single computational layer, it is considered a single-layer network.

- Multi-layer Perceptrons (MLPs) are a class of **feedforward ANN** that consists of multiple layers of nodes, each layer fully connected to the next one.
- MLPs can model complex, non-linear relationships in data and are used in various applications, from speech recognition to image classification.

Activation Functions

Recall: Example



x_1: Is the weather good?
x_2: Do we have company?
x_3:Is the theater nearby?

Will we go to the movie? No (-1)

- If the weather is bad (x_1=0), there is a company (x_2= , 1) and the thea
- Thus,
- $\hat{y} = sign(x_1\, w_1 + \; x_2\, w_2 + \; x_3\, w_3 \; + \; b)$
- $\hat{y} = sign\big(0 \times 4 \; + \; 1 \times 2 \; + \; 1 \times 2 + (-6)\big) \Rightarrow \; -1$

Activation Function (Φ)

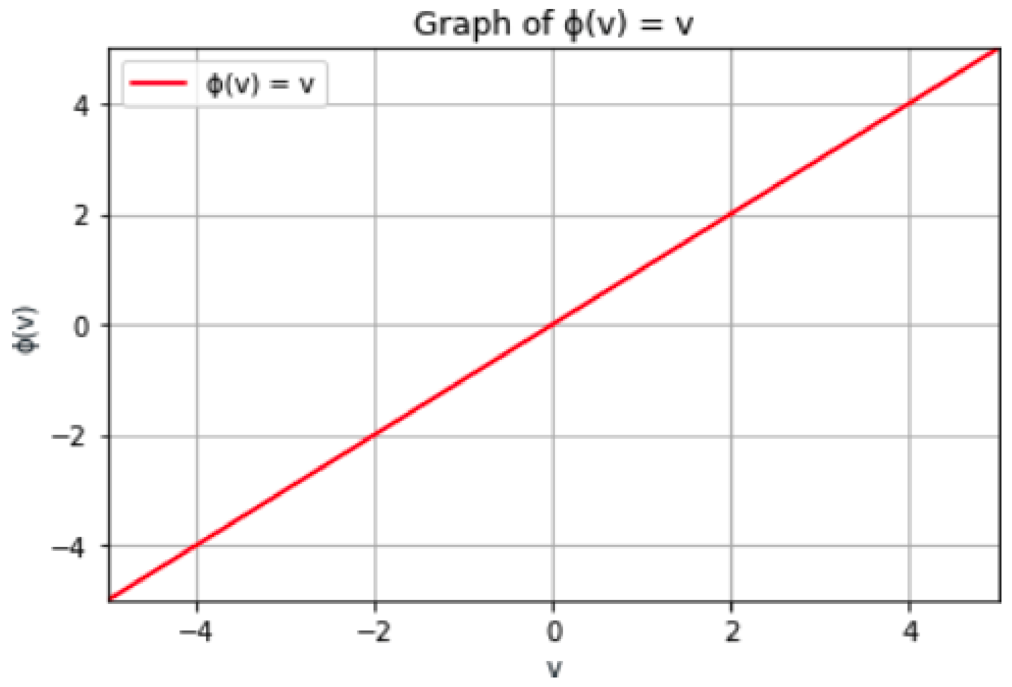
- Assumptions:
  - x_1 is double important compared to other features.
  - Visit the cinema only if the weather is good and at least one other condition is positive.
  - Bias (threshold) b = -6

Activation Functions

- The activation functions in ANN are mathematical equations that **determine** its output.
- These functions are crucial as they introduce **non-linearity** into the network, enabling it to learn complex patterns.
- Classical Activation Functions
  - Identity sign (or **signum**) activation
  - Identity or Linear Activation
  - **Sigmoid** activation is useful where the output is probability [0,1].
  - **Tanh** activation (the hyperbolic tangent function) is similar to the sigmoid function but zero-centered [-1,1].
  - **ReLu** activation – outputs the input directly if it is positive; otherwise, it outputs zero.

Identity or Linear Activation

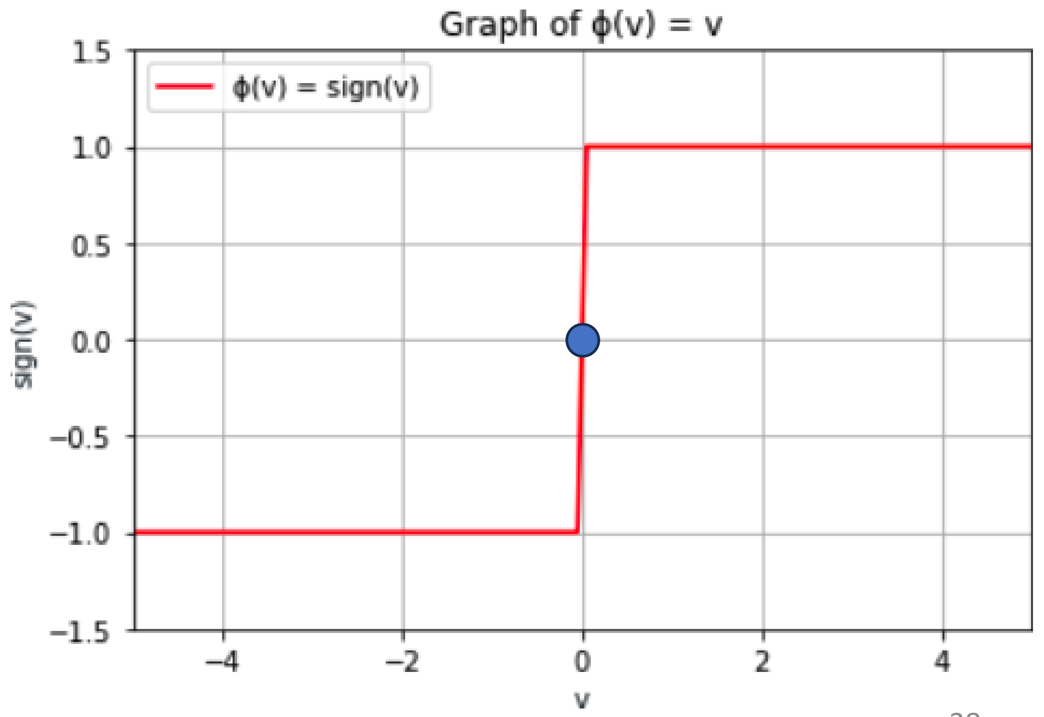
- The identity or linear activation function is a straightforward activation function used in cases where the output needs to be a linear combination of the input values.
- The function **does not bind** the output to a specific range, making it suitable for tasks where the prediction requires a range not confined to specific limits, such as regression tasks.
- **Limitation** – Using the identity function in hidden layers of a network limits the network's ability to learn complex, **non-linear relationships** in the data.



Sign (or Signum) Activation

- The sign (or signum) activation function produces a discrete output by extracting the sign of a real number.
- This function is particularly useful in models where the output needs to be binary or in scenarios requiring classification into two distinct categories.
- **Limitation** – The function is highly sensitive to small changes near the decision boundary (x=0), which can lead to instability in predictions for inputs close to this boundary.

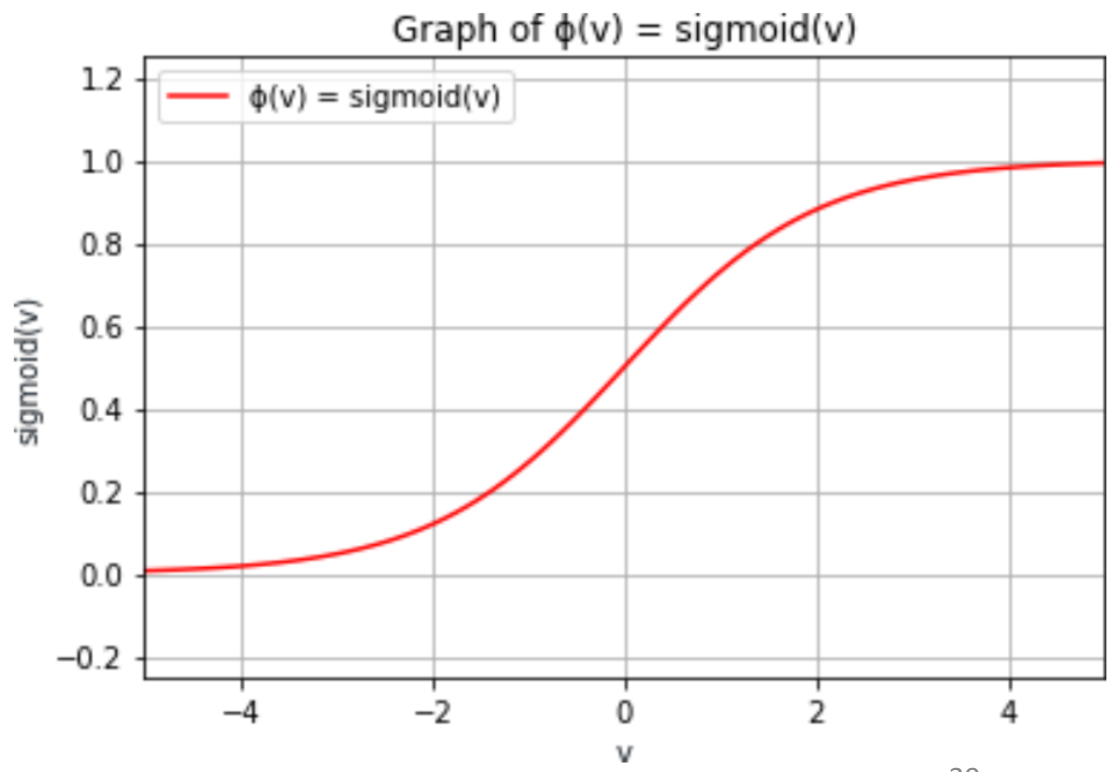
$$\phi(v) = \mathrm{sgn}(v) = \begin{cases} -1 & if\ v < 0 \\ 0 & if\ v = 0 \\ 1 & if\ v > 0 \end{cases}$$



Sigmoid Activation

- The sigmoid activation function is widely used in binary classification problems where the output must be constrained between 0 and 1.
- It's suitable for performing computations that should be interpreted as probabilities.
- **Limitation** – Large input values cause the sigmoid function to significantly slow the network's weight updates and the overall learning speed.

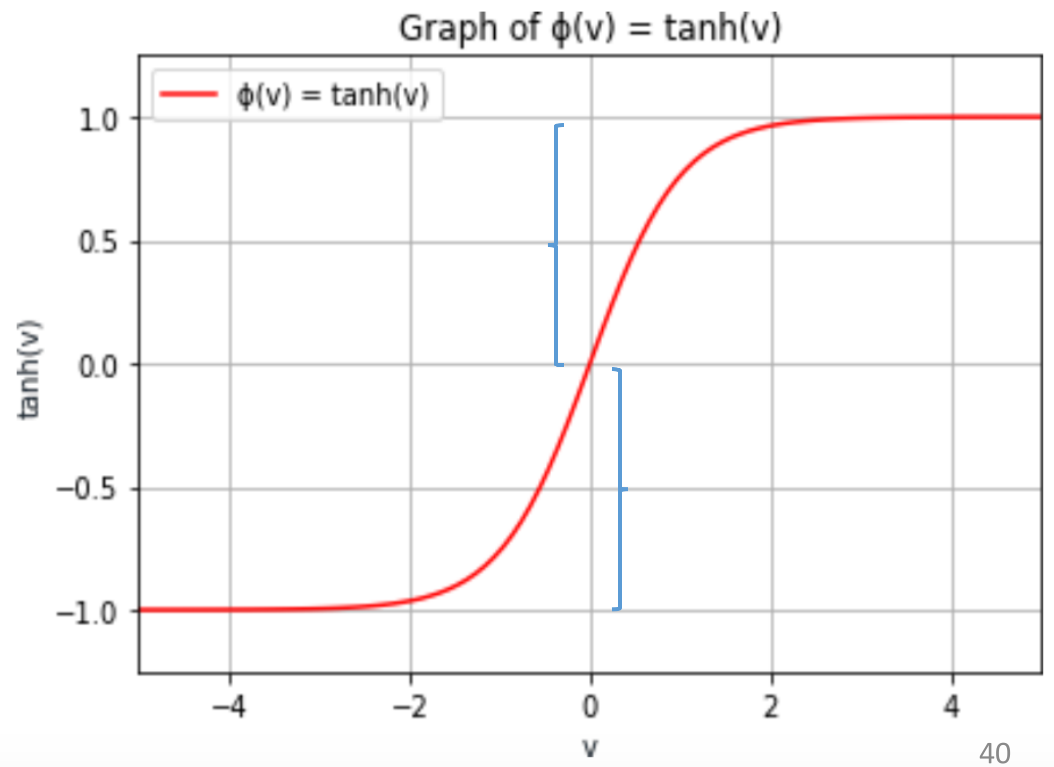
$$\phi(v) = \frac{1}{1+e^{-v}}$$



Tanh Activation

- The tanh function has a shape similar to the sigmoid function, except that it is vertically translated/re-scaled to [−1, 1].
- The tanh function is preferable to the sigmoid when the outputs of the computations are desired to be both positive and negative.
- **Limitation** – As sigmoid activation function.

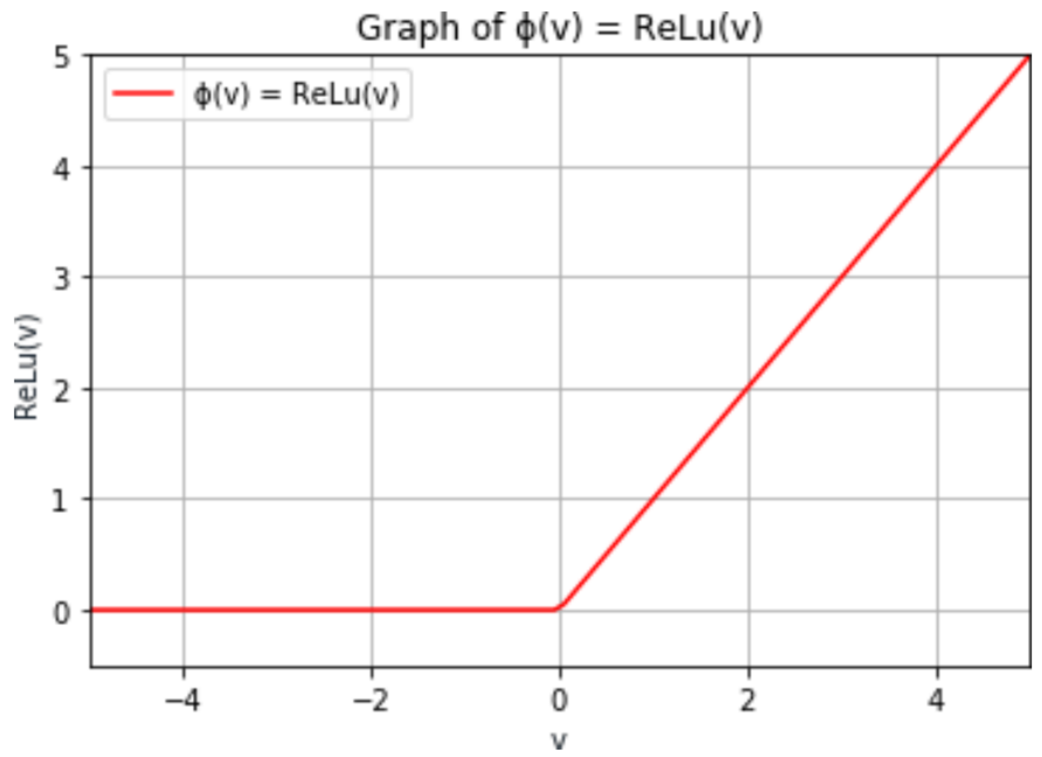
$$\phi(v) = \frac{e^{2v} - 1}{e^{2v} + 1}$$



ReLu Activation

- Piecewise linear activation function – **Re**ctified **L**inear **U**nit (**ReLu**)
- The ReLU activation function is popular in modern neural networks because of its ease of training multilayered neural networks.
- **Limitation** – For inputs to a ReLU that are consistently **negative**, the neuron may stop updating during training, a phenomenon often referred to as "**dying ReLUs**."

$$\phi\,(v) = \max\,\{v, 0\}$$



Softmax Activation

- All activation functions discussed so far map scalars to scalars. The **softmax activation** function maps vectors to vectors.
- The softmax activation function is crucial in deep learning, particularly in classification problems where the output can belong to one among **multiple classes**.
- It's used in the output layer of a neural network model to represent the **probabilities** of **each class**, ensuring that the sum of these probabilities is equal to 1 (Generalization of sigmoid activation).
- **Limitations** – The need to compute exponentials for each class score can be **computationally** more **intensive** than other activation functions, especially for a large number of classes.
- Also, **sensitivity** to **large scores**.

$$\mathrm{Softmax}(Z_i) = \frac{e^{Z_i}}{\sum_j e^{Z_j}}$$

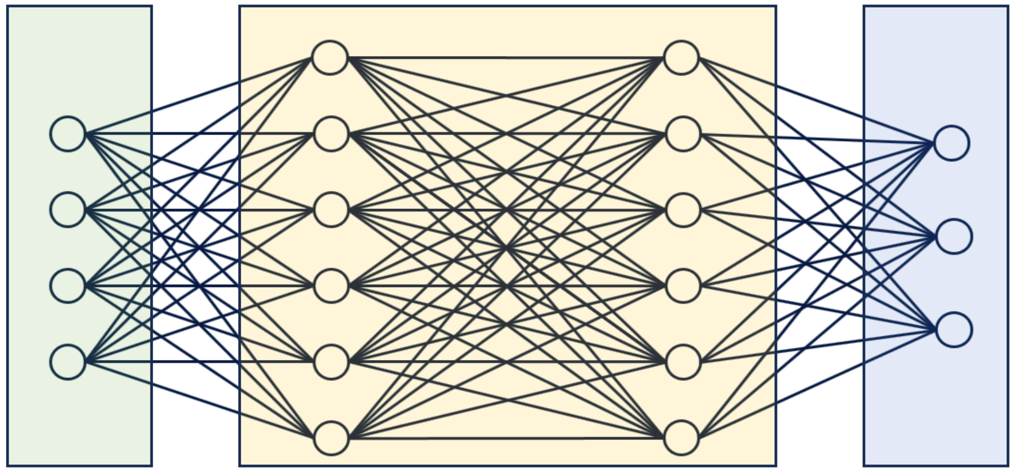
Where

- Z_i is the score for class *i* and
- e^{Z_i} is the exponential function applied to score *i*,
- The denominator is the sum of the exponentials of all raw class scores in the vector *Z*.

Feed-forward Neural Network

Feed-forward Neural Network

- A feed-forward neural network consists of multilayer neural networks that feed signals forward into a neural network and pass through different layers of the network in the form of activations, concluding in predictions at the output layer.
- The following animation illustrates how a feed-forward neural network processes input signals, categorizing them into one of three classes in the output.



- Feed-forward networks are the **simplest** form of artificial neural networks and serve as the foundation for understanding more complex neural network architectures.
- Their straightforward architecture makes feed-forward networks easier to understand and implement than complex models.
- **Limitations** – They learn a static mapping from inputs to outputs and cannot adapt to changing input patterns over time

Image source: https://vitalflux.com/feed-forward-neural-network-python-example/

Feed-forward Neural Network (cont.)

- A feed-forward neural network consists of multiple computational layers.
- $\boldsymbol{W}_1$: Matrix that contains the weights of the connections between the input layer and the first hidden layer. The size of this matrix is denoted as $k \ x \ p_1$, where $\boldsymbol{k}$ is the number of <u>input features</u>, and $\mathbf{p_1}$ is the <u>number of hidde</u> <u>nodes</u> in the first hidden layer.
- $\boldsymbol{W}_2$: Matrix that contains the weights of the connections between the hidden layer and the output layer. The size of this matrix is denoted as $o \ x \ p_2$, where $\boldsymbol{o}$ is the <u>number of output nodes</u>, and $\mathbf{p_2}$ is the <u>number of hidden nodes</u> in the second hidden layer.
- Additional $\boldsymbol{W}$ matrices if there are more hidden layers.

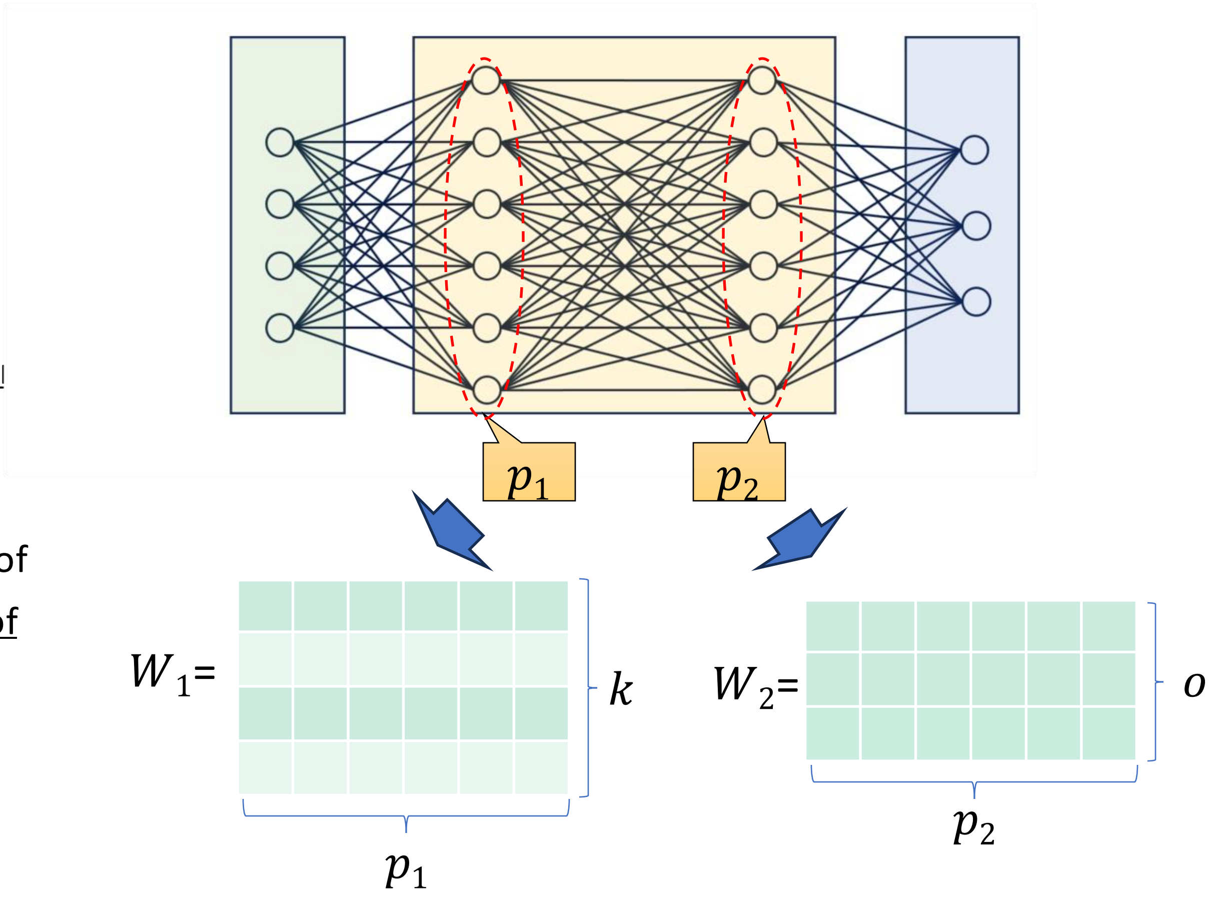
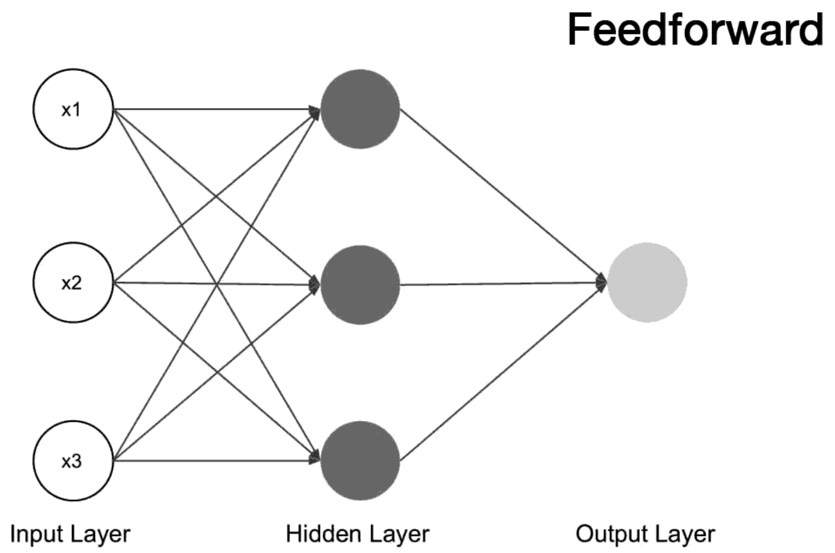


Image source: https://vitalflux.com/feed-forward-neural-network-python-example/

How ANN learn?

- We need to calculate a Loss/Cost over the whole training set, which is the objective the neural network tries to optimize.
- Neural networks learn through a process called "**backpropagation**," which essentially applies the **chain rule** from calculus used to compute **gradients** for all weights in the network.
- The computed gradients are used to update the weights and biases. The basic form of this step is to subtract the gradient multiplied by a learning rate from each weight and bias.



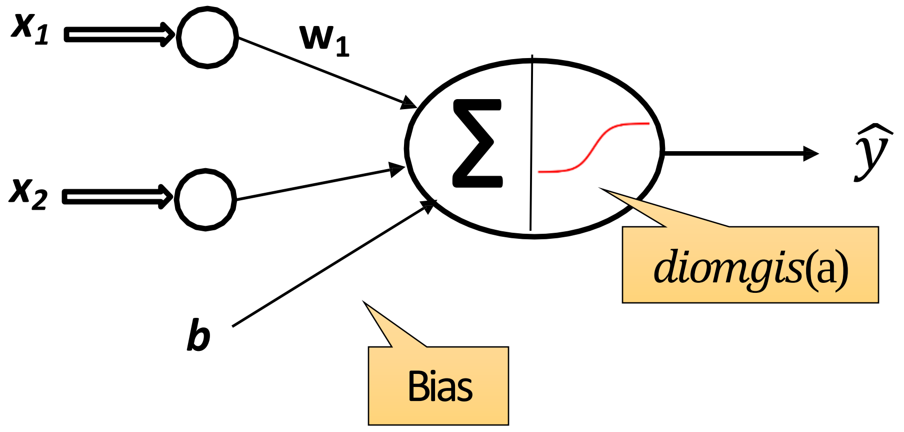
Source: https://towardsdatascience.com/creating-neural-networks-from-scratch-in-python-6f02b5dd911

Backpropagation

The Back-propagation (**BP**) Algorithm

- The Backpropagation (**BP**) algorithm for neural networks involves a two-phase cycle of forward and backward passes.
- In the **forward pass**, input data is passed through the network to compute the output.
- During the **backward pass**, the error between the predicted and actual outputs is calculated as $\mathcal{L}(\hat{y}, y)$, and this error is propagated backword through the network to update the weights and biases.
- Goal – Minimize the error by adjusting the weights and biases based on the gradient of the error with respect to these parameters.

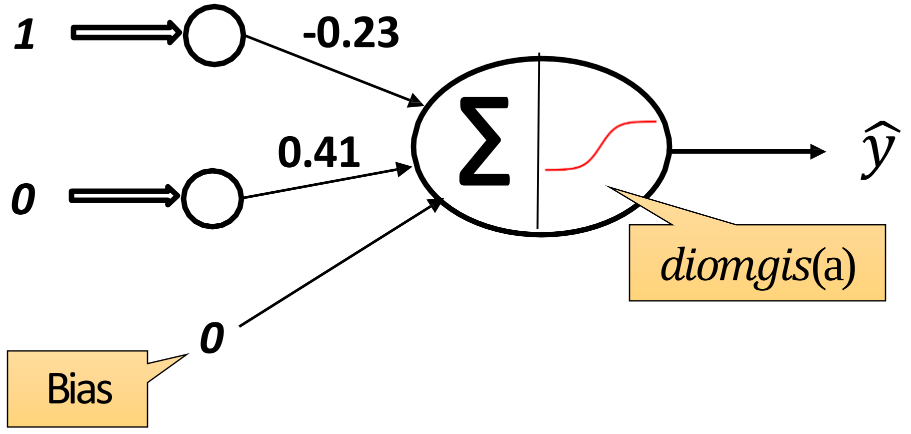
Single-Layer NN



$$a = \underline{\mathbf{w}^T \boldsymbol{x} + b} = w_1 x_1 + w_2 x_2 + b$$

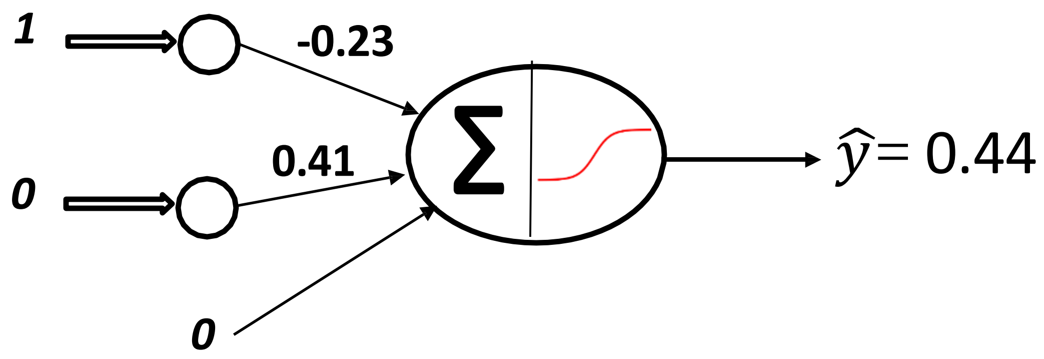
$$\hat{y} = \phi(a) = sigmoid(a) = \frac{1}{1+e^{-a}}$$

Initialization



Inputs	Targets
[0, 0]	0
[0, 1]	1
[1, 0]	1
[1, 1]	1

Forward Pass



- Calculations:

  $a = w_1x_1 + \ w_2x_2 + b$

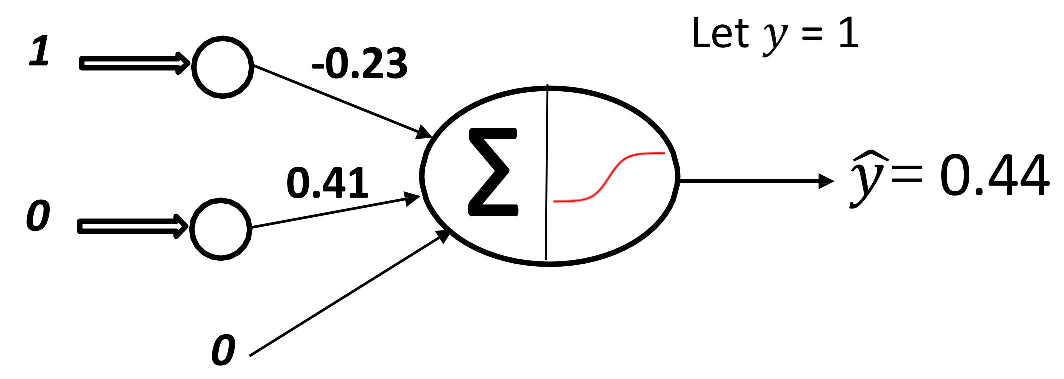
  $a = (1 \times -0.23) + (0 \times 0.41) + 0$

  $= -0.23$
- The output of the neuron using the sigmoid activation function is:

  $\hat{y} = sigmoid(\text{-}0.23) \approx 0.44$

P.S. Sigmoid Calculator: https://www.tinkershop.net/ml/sigmoid_calculator.html

Compute The Loss



- Now, the loss using the Mean Squared Error formula is

$$\mathcal{L}(y-\hat{y}) = \frac{1}{2}(y-\hat{y})^2 = 0.5(0.66)^2 = 0.2178$$

- Since we want to minimize the error, we perform gradient descent.
- To compute the derivative of the loss with respect to $\hat{y}$:

$$\frac{d\mathcal{L}}{d\hat{y}} = (y-\hat{y}) \cdot (-1) = (1-0.44) \cdot (-1) = -0.66$$

- if $\frac{d\mathcal{L}}{d\hat{y}}$ has a negative value with respect to $\hat{y}$, it generally indicates that increasing ($\hat{y}$) would lead to a decrease in the loss ($\mathcal{L}$).

The Gradients of The Loss

- The gradient of the loss with respect to w_1 is:

$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial a} \cdot \frac{\partial a}{\partial w_1}$$

- The gradient of the loss with respect to w_2 is:

$$\frac{\partial \mathcal{L}}{\partial w_2} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial a} \cdot \frac{\partial a}{\partial w_2}$$

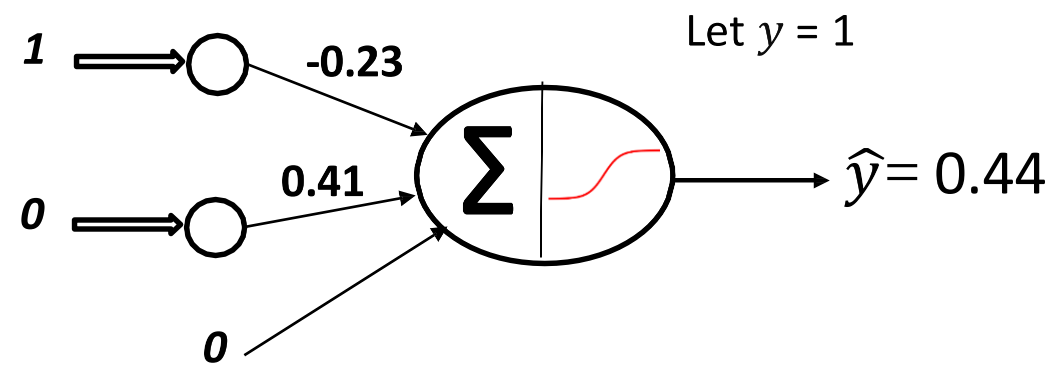
- Components:

  $\frac{\partial \mathcal{L}}{\partial \hat{y}}$ is the derivative of the Loss function with respect to the output. This represents how much the loss function will change with a small change in the output prediction.

  $\frac{\partial \hat{y}}{\partial a}$ is the derivative of the output $\hat{y}$ with respect to its total input. For a sigmoid activation function, this derivative is $\acute{\sigma}(z) = \sigma(z) \cdot (1 - \sigma(z))$

  $\frac{\partial a}{\partial w_1}$ is the derivative of the total input a with respect to the weight w_1, which is simply the input x_1 that is associated with that weight.

Compute The Gradients



- Now we calculate the gradients with respect to $w1$ and $w2$:

$$\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial a} \cdot \frac{\partial a}{\partial w_1} \quad \text{and} \quad \frac{\partial \mathcal{L}}{\partial w_2} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial a} \cdot \frac{\partial a}{\partial w_2}$$

- $\frac{\partial \mathcal{L}}{\partial w_1} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial a} \cdot \frac{\partial a}{\partial w_1} = (-0.66) \cdot \underbrace{\hat{y} \cdot (1 - \hat{y})}_{\left(\frac{\partial \hat{y}}{\partial a}\right) \to \text{sigmoid activation function}} \cdot x_1 = -0.66 \times 0.44 \times (1-0.44) \times 1 \approx -0.192$

- $\frac{\partial \mathcal{L}}{\partial w_2} = (-0.66) \times 0.44 \times 0.66 \times 0 = 0$

Backward Pass

- Once you have the gradients of the loss with respect to the weights, you can update the weights.
- Here's the mathematical expression for weight updating:

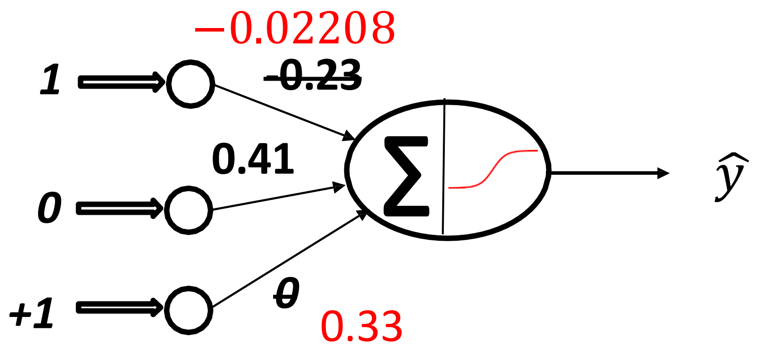
$$w_{new} = w_{old} - (\eta \times \frac{\partial \mathcal{L}}{\partial w})$$

$$\therefore w_{1new} = -0.23 - (0.5 \times -0.192)$$

$$w_{1new} = -0.02208$$

w_{2new} = 0.41 – 0.5 × 0 = 0.41

Let y = 1 & η = 0.5



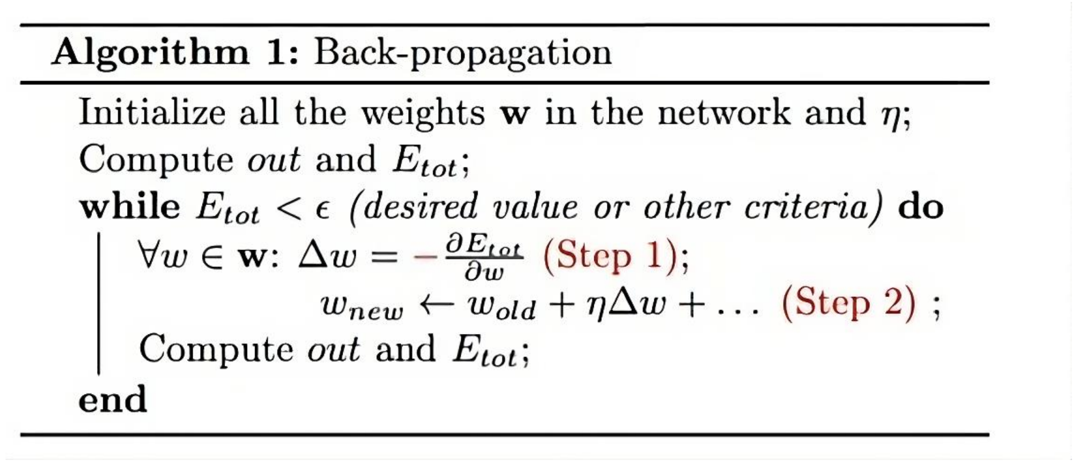
- To update the bias, we need to use the following: $b_{new} = b_{old} - (\eta \times \frac{\partial \mathcal{L}}{\partial b})$,

  for a simple neural network with a single output neuron using mean squared error as the loss function, the derivative of the loss function with respect to the bias *b* is calculated as: $\frac{\partial \mathcal{L}}{\partial b} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial b} = \frac{\partial \mathcal{L}}{\partial \hat{y}} \cdot 1 = \frac{\partial \mathcal{L}}{\partial \hat{y}}$

$\therefore b_{new} = bold - (\eta \times \frac{\partial \mathcal{L}}{\partial b})$ = 0 – (0.5 × $-$ 0.66) = 0.33

Backpropagation Algorithm

- **Initialization**: All the weights w in the network, and the learning rate η are initialized.
- **Computation**: The network computes the output and the total error E_{tot}.
- **Loop**: The algorithm enters a loop that continues until E_{tot} is less than a threshold ϵ, representing the desired error level or another stopping criterion.
    - a. For each weight w, the change in weight Δw is calculated as the negative gradient of the total error with respect to the weight. This is the first step in gradient descent optimization, aiming to reduce the total error.
    - b. The weight is then updated to a new value w_{new} by adding the product of the learning rate η and Δw to the current weight. This is the second step where the actual update takes place.
- **Recalculation**: The output and total error are recomputed, which will inform the next iteration's adjustments.



```
Algorithm 1: Back-propagation
Initialize all the weights $\mathbf{w}$ in the network and $\eta$;
Compute $out$ and $E_{tot}$;
while $E_{tot} < \epsilon$ (desired value or other criteria) do
    $\forall w \in \mathbf{w}$: $\Delta w = -\frac{\partial E_{tot}}{\partial w}$ (Step 1);
        $w_{new} \leftarrow w_{old} + \eta \Delta w + \dots$ (Step 2) ;
    Compute $out$ and $E_{tot}$;
end
```

Backpropagation pseudocode by A. Micheli, Computer Science Department @ University of Pisa

- This iterative process helps minimize the error by adjusting the weights, effectively 'training' the neural network.
- An **epoch** refers to one complete cycle through the **entire training dataset**.

Next

- Continue ANN!

Thank you