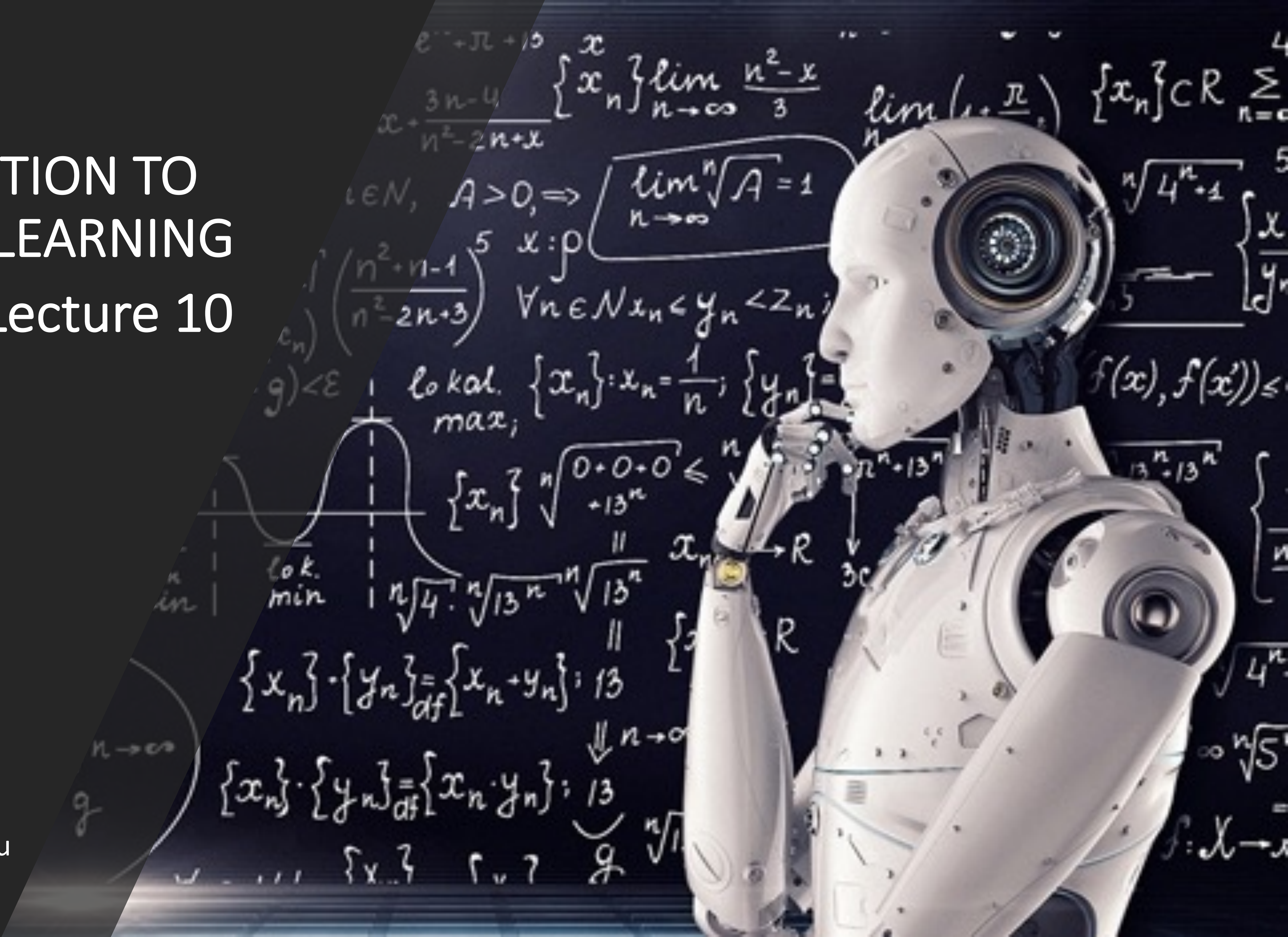
UMBC

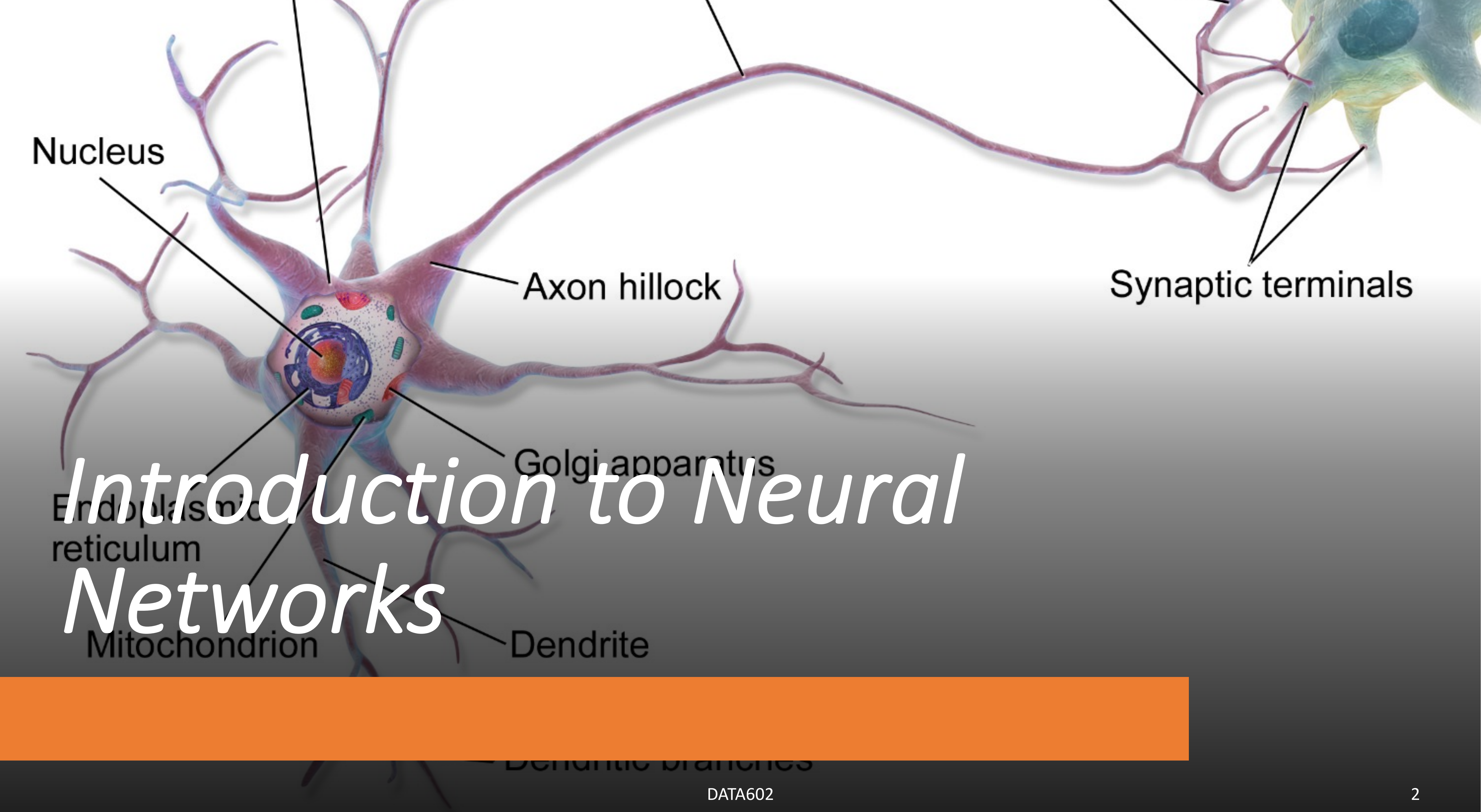
INTRODUCTION TO MACHINE LEARNING

DATA 602 Lecture 10



Dr. Tony Diana

tonydian@umbc.edu



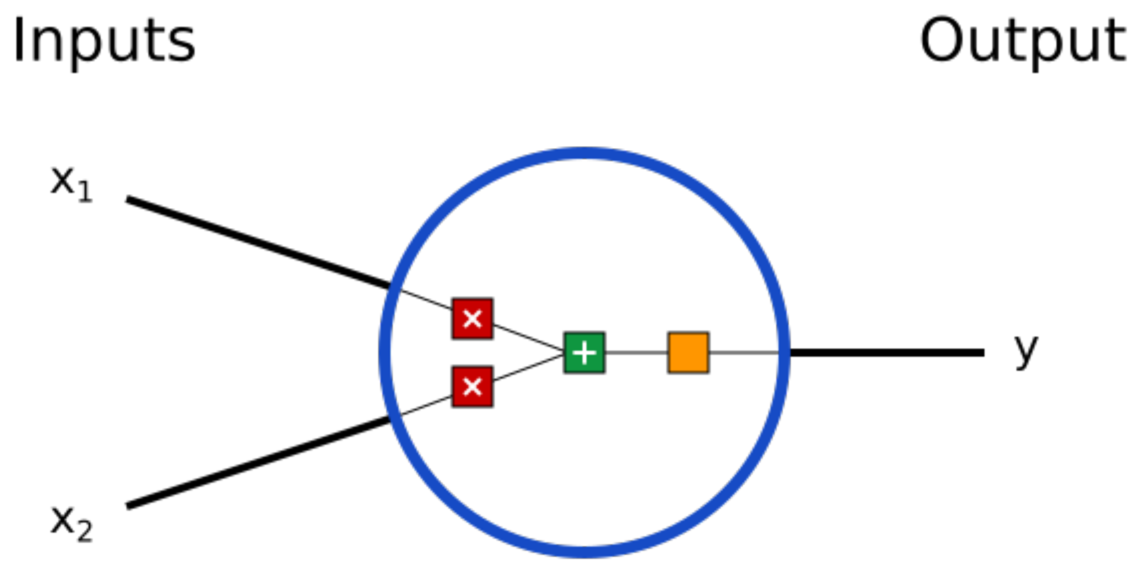
Introduction to Neural Networks

Introduction

- **Artificial Neural Networks** (ANN) are primarily used in supervised learning problems
  - Say we have a set of input information like images
  - We are training an algorithm to map the information to a desired output, such as a class or category
- When **neural networks or NN** are used in a **supervised learning context**, the images are fed to the network with a representation of the corresponding category labels being the desired output of the network
- **ANN** derives its name from the **biological neural networks** commonly found in the brain
- The **neuron** is the building block on which all neural networks are constructed, connecting a number of neurons in different configurations to form more powerful structures
- Each neuron in an ANN is composed of four individual parts:
  - An **input** value
  - A tunable **weight** (theta or Θ)
  - An **activation function** that operates on the input value, and
  - The resulting **output** value
- The **activation function** is specifically chosen depending upon the objective of the neural network being designed
- The main functions include **hyperbolic tangent** or **tanh**, **linear**, **sigmoid**, and **ReLU** (**Re**ctified **L**inear **U**nit), among others

Building Block

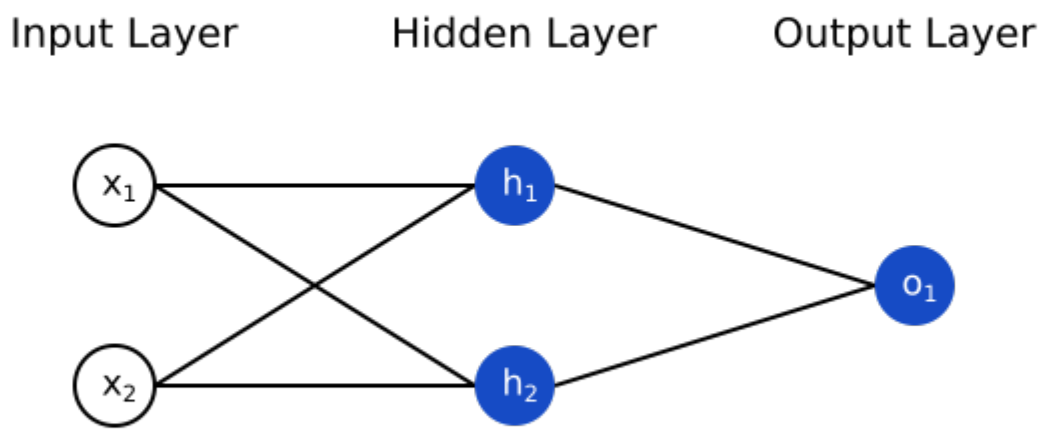
- **Neurons** are the basic units of a neural network
  - A neuron takes inputs, processes information, and produces one output



- First, each input is multiplied by a **weight**: x_1 --> $x_1 * w_1$ and x_2 --> $x_2 * w_2$
- Second, all the **weighted inputs** are added together with a **bias** b: $(x_1 * w_1) + (x_2 * w_2) + b$
- Third, the sum is passed through an **activation function**: $y = f(x_1 * w_1 + x_2 * w_2 + b)$
  - The activation function is used **to turn an unbounded input into an output with a predictable form**
  - A commonly used **activation function** is the **sigmoid function**

Building a Neural Network

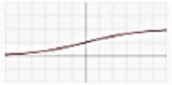
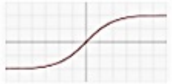
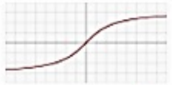
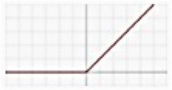
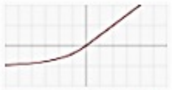
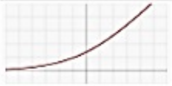
- A neural network is a group of connected neurons as in the example below:



- This network has two inputs: a **hidden layer** with 2 **neurons** (h_1 and h_2) and an **output layer** with 1 neuron (o_1)
  The **inputs** for o_1 are the **outputs** from h_1 and h_2, which makes it a **network**
- A **hidden layer** is any layer between the **input** (first) layer and **output** (last) layer
- There can be **multiple hidden layers**, hence the name **Deep Learning**

Activation Functions

- The sigmoid function only outputs numbers in the range (0,1)
- Generally, the activation function is used to compress outcomes from (−∞,+∞) to (0,1)

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

Activation Functions

- The **ReLU** activation function is often used in hidden layers
  - It is faster and does not create too many flat surfaces for Gradient Descent to stop at local minimum
- Sometimes, some of the neurons die during training when using ReLU, especially when using a high value for the learning rate. Therefore, the gradient can become zero
  - In this case, it is better to use **LeakyReLU.** The leak depends on the 'a' hyperparameter as 0.01 so that a(z)= max(az, z)
- In **PReLU**, 'a' is learned during training: it can overfit with smaller datasets
  - **PReLU**-Net is **a type of convolutional neural network** that utilizes parameterized **ReLU**s for its activation function
  - It also uses a robust initialization scheme, also known as Kaiming Initialization, to account for non-linear activation functions
  - **PReLU**-Net is **a type of convolutional neural network** that utilizes parameterized **ReLU**s for its activation function

Activation Functions

- For output layers, the best choice is **softmax** if we have mutually exclusive classes
- The problem with vanishing and exploding gradients can be attributed to the choice of the wrong activation function
- The **ELU** outperforms every **ReLU** variant in every experiment: training was faster and NN provides better test set performance
    - There are no bumps in this function, and it runs smoothly all the way, which helps gradient descent
    - If $z < 0$, then **ELU** will take negative values and the unit can have an output average near zero. This mitigates the problem of vanishing gradients
    - With large z negative values, the value approached by the **ELU** function is defined by the hyperparameter ‘a’, which is normally set to 1
    - Although rates of convergence can be fast, the **ELU** function computes slower than **ReLU** because it uses an exponential function

Feedforward

- Let's assume all neurons have the same **weights** w = [0,1], the same bias b = 0, and the same sigmoid activation function
- Let h_1, h_2, o_1 denote the **outputs** of the neurons they represent
- If we pass in the input x = [2, 3], then the output of the NN can be computed as follows:

$$
\begin{aligned}
h_1 = h_2 &= f(w \cdot x + b) \\
&= f((0 * 2) + (1 * 3) + 0) \\
&= f(3) \\
&= 0.9526
\end{aligned}
$$

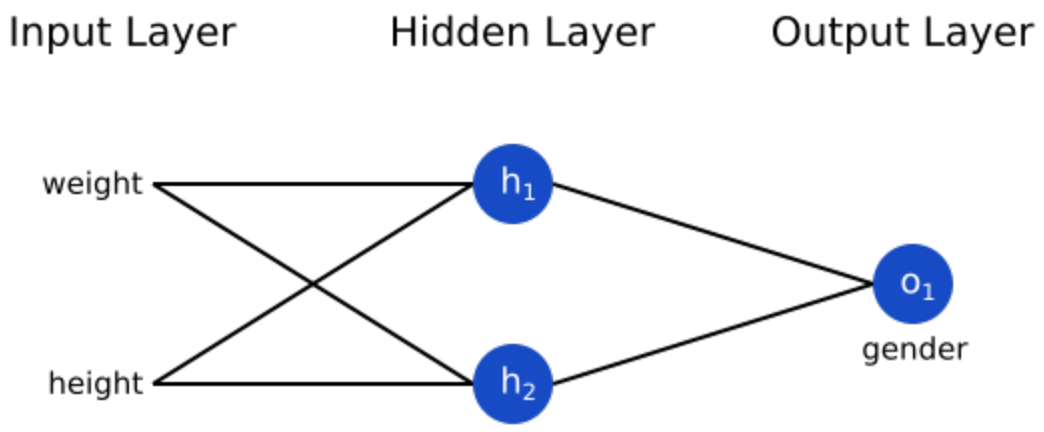
$$
\begin{aligned}
o_1 &= f(w \cdot [h_1, h_2] + b) \\
&= f((0 * h_1) + (1 * h_2) + 0) \\
&= f(0.9526) \\
&= \boxed{0.7216}
\end{aligned}
$$

The Loss Function

- Let's use the following information:

Name	Weight (lb)	Height (in)	Gender
Alice	133	65	F
Bob	160	72	M
Charlie	152	70	M
Diana	120	60	F

- Let's train the network to predict someone's gender given their weight and height:



The Loss Function (cont.)

- We represent Male as ‘0’ and Female as ‘1’
- We shift the data to make it easier to use, so that

Name	Weight (minus 135)	Height (minus 66)	Gender
Alice	-2	-1	1
Bob	25	6	0
Charlie	17	4	0
Diana	-15	-6	1

- Before we train the network, we first need to assess the **goodness of fit**
- The **loss** measures whether the network can do better
- We can use the **mean squared error** (MSE) loss as a measure of accuracy

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_{true} - y_{pred})^2$$

Better predictions = **Lower loss**

The Loss Function (cont.)

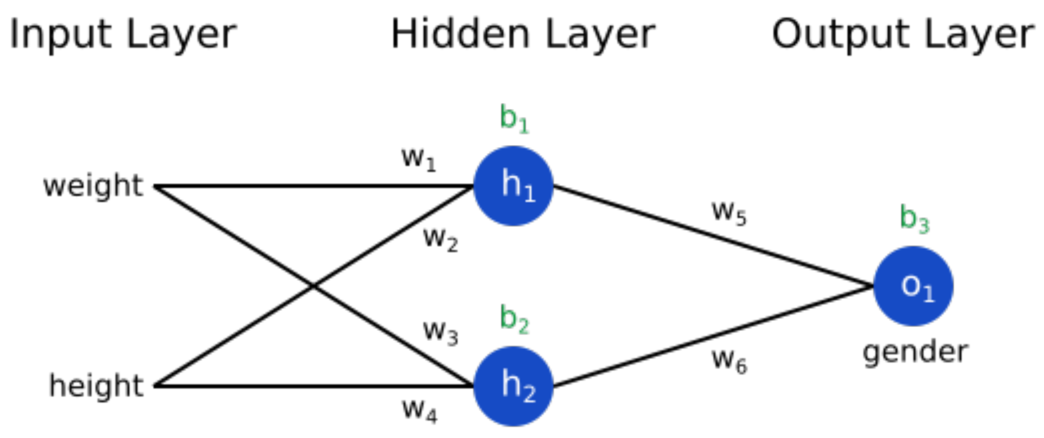
- Below is an example of how to compute the loss:

Name	y_{true}	y_{pred}	$(y_{true} - y_{pred})^2$
Alice	1	0	1
Bob	0	0	0
Charlie	0	0	0
Diana	1	0	1

$$\text{MSE} = \frac{1}{4}(1 + 0 + 0 + 1) = \boxed{0.5}$$

Training the Neural Network

- The goal of the NN is to **minimize the loss** of the neural network
- We know we can change the network's **weights** and **biases** to influence its **predictions**, but how do we do so in a way that decreases loss?
- Another way to think about **loss** is as a **function of weights and biases**
- Let's label each weight and bias in our network:



- Then, the loss is a multivariable function such that $L(w_1, w_2, w_3, w_4, w_5, w_6, b_1, b_2, b_3)$
- Imagine we wanted to tweak w_1. How would loss L change if we changed w_1?

Training the Neural Network (cont.)

- To start, let's rewrite the partial derivative in terms of ∂*y_pred*/∂w_1 instead (using the chain rule):

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial w_1}$$

- We can calculate ∂*L*/∂*y_pred* because we computed L= (1 − *y_pred*)² above:

$$y_{pred} = o_1 = f(w_5 h_1 + w_6 h_2 + b_3)$$

- Let h_1, h_2, o_1 be the outputs of the neurons they represent. Then

$$\frac{\partial L}{\partial y_{pred}} = \frac{\partial (1 - y_{pred})^2}{\partial y_{pred}} = \boxed{-2(1 - y_{pred})}$$

Training the Neural Network (cont.)

- Since w_1 only affects h_1 (not h_2), we can write:

$$\frac{\partial y_{pred}}{\partial w_1} = \frac{\partial y_{pred}}{\partial h_1} * \frac{\partial h_1}{\partial w_1}$$

$$\frac{\partial y_{pred}}{\partial h_1} = \boxed{w_5 * f'(w_5 h_1 + w_6 h_2 + b_3)}$$

- We do the same thing for $\partial h_1/\partial w_1$:

$$h_1 = f(w_1 x_1 + w_2 x_2 + b_1)$$

$$\frac{\partial h_1}{\partial w_1} = \boxed{x_1 * f'(w_1 x_1 + w_2 x_2 + b_1)}$$

Training the Neural Network (cont.)

- x_1 is the weight and x_2 is height. $f'(x)$ represents the derivate of the sigmoid function. Let's derive it:

$$f(x) = \frac{1}{1+e^{-x}}$$

$$f'(x) = \frac{e^{-x}}{(1+e^{-x})^2} = f(x) * (1 - f(x))$$

- We break down $\partial L/\partial w_1$ into several parts so we can calculate:

$$\boxed{\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial h_1} * \frac{\partial h_1}{\partial w_1}}$$

- This method of calculating partial derivatives by working *backwards* is known as **backpropagation**, or "backprop"

Calculating the Partial Derivative

- Let's take the example of Alice:

Name	Weight (minus 135)	Height (minus 66)	Gender
Alice	-2	-1	1

- We initialize all the weights to 1 and all the biases to 0
- If we do a feedforward pass through the network, we get:

$$
\begin{aligned}
h_1 &= f(w_1 x_1 + w_2 x_2 + b_1) \\
&= f(-2 + -1 + 0) \\
&= 0.0474
\end{aligned}
$$

$$h_2 = f(w_3 x_1 + w_4 x_2 + b_2) = 0.0474$$

$$
\begin{aligned}
o_1 &= f(w_5 h_1 + w_6 h_2 + b_3) \\
&= f(0.0474 + 0.0474 + 0) \\
&= 0.524
\end{aligned}
$$

Calculating the Partial Derivative

- The network outputs *y_pred* = 0.524, which does not strongly favor Male (0) or Female (1). Let's calculate $\partial L/\partial w_1$:

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial y_{pred}} * \frac{\partial y_{pred}}{\partial h_1} * \frac{\partial h_1}{\partial w_1}$$

$$\begin{aligned}\frac{\partial L}{\partial y_{pred}} &= -2(1 - y_{pred}) \\ &= -2(1 - 0.524) \\ &= -0.952\end{aligned}$$

$$\begin{aligned}\frac{\partial y_{pred}}{\partial h_1} &= w_5 * f'(w_5 h_1 + w_6 h_2 + b_3) \\ &= 1 * f'(0.0474 + 0.0474 + 0) \\ &= f(0.0948) * (1 - f(0.0948)) \\ &= 0.249\end{aligned}$$

$$\begin{aligned}\frac{\partial h_1}{\partial w_1} &= x_1 * f'(w_1 x_1 + w_2 x_2 + b_1) \\ &= -2 * f'(-2 + -1 + 0) \\ &= -2 * f(-3) * (1 - f(-3)) \\ &= -0.0904\end{aligned}$$

$$\begin{aligned}\frac{\partial L}{\partial w_1} &= -0.952 * 0.249 * -0.0904 \\ &= \boxed{0.0214}\end{aligned}$$

Gradient Descent

- It is an optimization algorithm used in training a model
- A gradient is a **vector-valued function** that represents the **slope of the tangent** of the graph of the function, pointing to the direction of the greatest rate of increase of the function
- It is a **derivative** that indicates the **incline or the slope of the cost function**
- The **Gradient Descent** finds the parameters that **minimize the cost function** (error in prediction)
- The **Gradient Descent** does this by moving through iterations toward a set of parameter values that minimize the function, taking steps in the opposite direction of the gradient
- Now, we have all the tools we need to train a neural network
- We use an optimization algorithm called **Stochastic Gradient Descent** (SGD) that tells us how to change our weights and biases to minimize loss. Here is the equation: $$w_1 \leftarrow w_1 - \eta \frac{\partial L}{\partial w_1}$$
- η is a constant called the **learning rate** that controls how fast we train. We are subtracting $\eta\ \partial w_1/\partial L$ from w_1:
  - If $\partial L/\partial w_1$ is positive, w_1 will decrease, which makes L decrease
  - If $\partial L/\partial w_1$ is negative, w_1 will increase, which makes L decrease
- If we do this for every weight and bias in the network, the loss will slowly decrease, and our network will improve
- **Convergence** is a situation when the loss function does not improve significantly, and we have found a point near to the minima

Types of Gradient Descent

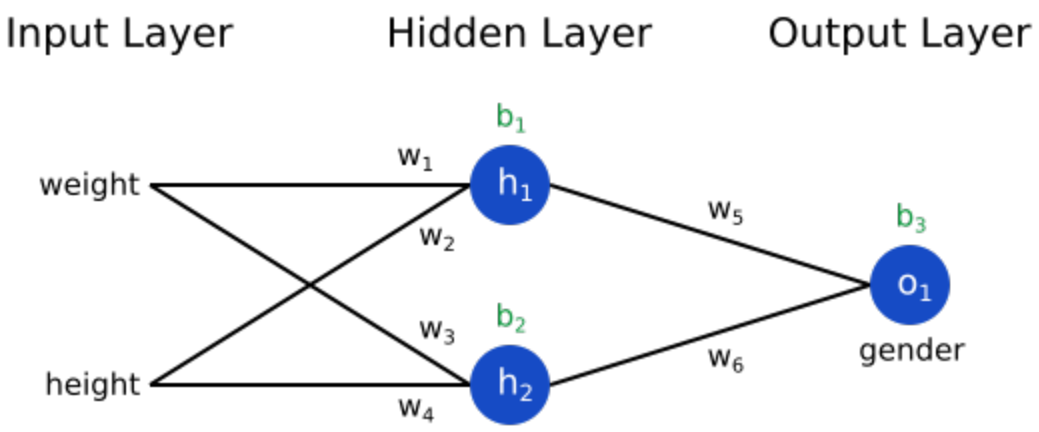
- **Batch Gradient Descent**, aka ‘Vanilla Gradient Descent’, calculates the error for each observation in the dataset but performs an update only after all observations have been evaluated
- **Stochastic Gradient Descent** (SGD) performs a parameter update for each observation. So instead of looping over each observation, *it just needs one* to perform the parameter update. SGD is usually faster than batch gradient descent, but its frequent updates cause a higher variance in the error rate, that can sometimes jump around instead of decreasing
- **Mini-Batch Gradient Descent** is a combination of both gradient descent and stochastic gradient descent. Mini-batch gradient descent performs an update for a batch of observations. It is the algorithm of choice for neural networks, and the batch sizes are usually from 50 to 256

Training Process

- Choose **one** sample from our dataset. This is what makes it *stochastic* gradient descent — we only operate on one sample at a time
- Calculate all the partial derivatives of loss with respect to weights or biases (e.g., $\partial L/\partial w_1$, $\partial L/\partial w_2$, etc).
- Use the update equation to update each weight and bias.
- Go back to step 1

Recap

Name	Weight (minus 135)	Height (minus 66)	Gender
Alice	-2	-1	1
Bob	25	6	0
Charlie	17	4	0
Diana	-15	-6	1



The Perceptron Model

The Perceptron

- The **Perceptron** invented by Frank Rosenblatt in 1957 is the basic ANN architecture
- It is based on an artificial neuron called a **Linear Threshold Unit** (**LTU**)
- **LTU**s compute weighted sums of all inputs. A step function is then applied to the resulting sum and the result is output as h(w) step (z) = step($\mathbf{w}^T * \mathbf{x}$)
- The **activation function** applies a step rule (convert the numerical output into +1 or -1) to check if the output of the weighting function is greater than zero or not
- The Perceptron uses the **Heaviside** step function and is characterized as follows:

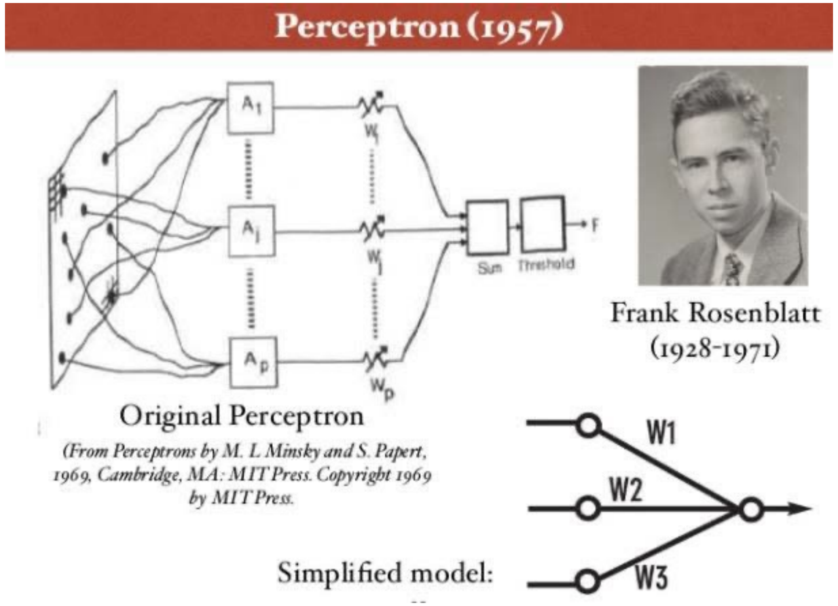
Heaviside (z) = 0 if $z < 0$ or 1 if $z \geq 0$

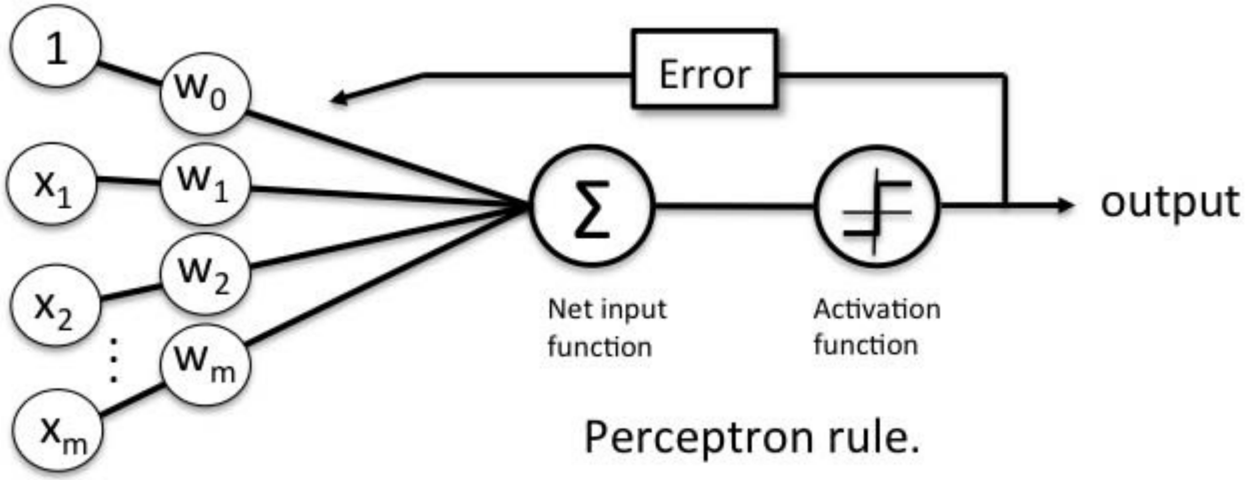
The sign of (z) is -1 if $z < 0$, 0 if $z = 0$, or +1 is $z > 0$

- The **step function** gets triggered above a certain value of the neuron output; else it outputs zero
- The **sign function** outputs +1 or -1 depending on whether neuron output is greater than zero or not
- The **sigmoid** is the S-curve and outputs a value between 0 and 1
- A **Perceptron** that has a pair of inputs and three outputs can simultaneously classify in no less than three binary classes, which makes it a **multi-output classifier**

The Perceptron

- There are two types of **Perceptrons**: **Single layer** and **Multilayer**
- Single layer Perceptrons can learn only linearly separable patterns
- Multilayer Perceptrons or feedforward neural networks with two or more layers have the greater processing power
- The **Perceptron** algorithm learns the weights for the input signals in order to draw a linear decision boundary
    - This enables the identification of the two linearly separable classes +1 and -1
- The **Perceptron** receives multiple input signals, and if the sum of the input signals exceeds a certain threshold, it either outputs a signal or does not return an output. In the context of **supervised learning** and **classification**, this can be used to predict the class of a sample





Perceptron rule.

Summary

- Introduced **neurons**, the building blocks of neural networks
- Used the **sigmoid activation function** in our neurons
- Saw that neural networks are just connected neurons
- Created a dataset with weight and height as inputs (or **features**) and ‘gender’ as the output (or **label**)
- Learned about **loss functions** and the **mean squared error** (MSE) loss
- Realized that training a network is just minimizing its loss
- Used **backpropagation** to calculate partial derivatives
- Used **Stochastic Gradient Descent** (SGD) to train our network
- We finished the lecture with the Perceptron model