
# Walchand College of Engineering, Sangli

## Department of

## Computer Science and Engineering

# TY CSE PE-2: Deep Learning 4CS335

Prof. Kiran P. Kamble

# Outline

- Course structure
- Desirable requirements
- Textbooks
- References
- Course Objectives
- Course Learning Outcomes
- CO-PO Mapping
- Teacher Assessment
- Course Contents
- Module wise Measurable Students Learning Outcomes

# Course structure

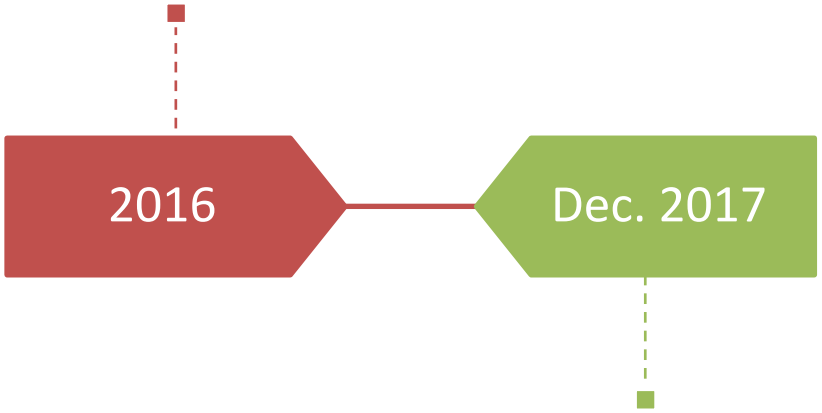
| L | T | P | Cr |
|---|---|---|---|
| 3 | - | - | 3 |

# Desirable requirements

Working knowledge of Linear Algebra, Statistics and Probability Theory

# Textbooks

1.Ian Goodfellow, Yoshua Bengio and Aaron Courville Deep Learning, MIT Press




2.Aurelien Geron, “ Hands-On Machine Learning with Scikit-Learn & TensorFlow”, O’REILLY

# References


- 1. Neural Networks: A Systematic Introduction, Raúl Rojas, 1996
- 2. Pattern Recognition and Machine Learning, Christopher Bishop, 2007
- 3. Prof. Mitesh M. Khapra, “Deep Learning”, course on NPTEL, July 2018
- 4. Andrew Ng, “Deep Learning Specialization”, Coursera online course

# Course Objectives

1. To explain the fundamentals of neural networks, recurrent neural networks (RNN), long short term memory cells and convolutional neural networks (CNN).
2. To demonstrate various learning models for practical application.
3. To discuss CNN, RNN and Generative model according to accuracy and speed evaluation parameter’s

# Course Learning Outcomes

| Course Outcomes (CO) with Bloom's Taxonomy Level | | |
|---|---|---|
| **CO1** | Illustrate fundamentals of deep learning using foundation of mathematics terminology | Understanding |
| **CO2** | Compare various deep learning models by hyper tuning various parameters | Analyzing |
| **CO3** | Demonstrate various case studies of deep learning. | Applying |
| **CO4** | Design and deploy deep learning models on various frameworks and platform. | Creating |

# CO-PO Mapping

- *1: Low, 2: Medium, 3: High*

| **PO** | **PO1** | **PO2** | **PO3** | **PO4** | **PO5** | **PO6** | **PO7** | **PO8** | **PO9** | **PO10** | **PO11** | **PO12** | **PSO1** | **PSO2** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **CO1** | 2 | | | | | | | | | | | | | |
| **CO2** | | 2 | | 1 | | | | | | | | | | |
| **CO3** | | | 2 | | | | | | | | | | | 1 |
| **CO4** | | | 3 | | 1 | | | | | | | | | 2 |

| Teaching Scheme | | Examination Scheme (Marks) | | | |
|---|---|---|---|---|---|
| **Lecture** | 2 Hrs/week | **T1** | **T2** | **ESE** | **Total** |
| **Tutorial** | - | 20 | 20 | 60 | 100 |
| **Practical** | - | | | | |
| **Interaction** | - | **Credits: 2** | | | |

# Teacher Assessment

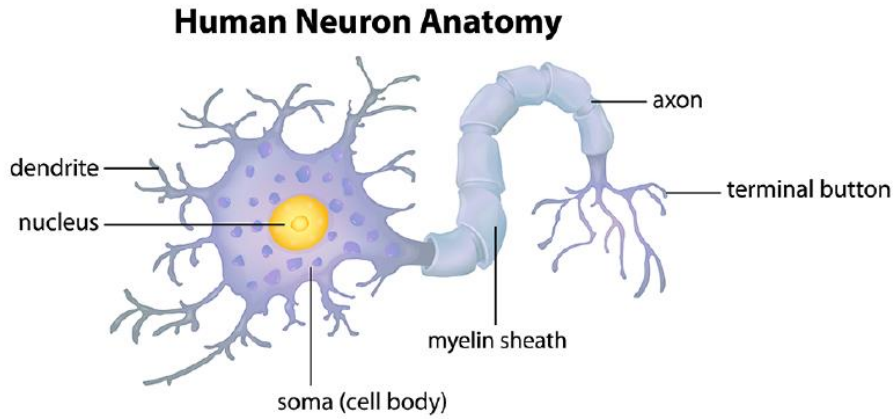
# Course Contents

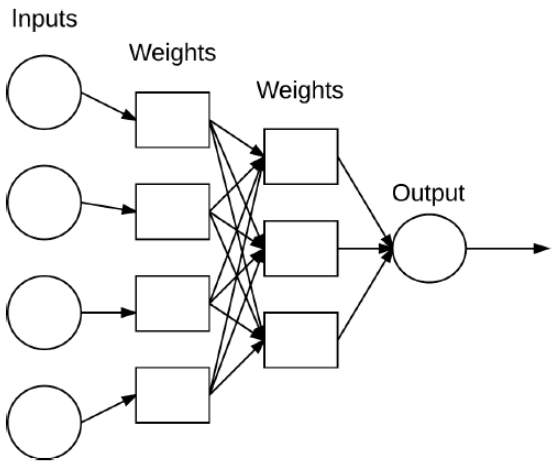
| | |
|---|---|
| **Module 1: Introduction to Deep Learning** | **7 Hrs.** |
| **Neural network fundamentals**: General Introduction to Deep Learning, Perceptron algorithm, Back propagation and Multi-layer Networks.<br>**Image fundamentals**: Pixels, Image coordinate, scaling and aspect ratios | |
| **Module 2: Parameterized Learning and Optimization Methods** | **6 Hrs.** |
| **Parameterized Learning:** Introduction to linear classification, Four components of parameterized learning, role of loss function.<br>**Optimization Methods**: Optimization Methods: Gradient descent, stochastic gradient descent (SGD) and extensions to SGD, regularization | |
| **Module 3: Convolutional Neural Networks (CNN)** | **7 Hrs.** |
| **Understanding Convolutions**: Convolutions versus Cross-correlation, The “Big Matrix” and “Tiny Matrix" Analogy, Kernels, A Hand Computation Example of Convolution The Role of Convolutions in Deep Learning.<br>**CNN Building blocks**: Layer Types, Convolutional Layers, Activation Layers , Pooling Layers , Fully-connected Layers , Batch Normalization , Dropout, ShallowNEt, LeNet, MiniVGGNET | |
| **Module 4: Deep learning based object detection** | **6 Hrs.** |
| Fundamentals of Object detection, Family of R-CNN, Single shot detectors (SSD), You only look once (YOLO) | |
| **Module 5: Sequence Models** | **7 Hrs.** |
| Recurrent Neural Networks, Vanishing gradients, Gated Recurrent Units (GRU), Long-short-term-memories (LSTMs) | |
| **Module 6: Generative Models** | **6 Hrs.** |
| Autoencoders, Variational Autoencoders, Generative Adversarial Networks | |

# Module 1: Introduction to Deep Learning

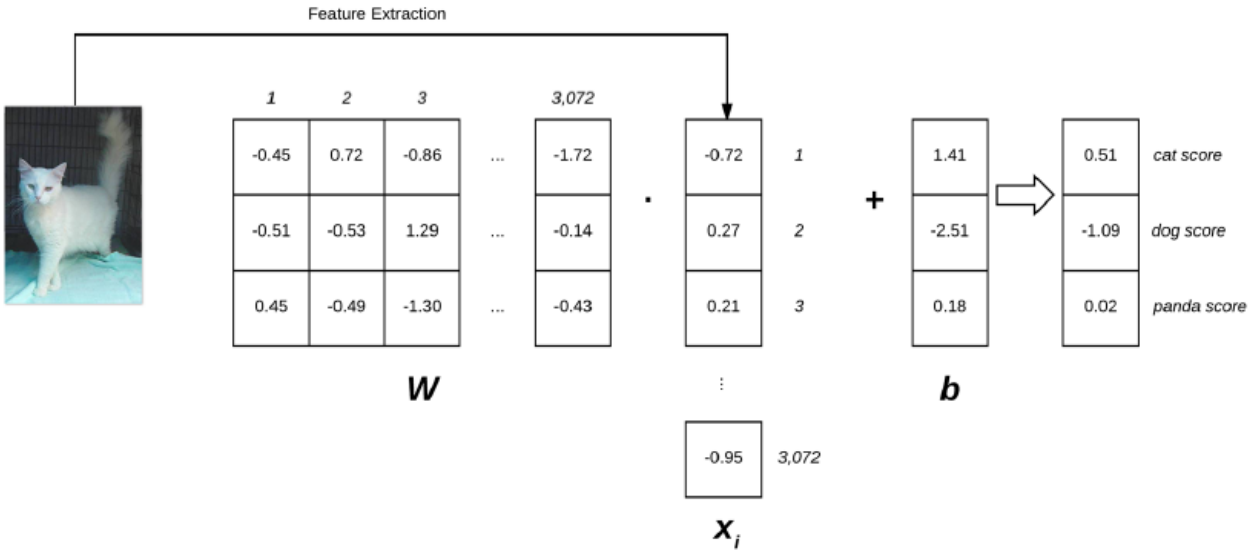
- To understand fundamentals of deep learning

Relation to Biology



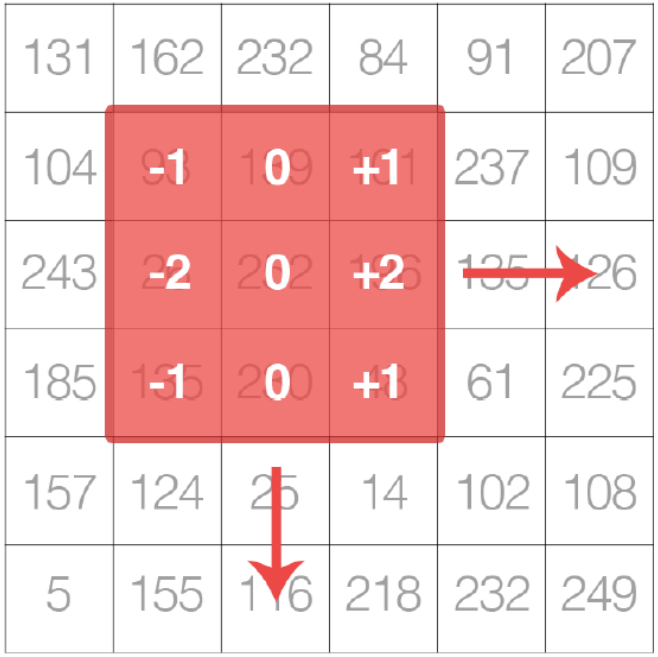



# Module 2: Parameterized Learning and Optimization Methods




Identify various methods of Fine tuning of optimization parameter

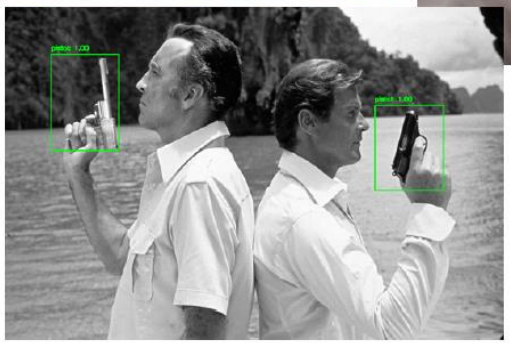
# Module 3: Convolutional Neural Networks (CNN)

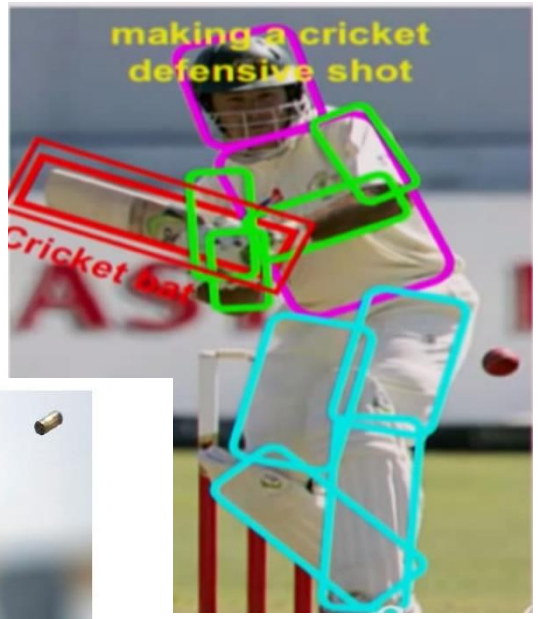



- Design CNN model for different application

# Module 4: Deep learning based object detection

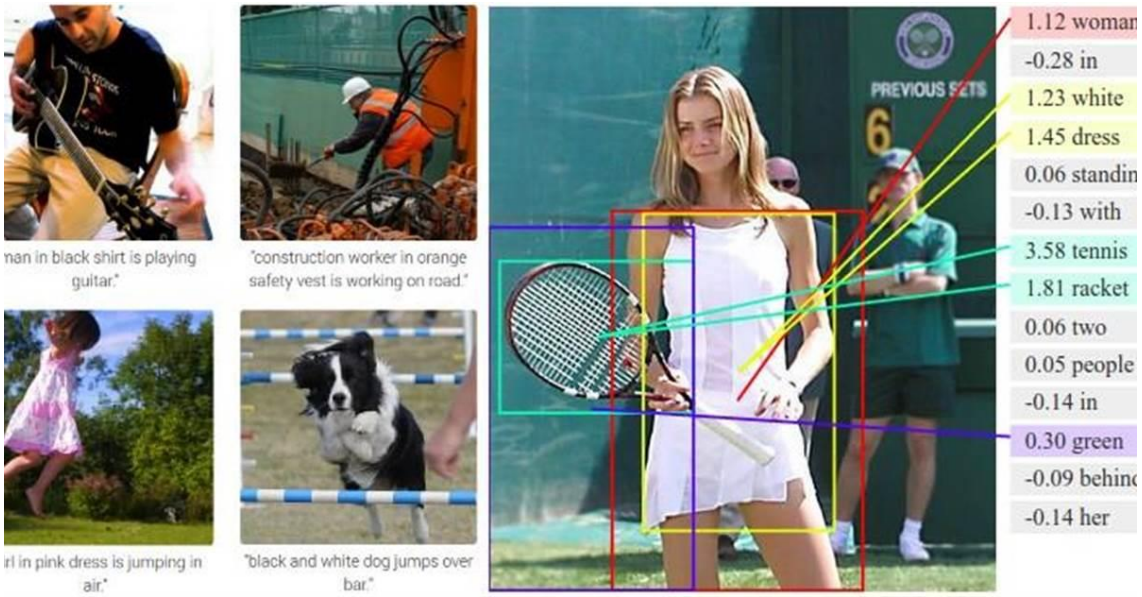





- Understand object detection using CNN model and to analyze performance.

# Module 5: Sequence Models




- Compare various sequence model architectures.

# Module 6: Generative Models

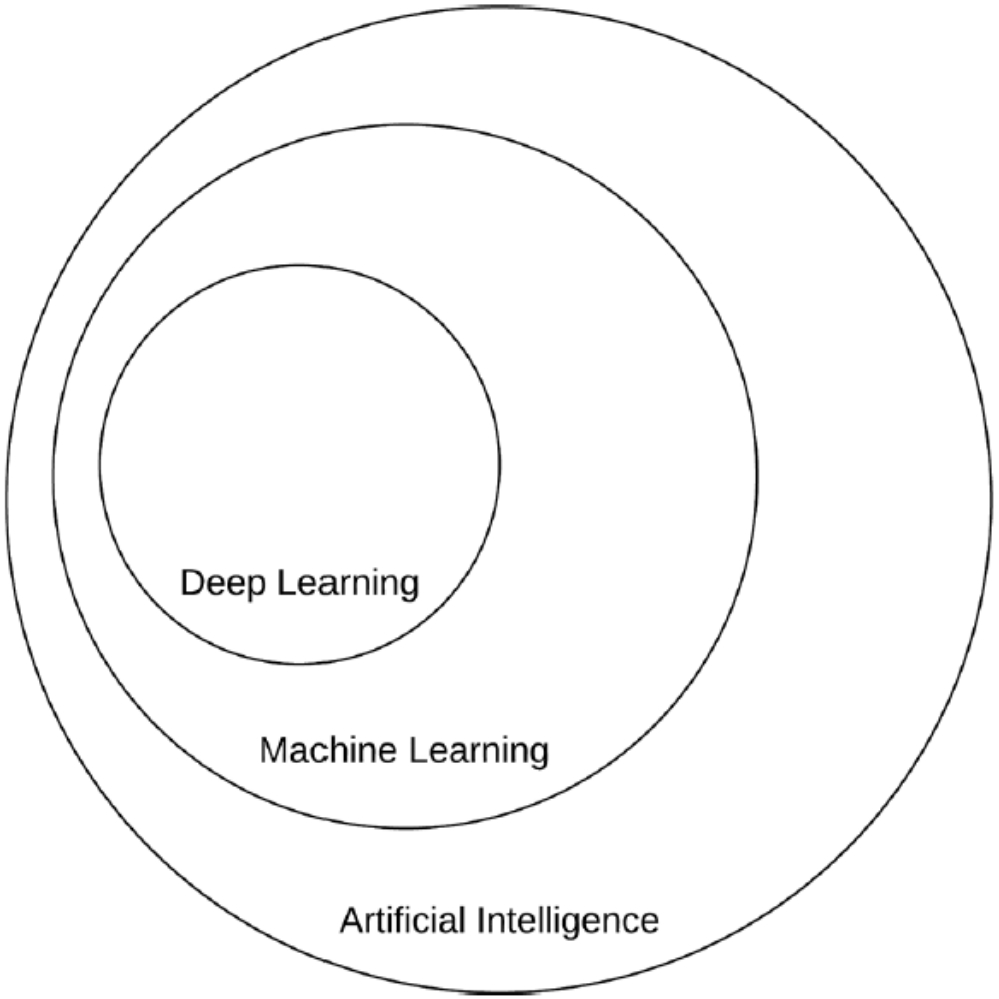
- Demonstrate generative model on a simple Dataset

# Module 1: Introduction to Deep Learning

# What Is Deep Learning?

*"Deep learning methods are representation-learning methods with multiple levels of representation, obtained by composing simple but nonlinear modules that each transform the representation at one level (starting with the raw input) into a representation at a higher, slightly more abstract level. [...] The key aspect of deep learning is that these layers are not designed by human engineers: they are learned from data using a general-purpose learning procedure"* – Yann LeCun, Yoshua Bengio, and Geoffrey Hinton, Nature 2015. [9]

# What Is Deep Learning?



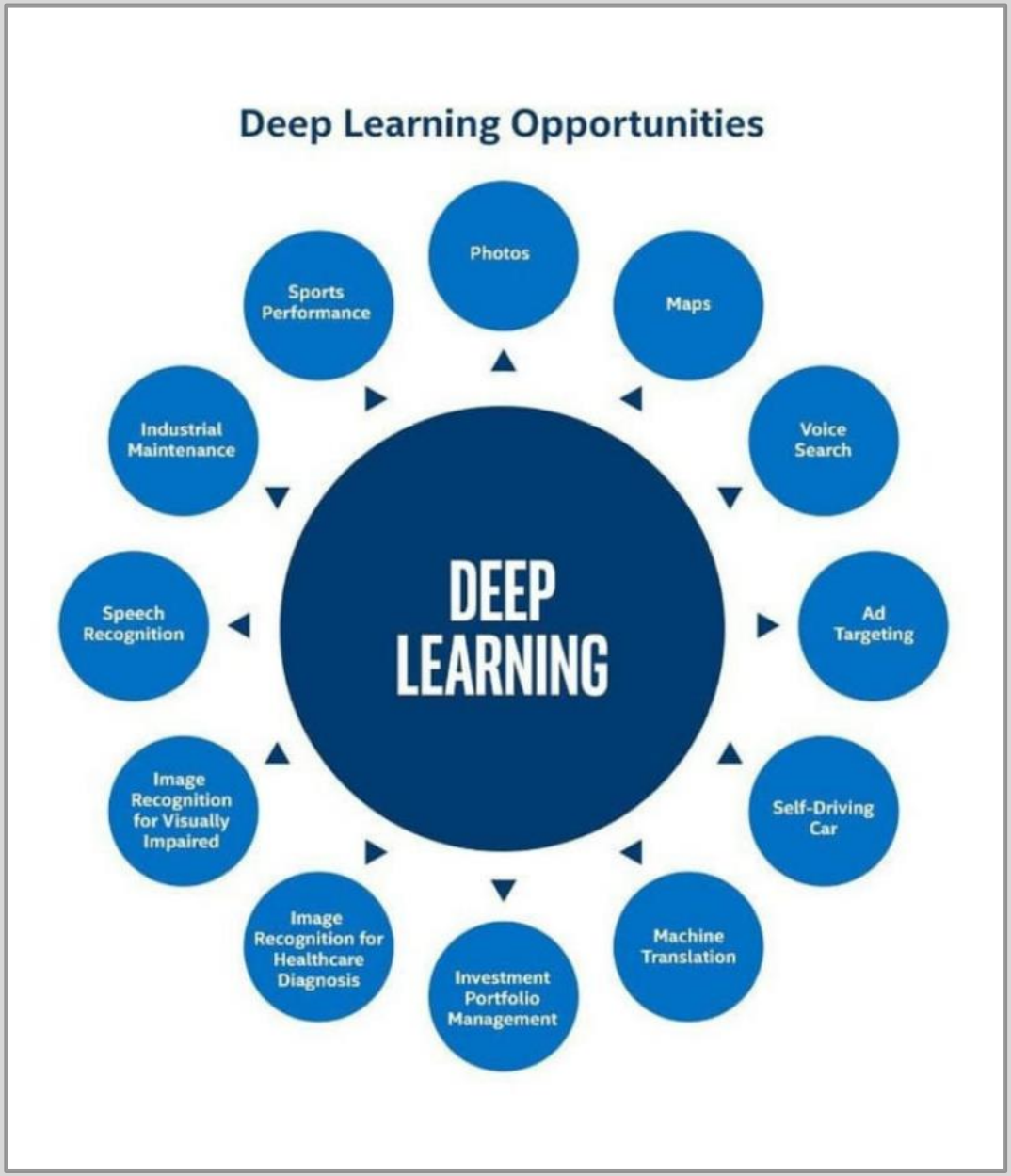

A Venn diagram describing deep learning as a subfield of machine learning which is in turn a subfield of artificial intelligence (Image inspired by Figure 1.4 of Goodfellow et al. [10]).

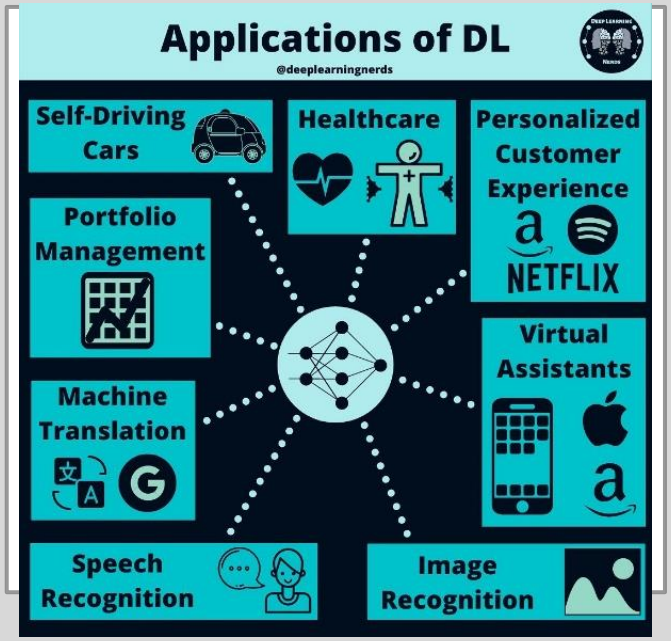
# How "Deep" Is Deep?

*“When you hear the term deep learning, just think of a large, deep neural net. Deep refers to the number of layers typically and so this kind of the popular term that’s been adopted in the press.”*

This is an excellent quote as it allows us to conceptualize deep learning as large neural networks where layers build on top of each other, gradually increasing in depth. The problem is we still don’t have a concrete answer to the question, “How many layers does a neural network need to be considered deep?”

*The short answer is there is no consensus amongst experts on the depth of a network to be considered deep*

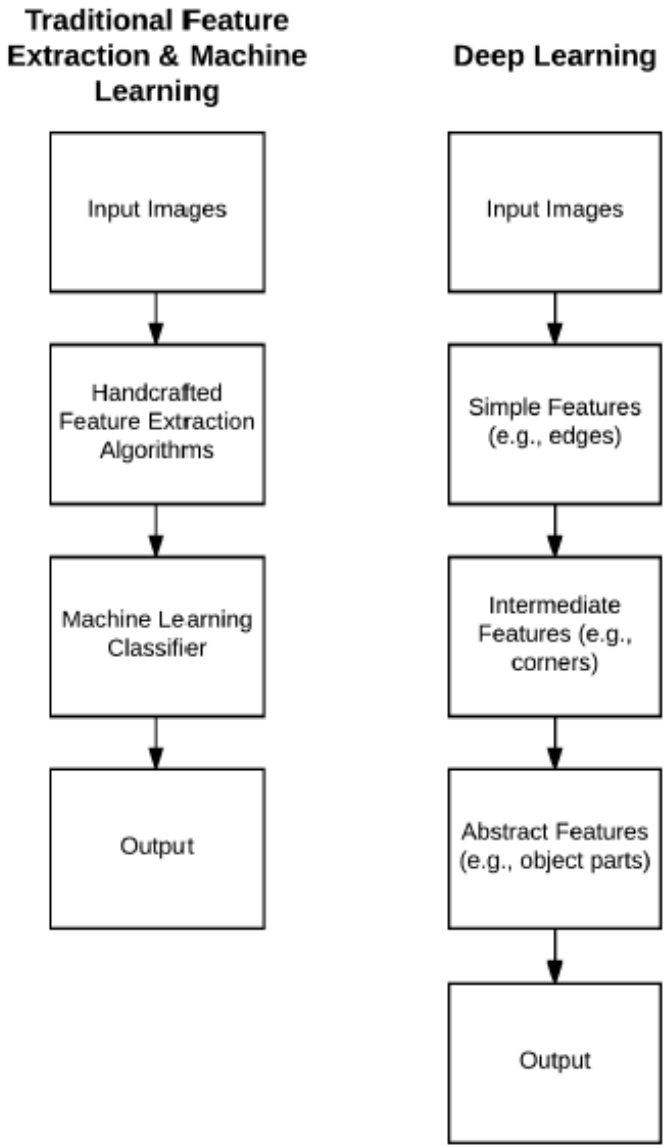

Deep Learning Opportunities
DEEP LEARNING
Photos
Maps
Voice Search
Ad Targeting
Self-Driving Car
Machine Translation
Investment Portfolio Management
Image Recognition for Healthcare Diagnosis
Image Recognition for Visually Impaired
Speech Recognition
Industrial Maintenance
Sports Performance


Applications of DL
@deeplearningnerds
Self-Driving Cars
Healthcare
Personalized Customer Experience
NETFLIX
Portfolio Management
Virtual Assistants
Machine Translation
Speech Recognition
Image Recognition

# Why Deep Learning

- **Left**: Traditional process of taking an input set of images, applying hand-designed feature extraction algorithms, followed by training a machine learning classifier on the features.

- **Right**: Deep learning approach of stacking layers on top of each other that automatically learn more complex, abstract, and discriminating features.



# Why Deep Learning

Previous research in ANNs that were heavily handicapped by:

1. Our lack of large, labeled datasets available for training
2. Our computers being too slow to train large neural networks
3. Inadequate activation functions Because of these problems, we could not easily train networks with more than two hidden layers during the 1980s and 1990s

# Why Deep Learning

In the current incarnation we can see that the tides have changed. We now have:

1. Faster computers

2. Highly optimized hardware (i.e., GPUs)

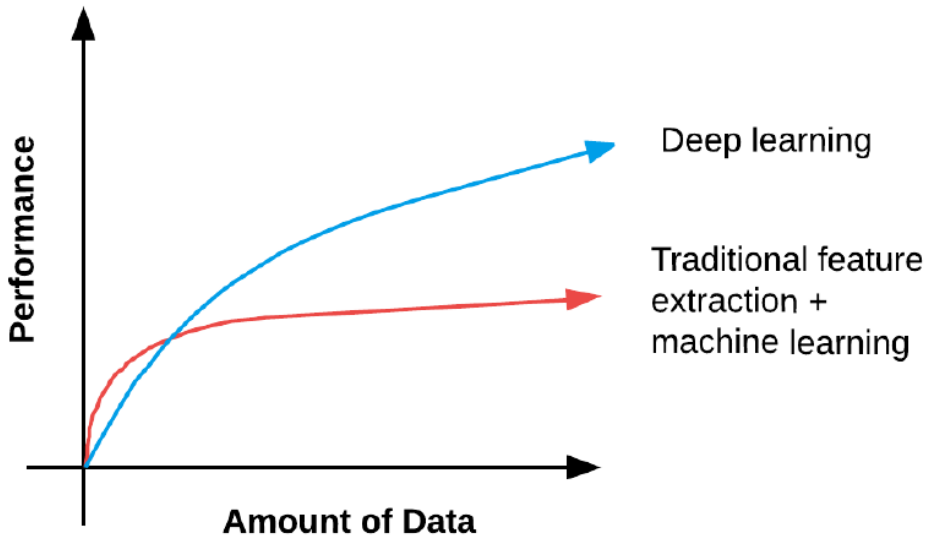
3. Large, labeled datasets in the order of millions of images

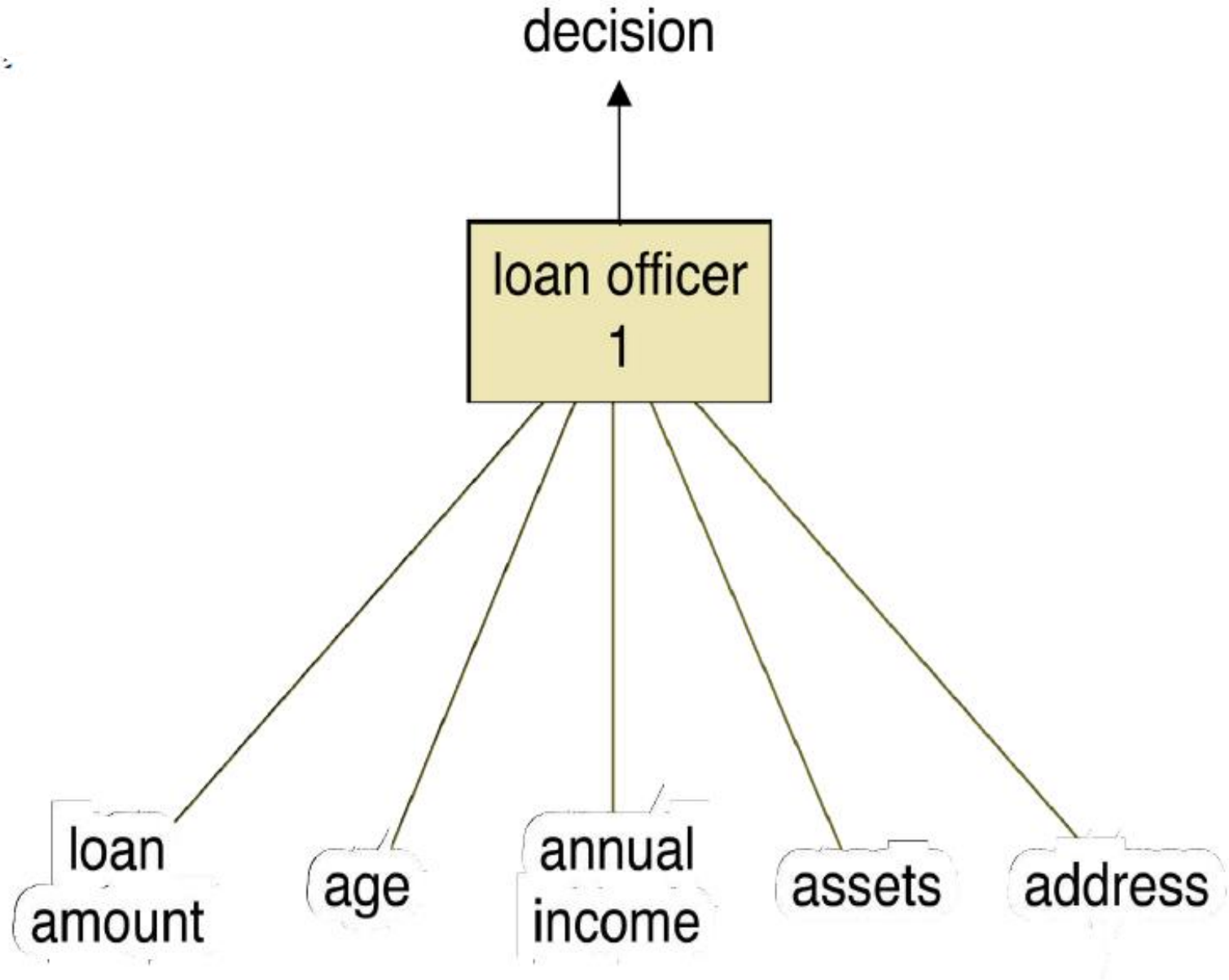
4. A better understanding of weight initialization functions and what does/does not work

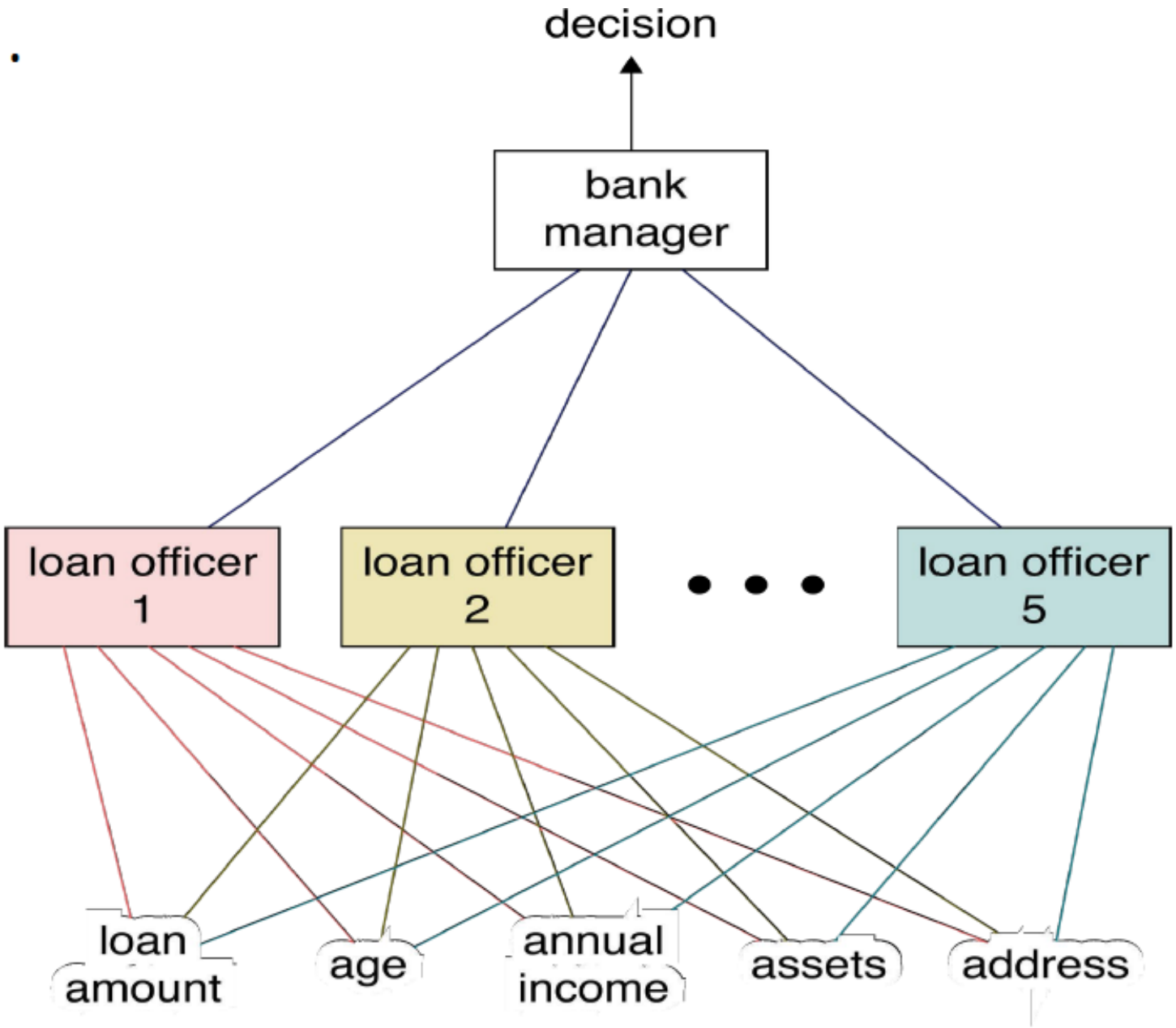
5. Superior activation functions and an understanding regarding why previous nonlinearity functions stagnated research

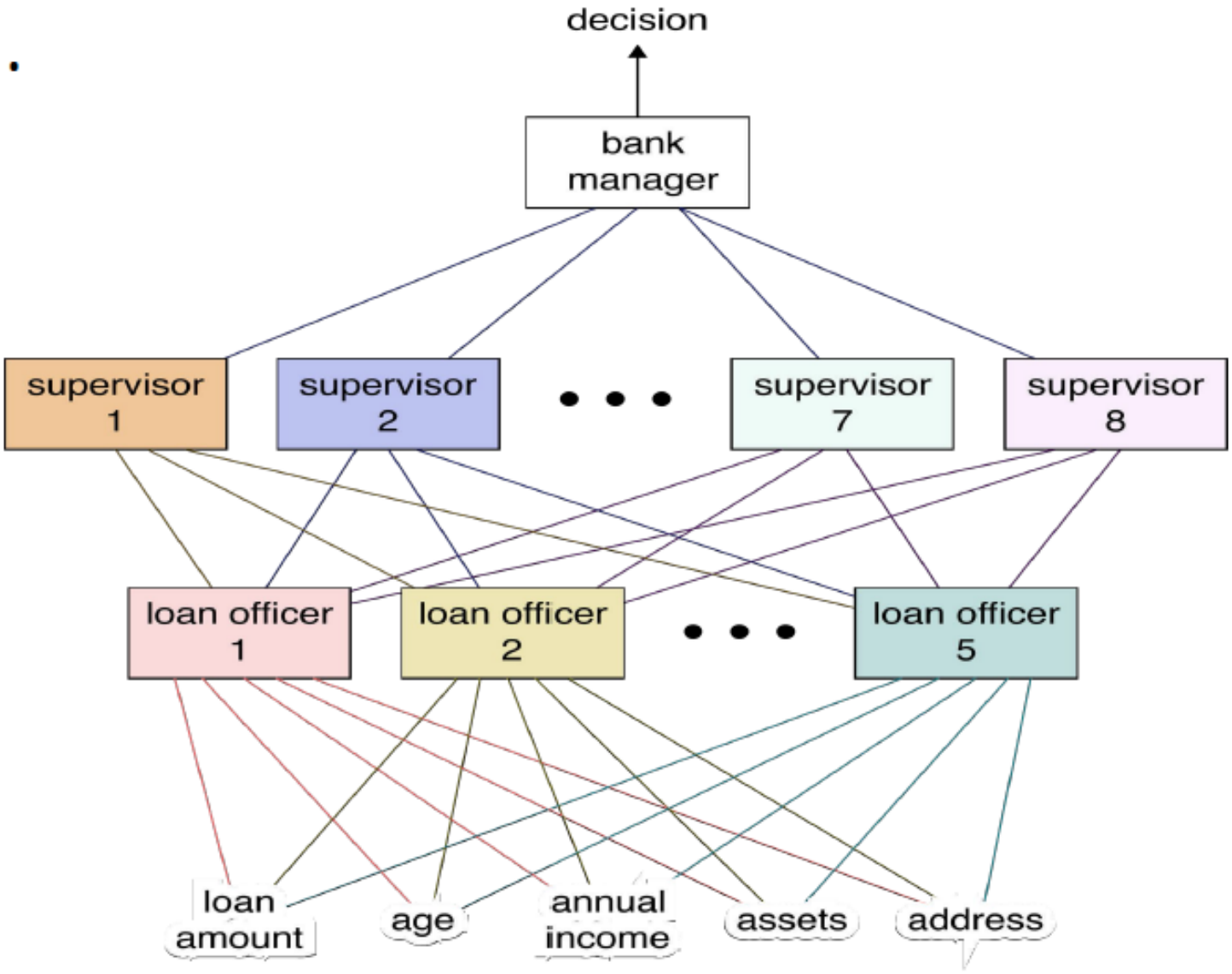
# Why Deep Learning

Figure : As the amount of data available to deep learning algorithms increases, accuracy does as well, substantially outperforming traditional feature extraction + machine learning approaches.

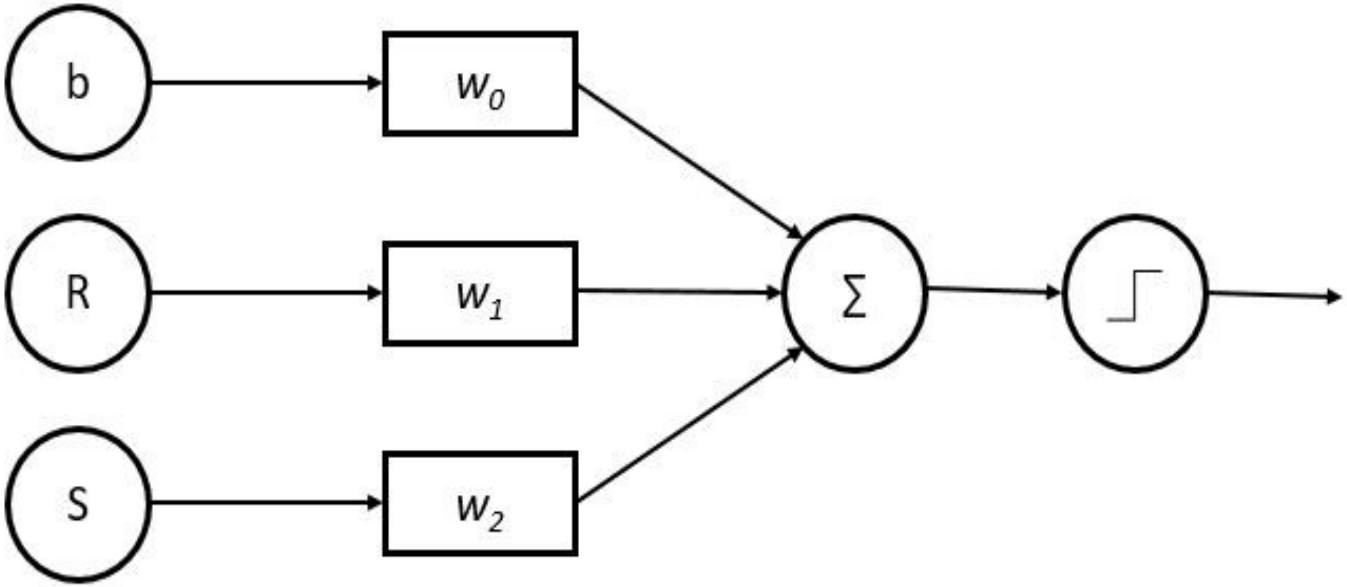



decision
loan officer 1
loan amount
age
annual income
assets
address


decision
bank manager
loan officer 1
loan officer 2
loan officer 5
loan amount
age
annual income
assets
address


decision
bank manager
supervisor 1
supervisor 2
supervisor 7
supervisor 8
loan officer 1
loan officer 2
loan officer 5
loan amount
age
annual income
assets
address

| Red in Colour (R) | Spherical in Shape (S) | Cricket Ball? (C) |
|---|---|---|
| No (0) | No (0) | No (0) |
| No (0) | Yes (1) | No (0) |
| Yes (1) | No (0) | No (0) |
| Yes (1) | Yes (1) | Yes (1) |




$$SUM = w_0 * b + w_1 * R + w_2 * S$$

$$w_0 = -6, w_1 = 5, w_2 = 5 \; and \; b = 1$$


$$SUM = (-6) * 1 + 0 * 5 + 0 * 5 = -6$$

$$w_0 = -6, w_1 = 5, w_2 = 5 \; and \; b = 1$$

$$SUM = (-6) * 1 + 1 * 5 + 0 * 5 = -1$$

$$SUM = (-6) * 1 + 1 * 5 + 1 * 5 = 4$$

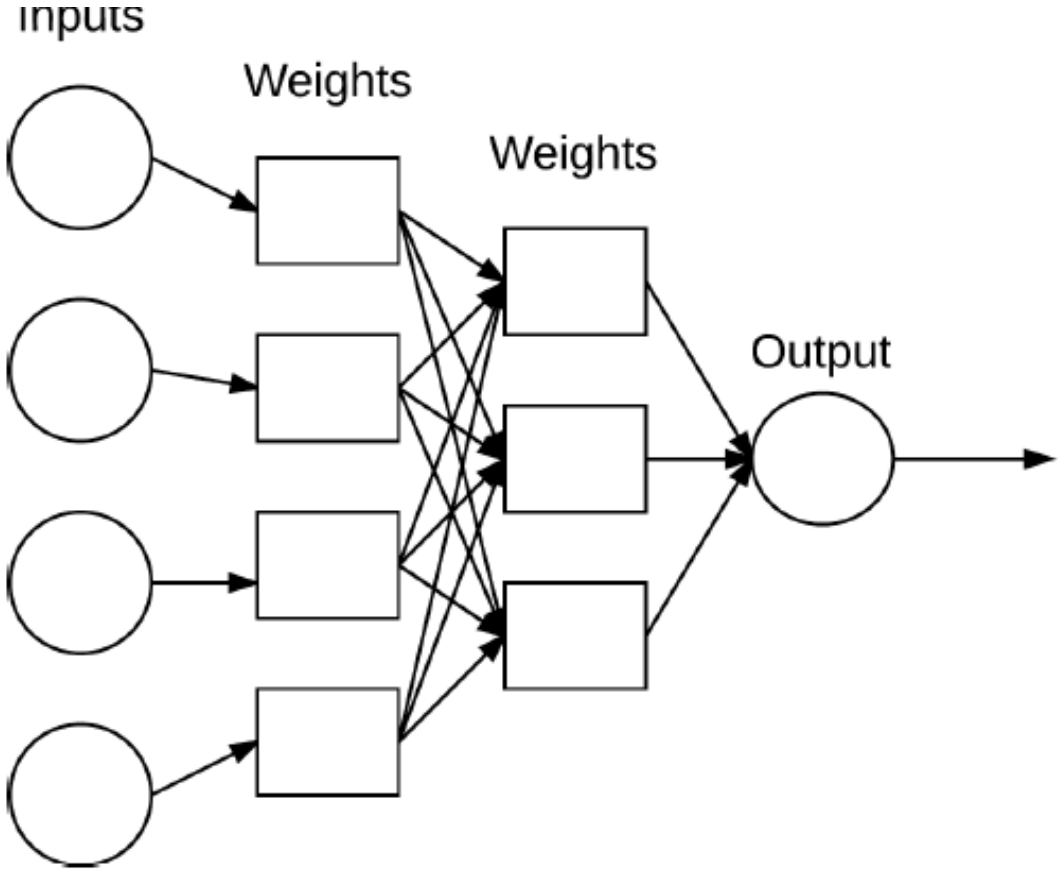



Figure : A simple neural network architecture. Inputs are presented to the network. Each connection carries a signal through the two hidden layers in the network. A final function computes the output class label.

# Neural Network Basics

Artificial Neural Networks and their relation to biology.

Relation to Biology

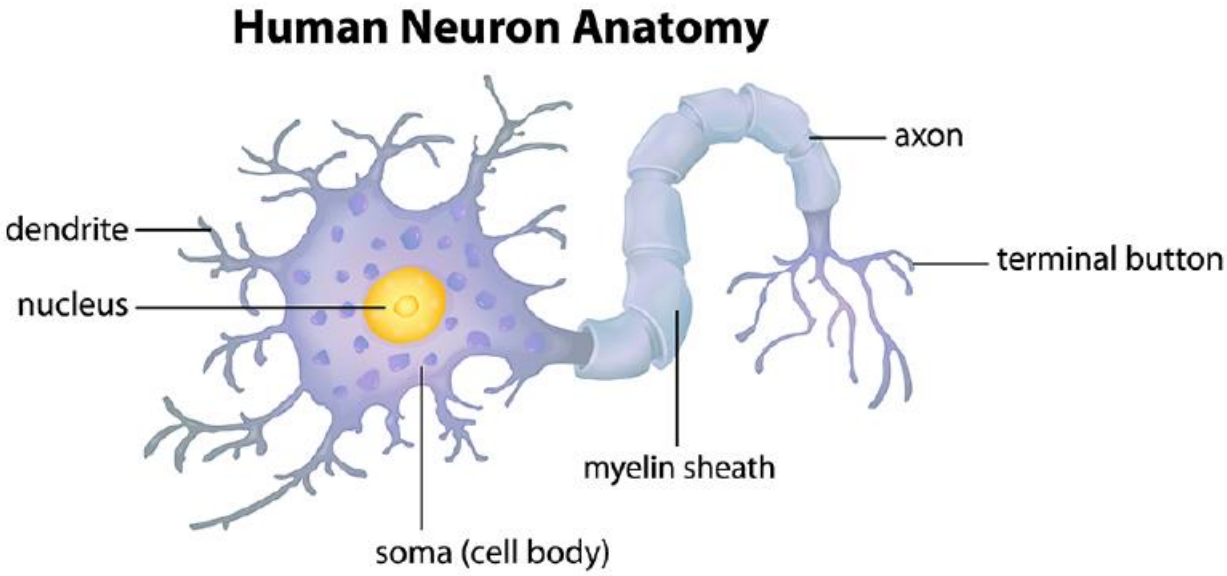



Figure: The structure of a biological neuron. Neurons are connected to other neurons through their dendrites and neurons.

# Neural Network Basics

Artificial Neural Networks and their relation to biology.

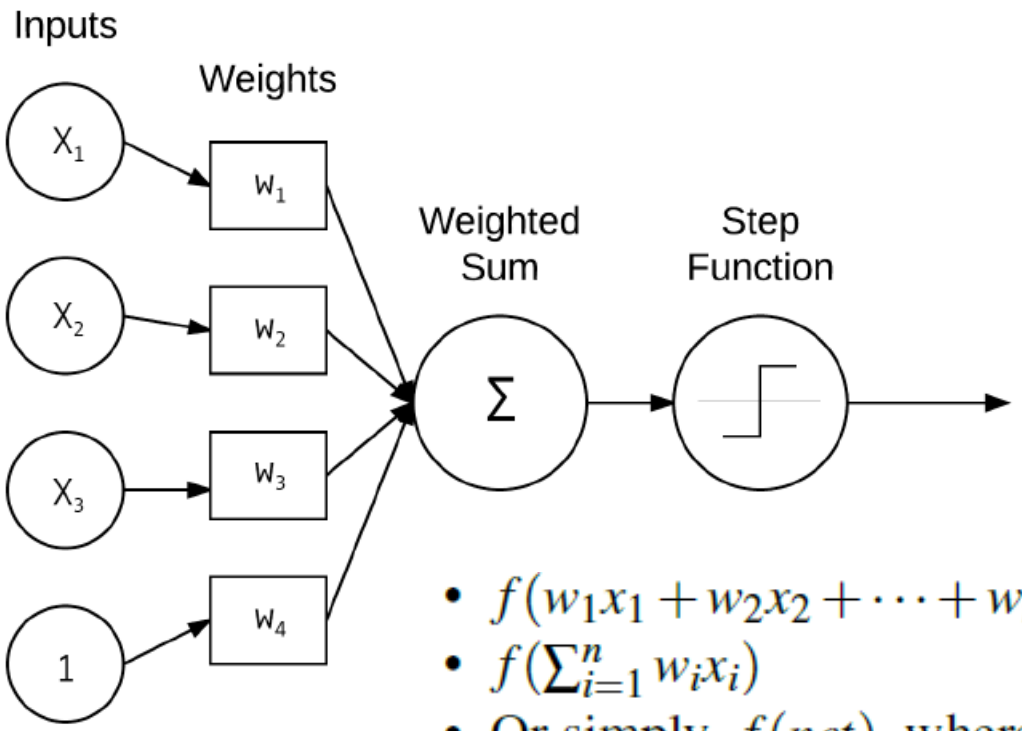
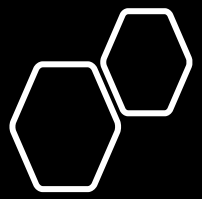



Figure: A simple NN that takes the weighted sum of the input x and weights w. This weighted sum is then passed through the activation function to determine if the neuron fires.

# Neural Network Basics

Artificial Neural Networks and their relation to biology.



# Perceptron Algorithm

First introduced by Rosenblatt in 1958

The Perceptron: A Probabilistic Model for Information Storage and Organization in the Brain is arguably the oldest and most simple of the ANN algorithms.

AND, OR, and XOR (exclusive OR) Datasets

# Perceptron Algorithm

- AND, OR, and XOR (exclusive OR) Datasets

| $x_0$ | $x_1$ | $x_0 \& x_1$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

| $x_0$ | $x_1$ | $x_0 \mid x_1$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

| $x_0$ | $x_1$ | $x_0 \wedge x_1$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

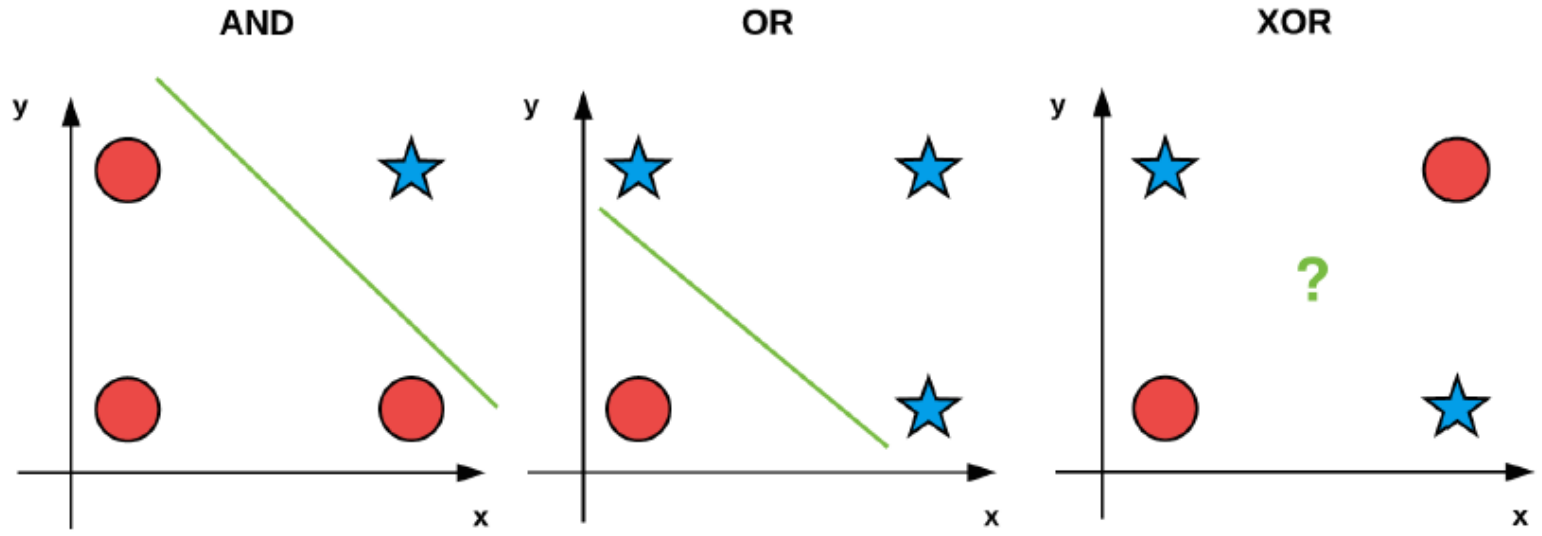
**Left**: The bitwise AND dataset. Given two inputs, the output is only 1 if both inputs are 1.

**Middle**: The bitwise OR dataset. Given two inputs, the output is 1 if either of the two inputs is 1.

**Right**: The XOR (e(X)clusive OR) dataset. Given two inputs, the output is 1 if and only if one of the inputs is 1, but not both.

# Perceptron Algorithm

- AND, OR, and XOR (exclusive OR) Datasets




Visualize the AND, OR, and XOR values (with red circles being zero outputs and blue stars one outputs)

***You'll notice an interesting pattern***

XOR is,

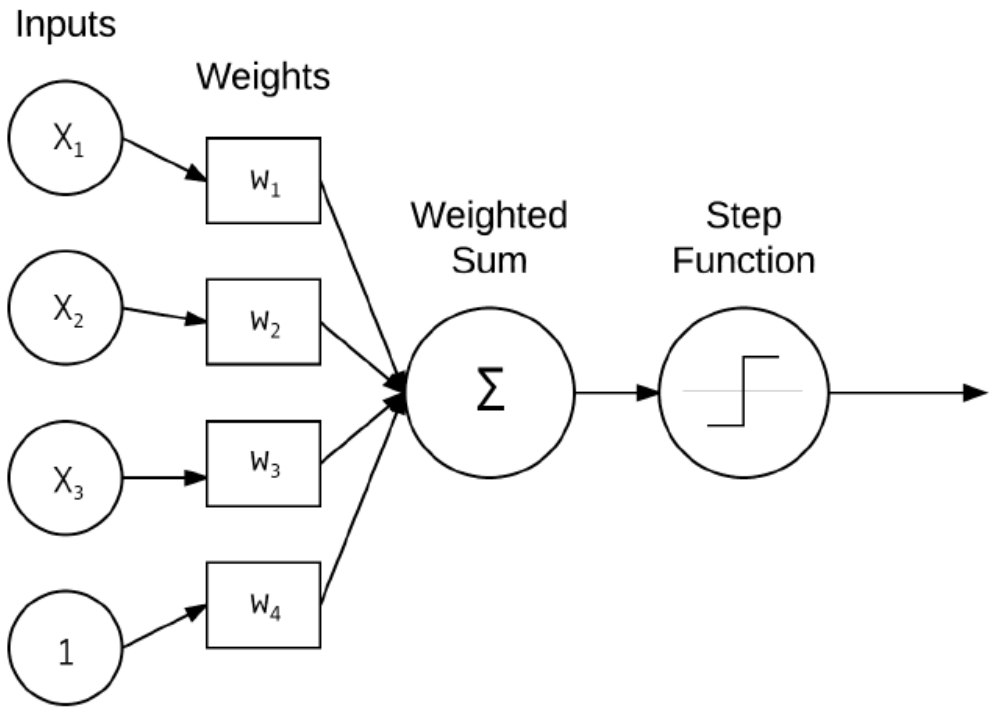
therefore, an example of a nonlinearly separable dataset.

# Perceptron Algorithm

- Ideally, we would like our AI algorithms to be able to separate nonlinear classes as most datasets encountered in the real-world are nonlinear
- Therefore, when constructing, debugging, and evaluating a given machine learning algorithm, we may use the bitwise values x0 and x1 as our design matrix and then try to predict the corresponding y values.

# Perceptron Algorithm

- Perceptron algorithm can correctly classify the AND and OR functions but fails to classify the XOR data.
- Perceptron as a system that learns using labeled examples (i.e., supervised learning) of feature vectors (or raw pixel intensities), mapping these inputs to their corresponding output class labels.




# Perceptron Algorithm

Architecture of the Perceptron network.

# Perceptron Algorithm

1. Initialize our weight vector $\boldsymbol{w}$ with small random values
2. Until Perceptron converges:
   (a) Loop over each feature vector $x_j$ and true class label $d_i$ in our training set $D$
   (b) Take $\boldsymbol{x}$ and pass it through the network, calculating the output value: $y_j = f(\boldsymbol{w}(\boldsymbol{t}) \cdot \boldsymbol{x}_j)$
   (c) Update the weights $\boldsymbol{w}$: $w_i(t+1) = w_i(t) + \alpha(d_j - y_j)x_{j,i}$ for all features $0 <= i <= n$

α is our learning rate and controls how large (or small) of a step we take.

The Perceptron training process is allowed to proceed until all training samples are classified correctly or a preset number of epochs is reached. Termination is ensured if a is sufficiently small and the training data is linearly separable.

**The Perceptron algorithm training procedure.**

# Perceptron Algorithm

1. Initialize our weight vector $\boldsymbol{w}$ with small random values
2. Until Perceptron converges:
   (a) Loop over each feature vector $x_j$ and true class label $d_i$ in our training set $D$
   (b) Take $\boldsymbol{x}$ and pass it through the network, calculating the output value: $y_j = f(\boldsymbol{w}(\boldsymbol{t}) \cdot \boldsymbol{x}_j)$
   (c) Update the weights $\boldsymbol{w}$: $w_i(t+1) = w_i(t) + \alpha(d_j - y_j)x_{j,i}$ for all features $0 <= i <= n$

So, what happens if our data is not linearly separable or we make a poor choice in a? Will training continue infinitely? In this case, no – we normally stop after a set number of epochs has been hit or if the number of misclassifications has not changed in a large number of epochs (indicating that the data is not linearly separable).

**The Perceptron algorithm training procedure.**

# Perceptron Algorithm

## The Perceptron algorithm Demo

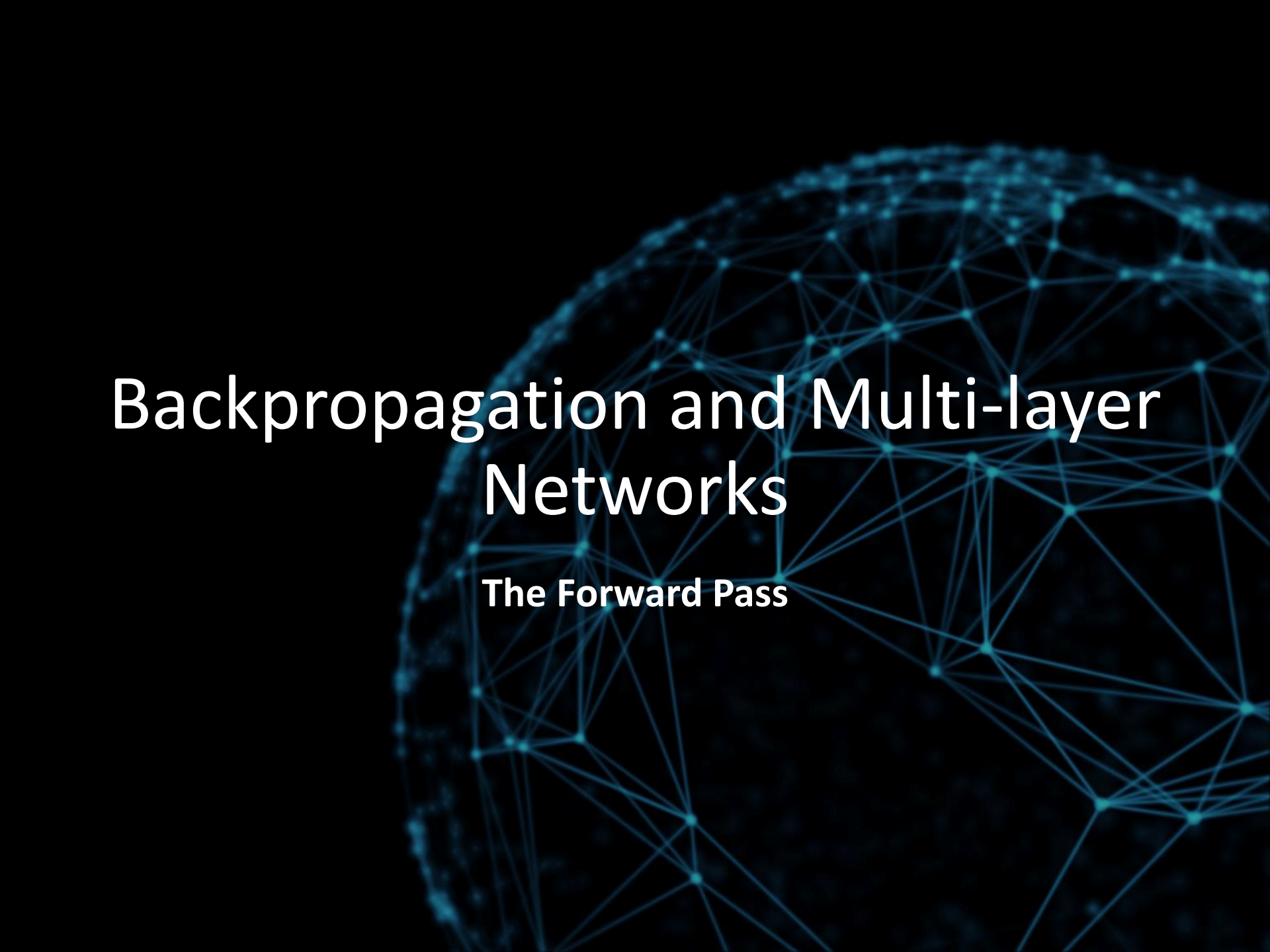
- No matter how many times you run this experiment with varying learning rates or different weight initialization schemes, you will never be able to correctly model the XOR function with a single layer Perceptron. Instead, what we need is **more layers with nonlinear activation functions – and with that, comes the start of deep learning**.

# Backpropagation and Multi-layer Networks

- The most important algorithm in neural network history.
- The cornerstone of modern neural networks and deep learning.
- In the 1970s

# Backpropagation and Multi-layer Networks

- The backpropagation algorithm consists of two phases:
  - 1. The *forward pass* where our inputs are passed through the network and output predictions obtained (also known as the propagation phase).
  - 2. The backward pass where we compute the gradient of the loss function at the final layer (i.e., predictions layer) of the network and use this gradient to recursively apply the chain rule to update the weights in our network (also known as the weight update phase).



# Backpropagation and Multi-layer Networks

**The Forward Pass**

## Intermediate **Variables**

(forward propagation)

$$h_1 = xW_1 + b_1$$

$$z_1 = \sigma(h_1)$$

$$z_2 = z_1W_2 + b_2$$

$$Loss = (z_2 - y)^2$$

## Intermediate **Gradients**

(backward propagation)

$$\frac{\partial h_1}{\partial x} = W_1^T$$

$$\frac{\partial z_1}{\partial h_1} = \sigma'(h_1) = z_1 \circ (1 - z_1)$$

$$\frac{\partial z_2}{\partial z_1} = W_2^\top$$

$$\frac{\partial Loss}{\partial z_2} = 2(z_2 - y)$$

# The backpropagation algorithm

The backpropagation equations provide us with a way of computing the gradient of the cost function. Let's explicitly write this out in the form of an algorithm:

1. **Input $x$:** Set the corresponding activation $a^1$ for the input layer.

2. **Feedforward:** For each $l = 2, 3, \ldots, L$ compute $z^l = w^l a^{l-1} + b^l$ and $a^l = \sigma(z^l)$.

3. **Output error $\delta^L$:** Compute the vector $\delta^L = \nabla_a C \odot \sigma'(z^L)$.

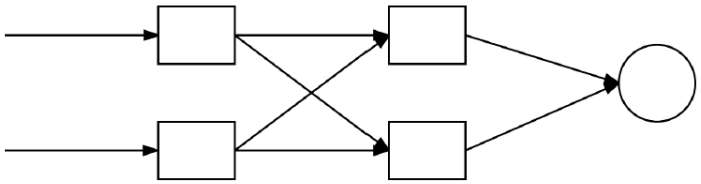
4. **Backpropagate the error:** For each $l = L-1, L-2, \ldots, 2$ compute $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$.

5. **Output:** The gradient of the cost function is given by $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$ and $\frac{\partial C}{\partial b^l_j} = \delta^l_j$.

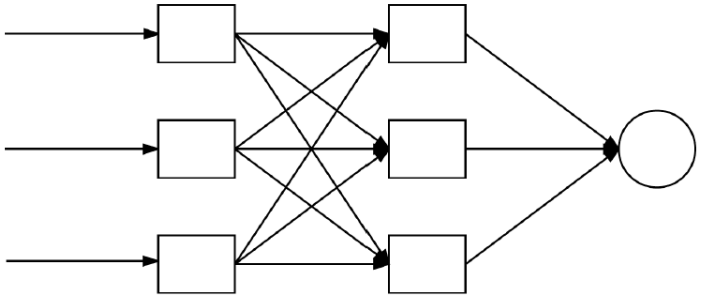
# Backpropagation and Multi-layer Networks

**The Forward Pass**

**2-2-1**



**3-3-1**



| $x_0$ | $x_1$ | $y$ |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

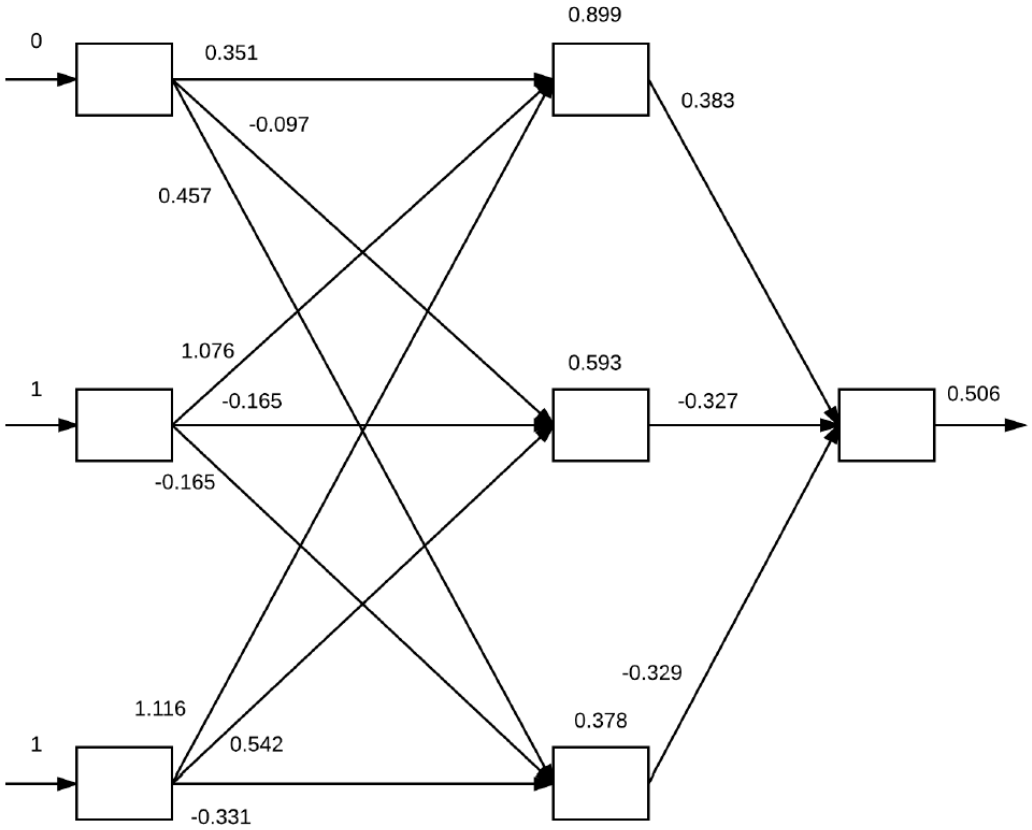
| $x_0$ | $x_1$ | $x_2$ |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

**Left**: The bitwise XOR dataset (including class labels).

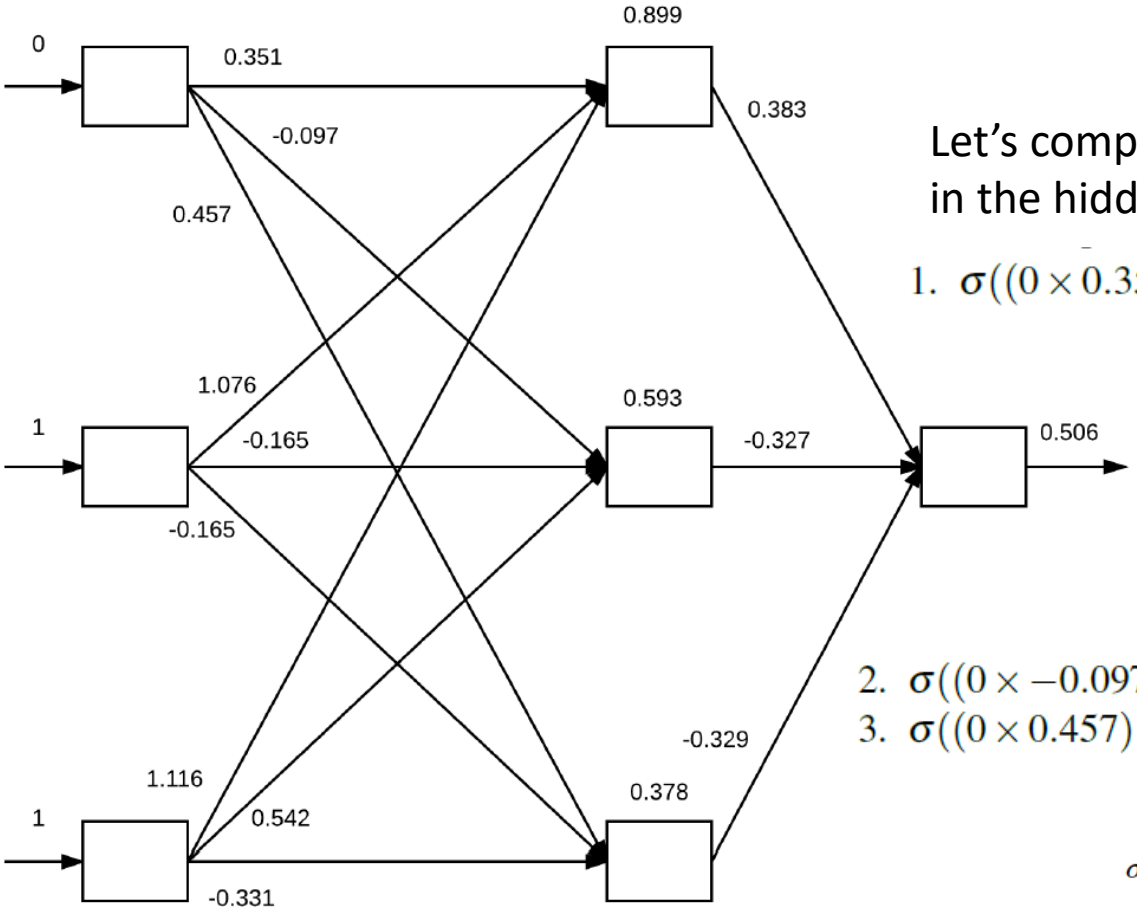
**Right**: The XOR dataset design matrix with a bias column inserted (excluding class labels for brevity).

# The Forward Pass

• An example of the forward propagation pass. The input vector [0;1;1] is presented to the network. The dot product between the inputs and weights are taken, followed by applying the sigmoid activation function to obtain the values in the hidden layer (0:899, 0:593, and 0:378, respectively). Finally, the dot product and sigmoid activation function is computed for the final layer, yielding an output of 0:506. Applying the step function to 0:506 yields 1, which is indeed the correct target class label.



# The Forward Pass



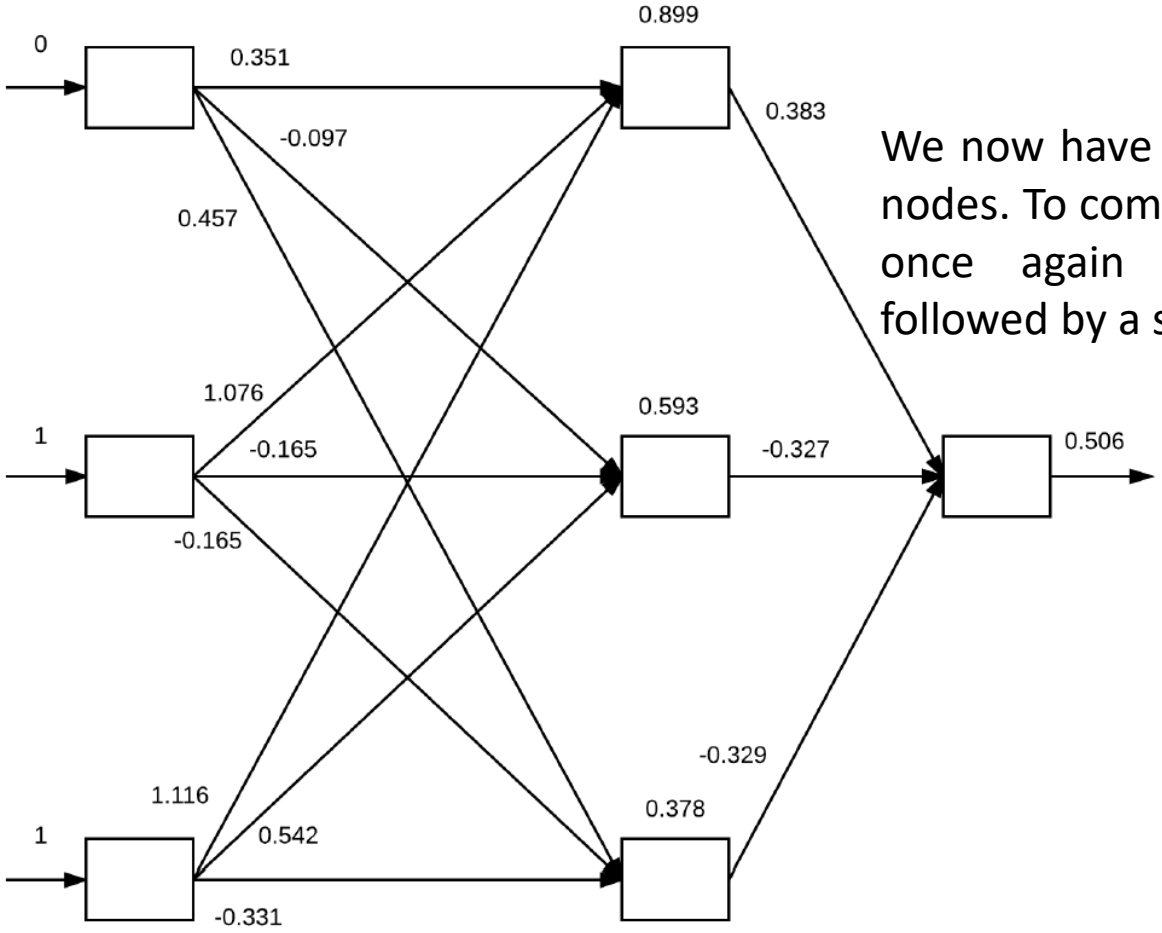

Let's compute the inputs to the three nodes in the hidden layers

1. $\sigma((0 \times 0.351) + (1 \times 1.076) + (1 \times 1.116)) = 0.899$
2. $\sigma((0 \times -0.097) + (1 \times -0.165) + (1 \times 0.542)) = 0.593$
3. $\sigma((0 \times 0.457) + (1 \times -0.165) + (1 \times -0.331)) = 0.378$

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

$$\frac{d\sigma(x)}{dx} = \frac{e^{-x}}{(1+e^{-x})^2} = \left(\frac{1+e^{-x}-1}{1+e^{-x}}\right)\left(\frac{1}{1+e^{-x}}\right) = (1-\sigma(x))\,\sigma(x)$$

# The Forward Pass




We now have our inputs to the hidden layer nodes. To compute the output prediction, we once again compute the dot product followed by a sigmoid activation:

$$f(net) = \begin{cases} 1 & if net > 0 \\ 0 & otherwise \end{cases}$$

$$((0.899 \times 0.383) + (0.593 \times -0.327) + (0.378 \times -0.329)) = 0.506$$

# The Forward Pass

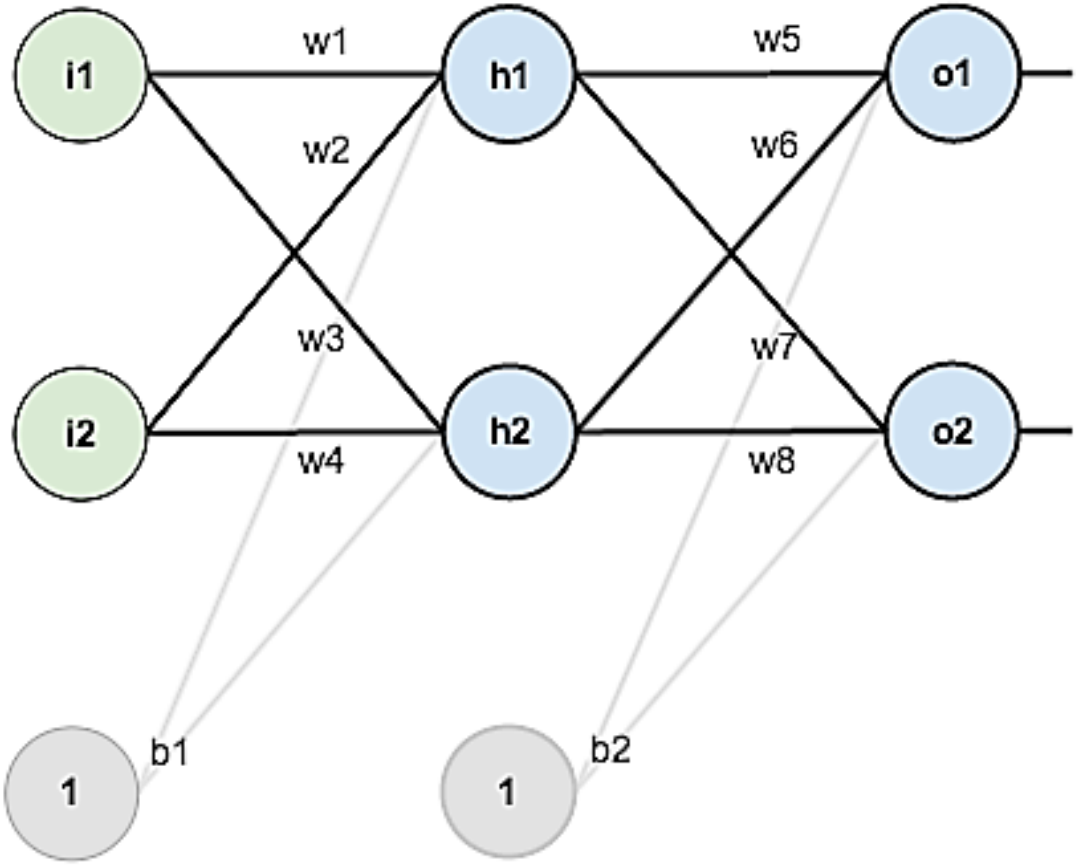
Applying the step function with $net = 0.506$ we see that our network predicts 1 which is, in fact, the correct class label. However, our network is *not very confident* in this class label – the predicted value 0.506 is very close to the threshold of the step. Ideally, this prediction should be closer to $0.98 - 0.99$, implying that our network has truly learned the underlying pattern in the dataset. In order for our network to actually "learn", we need to apply the backward pass.

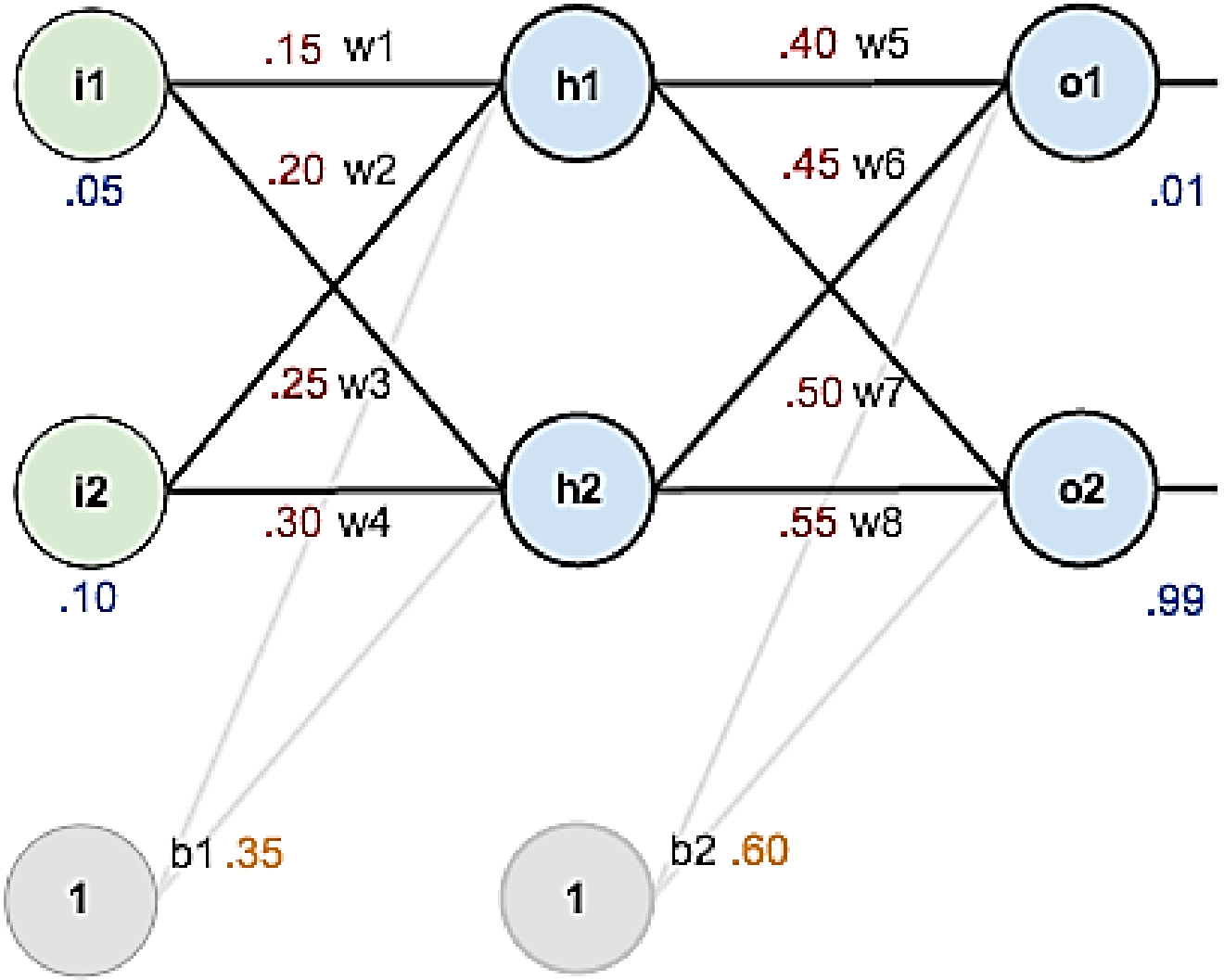
# The Backward Pass

$$\frac{\partial E}{\partial w_{i,j}} = \frac{\partial E}{\partial o_j} \frac{\partial o_j}{\partial net_j} \frac{\partial net_j}{\partial w_{i,j}}$$

Where we compute the gradient of the loss function and use this information to iteratively apply the chain rule to update the weights in our network.

# Backpropagation Algo. Basic structure:




i1
i2
h1
h2
o1
o2
.05
.10
.01
.99
.15 w1
.20 w2
.25 w3
.30 w4
.40 w5
.45 w6
.50 w7
.55 w8
1
b1 .35
1
b2 .60

Here's how we calculate the total net input for $h_1$:

$$net_{h1} = w_1 * i_1 + w_2 * i_2 + b_1 * 1$$

$$net_{h1} = 0.15 * 0.05 + 0.2 * 0.1 + 0.35 * 1 = 0.3775$$

We then squash it using the logistic function to get the output of $h_1$:

$$out_{h1} = \frac{1}{1+e^{-net_{h1}}} = \frac{1}{1+e^{-0.3775}} = 0.593269992$$

Carrying out the same process for $h_2$ we get:

$$out_{h2} = 0.596884378$$

We repeat this process for the output layer neurons, using the output from the hidden layer neurons as inputs.

Here's the output for $o_1$:

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$net_{o1} = 0.4 * 0.593269992 + 0.45 * 0.596884378 + 0.6 * 1 = 1.105905967$$

$$out_{o1} = \frac{1}{1+e^{-net_{o1}}} = \frac{1}{1+e^{-1.105905967}} = 0.75136507$$

And carrying out the same process for $o_2$ we get:

$$out_{o2} = 0.772928465$$

### Calculating the Total Error

We can now calculate the error for each output neuron using the squared error function and sum them to get the total error:

$$E_{total} = \sum \frac{1}{2}(target - output)^2$$

> Some sources refer to the target as the *ideal* and the output as the *actual*.

> The $\frac{1}{2}$ is included so that exponent is cancelled when we differentiate later on. The result is eventually multiplied by a learning rate anyway so it doesn't matter that we introduce a constant here [1].

For example, the target output for $o_1$ is 0.01 but the neural network output 0.75136507, therefore its error is:

$$E_{o1} = \frac{1}{2}(target_{o1} - out_{o1})^2 = \frac{1}{2}(0.01 - 0.75136507)^2 = 0.274811083$$

Repeating this process for $o_2$ (remembering that the target is 0.99) we get:

$$E_{o2} = 0.023560026$$

The total error for the neural network is the sum of these errors:

$$E_{total} = E_{o1} + E_{o2} = 0.274811083 + 0.023560026 = 0.298371109$$

## The Backwards Pass

Our goal with backpropagation is to update each of the weights in the network so that they cause the actual output to be closer the target output, thereby minimizing the error for each output neuron and the network as a whole.

# Output Layer

**Output Layer**

Consider $w_5$. We want to know how much a change in $w_5$ affects the total error, aka $\frac{\partial E_{total}}{\partial w_5}$.

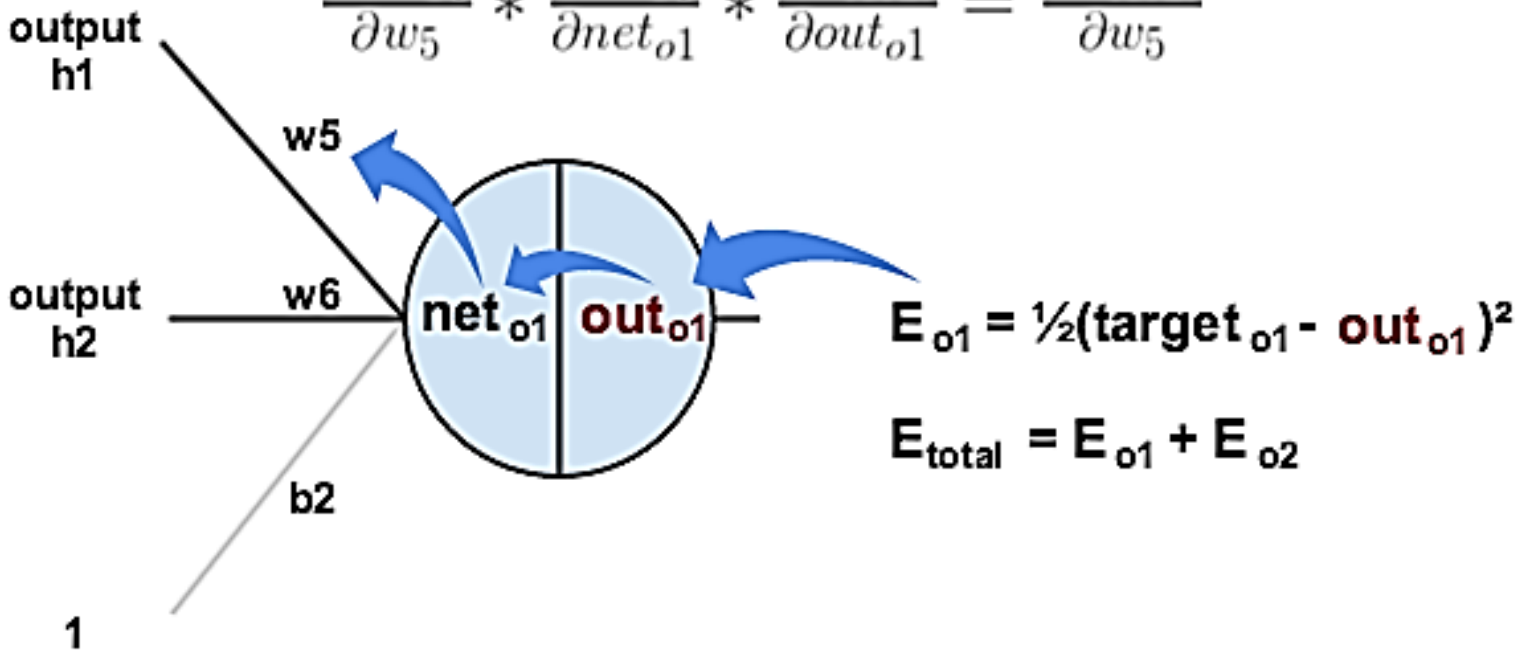
> $\frac{\partial E_{total}}{\partial w_5}$ is read as "the partial derivative of $E_{total}$ with respect to $w_5$". You can also say "the gradient with respect to $w_5$".

By applying the chain rule we know that:

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

# Visually, here's what we're doing

$$\frac{\partial net_{o1}}{\partial w_5} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial E_{total}}{\partial out_{o1}} = \frac{\partial E_{total}}{\partial w_5}$$



We need to figure out each piece in this equation.

First, how much does the total error change with respect to the output?

$$E_{total} = \frac{1}{2}(target_{o1} - out_{o1})^2 + \frac{1}{2}(target_{o2} - out_{o2})^2$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = 2 * \frac{1}{2}(target_{o1} - out_{o1})^{2-1} * -1 + 0$$

$$\frac{\partial E_{total}}{\partial out_{o1}} = -(target_{o1} - out_{o1}) = -(0.01 - 0.75136507) = 0.74136507$$

$-(target - out)$ is sometimes expressed as $out - target$

When we take the partial derivative of the total error with respect to $out_{o1}$, the quantity $\frac{1}{2}(target_{o2} - out_{o2})^2$ becomes zero because $out_{o1}$ does not affect it which means we're taking the derivative of a constant which is zero.

Next, how much does the output of $o_1$ change with respect to its total net input?

The partial derivative of the logistic function is the output multiplied by 1 minus the output:

$$out_{o1} = \frac{1}{1+e^{-net_{o1}}}$$

$$\frac{\partial out_{o1}}{\partial net_{o1}} = out_{o1}(1 - out_{o1}) = 0.75136507(1 - 0.75136507) = 0.186815602$$

Finally, how much does the total net input of $o1$ change with respect to $w_5$?

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$\frac{\partial net_{o1}}{\partial w_5} = 1 * out_{h1} * w_5^{(1-1)} + 0 + 0 = out_{h1} = 0.593269992$$

Putting it all together:

$$\frac{\partial E_{total}}{\partial w_5} = \frac{\partial E_{total}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial w_5}$$

$$\frac{\partial E_{total}}{\partial w_5} = 0.74136507 * 0.186815602 * 0.593269992 = 0.082167041$$

To decrease the error, we then subtract this value from the current weight (optionally multiplied by some learning rate, eta, which we'll set to 0.5):

$$w_5^+ = w_5 - \eta * \frac{\partial E_{total}}{\partial w_5} = 0.4 - 0.5 * 0.082167041 = 0.35891648$$

We can repeat this process to get the new weights $w_6$, $w_7$, and $w_8$:

$$w_6^+ = 0.408666186$$

$$w_7^+ = 0.511301270$$

$$w_8^+ = 0.561370121$$

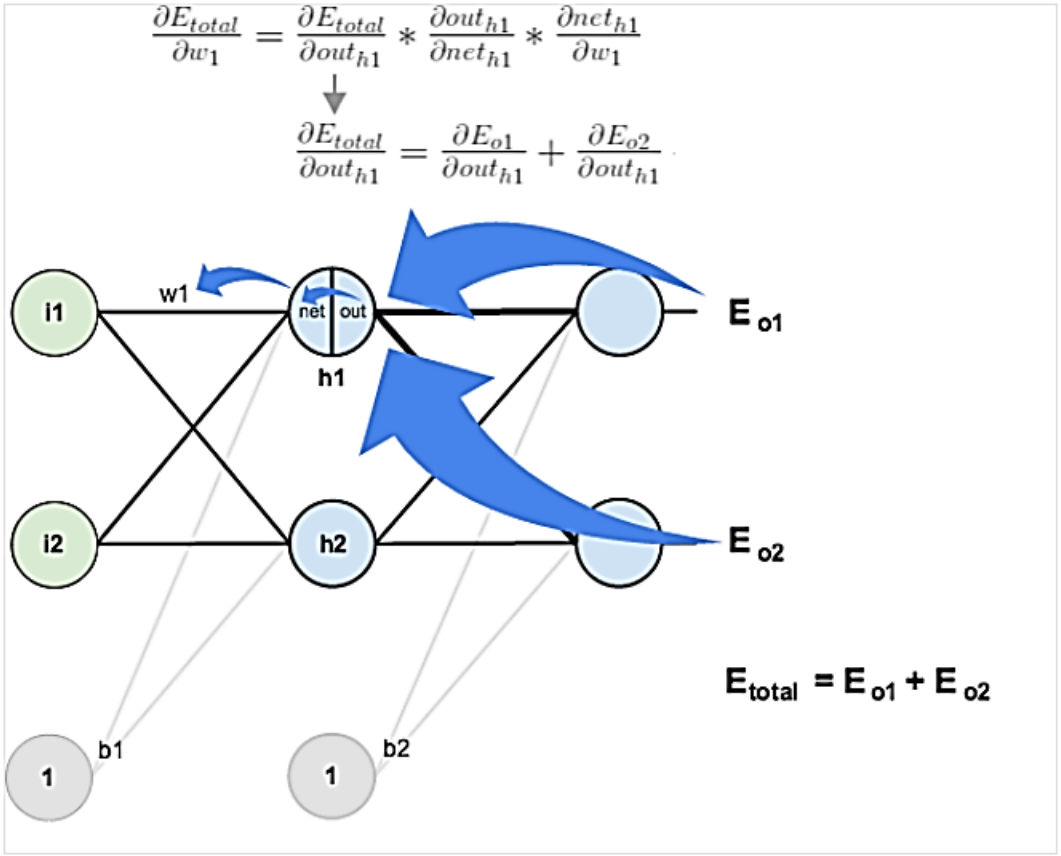
### Hidden Layer

Next, we'll continue the backwards pass by calculating new values for $w_1$, $w_2$, $w_3$, and $w_4$.

Big picture, here's what we need to figure out:

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

Visually:




We're going to use a similar process as we did for the output layer, but slightly different to account for the fact that the output of each hidden layer neuron contributes to the output (and therefore error) of multiple output neurons. We know that $out_{h1}$ affects both $out_{o1}$ and $out_{o2}$ therefore the $\frac{\partial E_{total}}{\partial out_{h1}}$ needs to take into consideration its effect on the both output neurons:

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$

Starting with $\frac{\partial E_{o1}}{\partial out_{h1}}$:

$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial out_{h1}}$$

We can calculate $\frac{\partial E_{o1}}{\partial net_{o1}}$ using values we calculated earlier:

$$\frac{\partial E_{o1}}{\partial net_{o1}} = \frac{\partial E_{o1}}{\partial out_{o1}} * \frac{\partial out_{o1}}{\partial net_{o1}} = 0.74136507 * 0.186815602 = 0.138498562$$

And $\frac{\partial net_{o1}}{\partial out_{h1}}$ is equal to $w_5$:

$$net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1$$

$$\frac{\partial net_{o1}}{\partial out_{h1}} = w_5 = 0.40$$

Plugging them in:

$$\frac{\partial E_{o1}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial net_{o1}} * \frac{\partial net_{o1}}{\partial out_{h1}} = 0.138498562 * 0.40 = 0.055399425$$

Following the same process for $\frac{\partial E_{o2}}{\partial out_{h1}}$, we get:

$$\frac{\partial E_{o2}}{\partial out_{h1}} = -0.019049119$$

Therefore:

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}} = 0.055399425 + -0.019049119 = 0.036350306$$

Now that we have $\frac{\partial E_{total}}{\partial out_{h1}}$, we need to figure out $\frac{\partial out_{h1}}{\partial net_{h1}}$ and then $\frac{\partial net_{h1}}{\partial w}$ for each weight:

$$out_{h1} = \frac{1}{1+e^{-net_{h1}}}$$

$$\frac{\partial out_{h1}}{\partial net_{h1}} = out_{h1}(1 - out_{h1}) = 0.59326999(1 - 0.59326999) = 0.241300709$$

We calculate the partial derivative of the total net input to $h_1$ with respect to $w_1$ the same as we did for the output neuron:

$$net_{h1} = w_1 * i_1 + w_3 * i_2 + b_1 * 1$$

$$\frac{\partial net_{h1}}{\partial w_1} = i_1 = 0.05$$

Putting it all together:

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial w_1} = 0.036350306 * 0.241300709 * 0.05 = 0.000438568$$

We can now update $w_1$:

$$w_1^+ = w_1 - \eta * \frac{\partial E_{total}}{\partial w_1} = 0.15 - 0.5 * 0.000438568 = 0.149780716$$

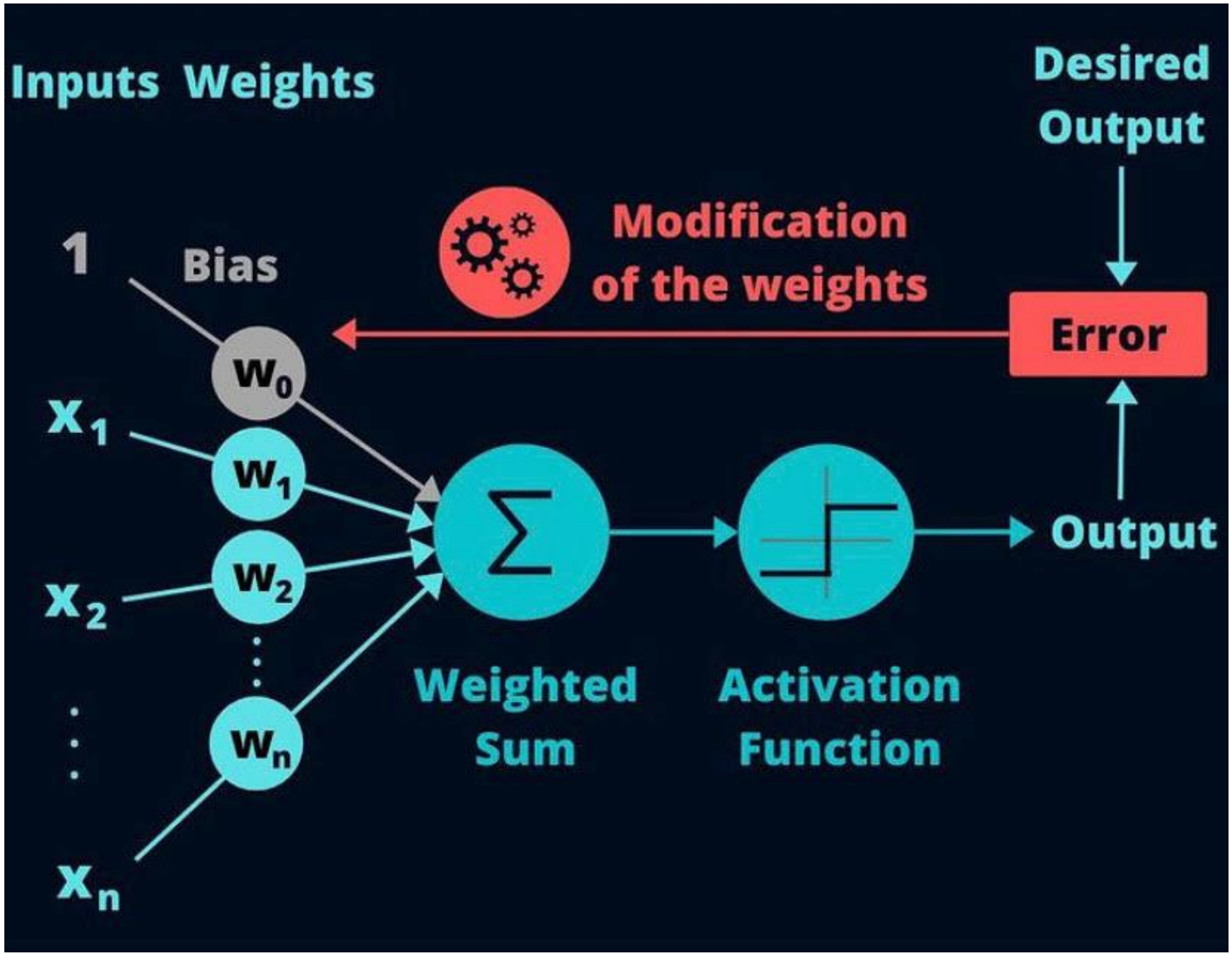
Repeating this for $w_2$, $w_3$, and $w_4$

$$w_2^+ = 0.19956143$$

$$w_3^+ = 0.24975114$$

$$w_4^+ = 0.29950229$$

Finally, we've updated all of our weights! When we fed forward the 0.05 and 0.1 inputs originally, the error on the network was 0.298371109. After this first round of backpropagation, the total error is now down to 0.291027924. It might not seem like much, but after repeating this process 10,000 times, for example, the error plummets to 0.0000351085. At this point, when we feed forward 0.05 and 0.1, the two outputs neurons generate 0.015912196 (vs 0.01 target) and 0.984065734 (vs 0.99 target).


Inputs Weights
Desired Output
Modification of the weights
1
Bias
Error
$w_0$
$x_1$
$w_1$
$w_2$
$x_2$
Output
$w_n$
Weighted Sum
Activation Function
$x_n$

# Multi-layer Networks

# Image Fundamentals

- We'll start with the buildings blocks of an image – the pixel.
- We'll discuss exactly what a pixel is
- How they are used to form an image
- How to access pixels that are represented as NumPy arrays

# Image Fundamentals

- Pixels: The Building Blocks of Images
  - Every image consists of a set of pixels.
  - Pixel is considered the “color” or the “intensity” of light that appears in a given place in our image

# Pixels: The Building Blocks of Images

Most pixels are represented in two ways:

1. Grayscale/single channel
2. Color

- **Resolution of 1,000 X 750, meaning that it is 1,000 pixels wide and 750 pixels tall.**
- **We can conceptualize an image as a (multidimensional) matrix.**
- **In this case, our matrix has 1, 000 columns (the width) with 750 rows (the height).**
- **Overall, there are 1, 000 X 750 = 750, 000 total pixels in our image.**

# Pixels: The Building Blocks of Images

- In a grayscale image, each pixel is a scalar value between 0 and 255, where zero corresponds to "**black**" and 255 being "white". Values between 0 and 255 are varying shades of gray, where
- Values closer to 0 are darker and values closer to 255 are lighter. The grayscale gradient image in
- Figure demonstrates darker pixels on the left-hand side and progressively lighter pixels on the right-hand side.

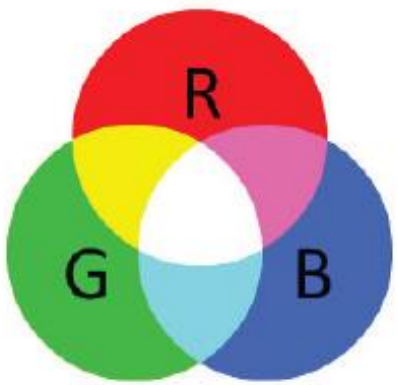
# Pixels: The Building Blocks of Images

- Color pixels; however, are normally represented in the RGB color space (other color spaces do exist )

Each Red, Green, and Blue channel can have values defined in the range $[0, 255]$ for a total of 256 "shades", where 0 indicates no representation and 255 demonstrates full representation. Given that the pixel value only needs to be in the range $[0, 255]$, we normally use 8-bit unsigned integers to represent the intensity.

**Left**: The RGB color space is additive. The more red, green and blue you mix together, the closer you get to white.

**Right**: The RGB cube.




# Pixels: The Building Blocks of Images

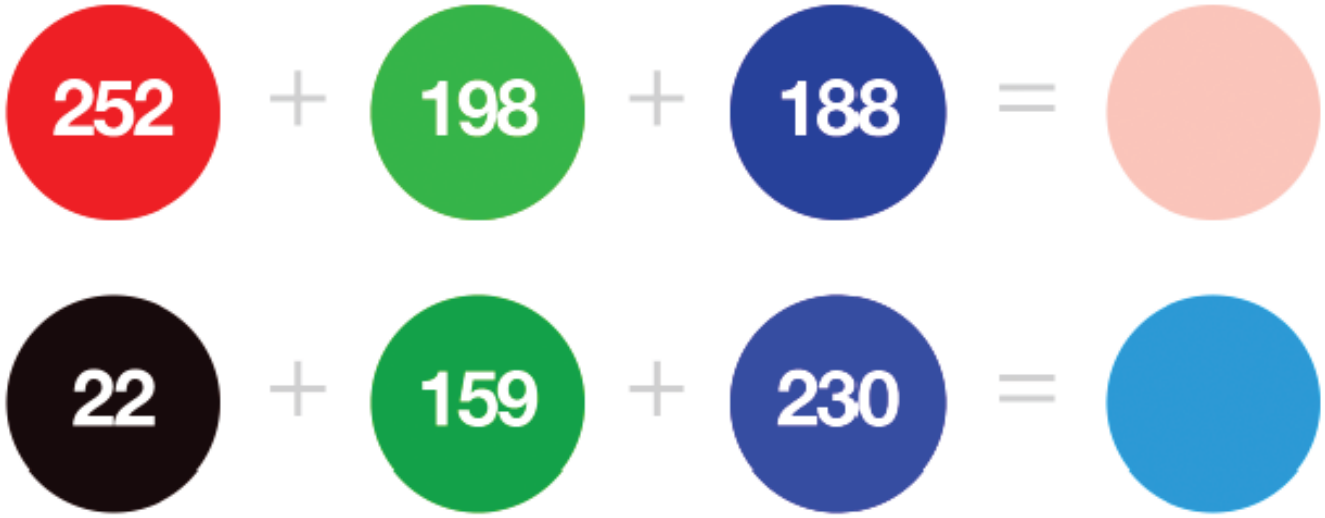
- Color pixels; however, are normally represented in the RGB color space (other color spaces do exist )

To make this example more concrete, let's again consider the color "white" – we would fill each of the red, green, and blue buckets up completely, like this: `(255, 255, 255)`. Then, to create the color black, we would empty each of the buckets out `(0, 0, 0)`, as black is the absence of color. To create a pure red color, we would fill up the red bucket (and only the red bucket) completely: `(255, 0, 0)`.

The RGB color space is also commonly visualized as a cube Since an RGB color is defined as a 3-valued tuple, which each value in the range [0;255] we can thus think of the cube containing 256 X 256 X 256 = **16, 777, 216 possible colors,** depending on how much Red, Green, and Blue are placed into each bucket.

# Pixels: The Building Blocks of Images

- Color pixels; however, are normally represented in the RGB color space (other color spaces do exist )

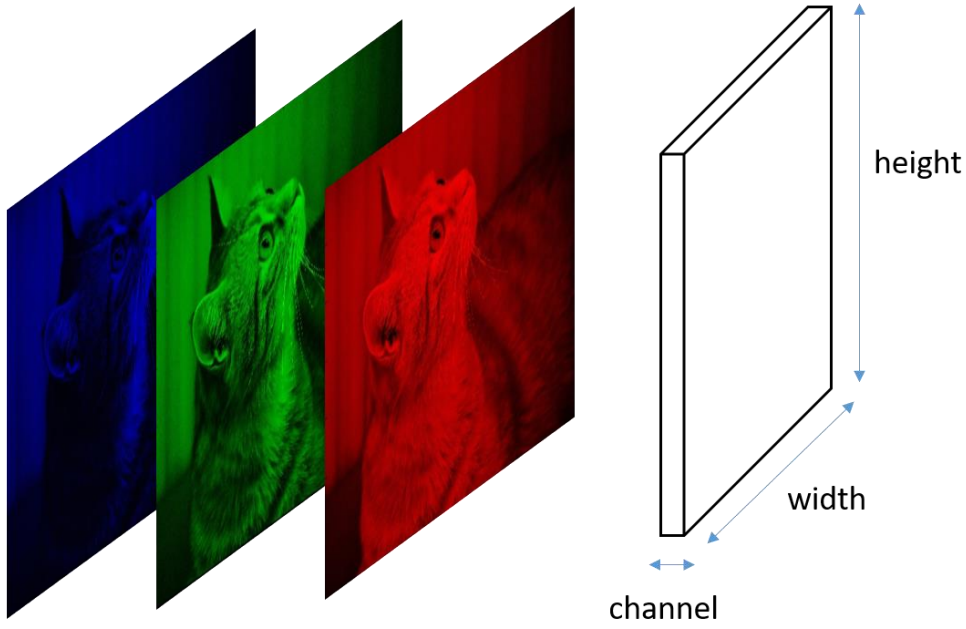



**It doesn’t mimic how humans perceive color.**

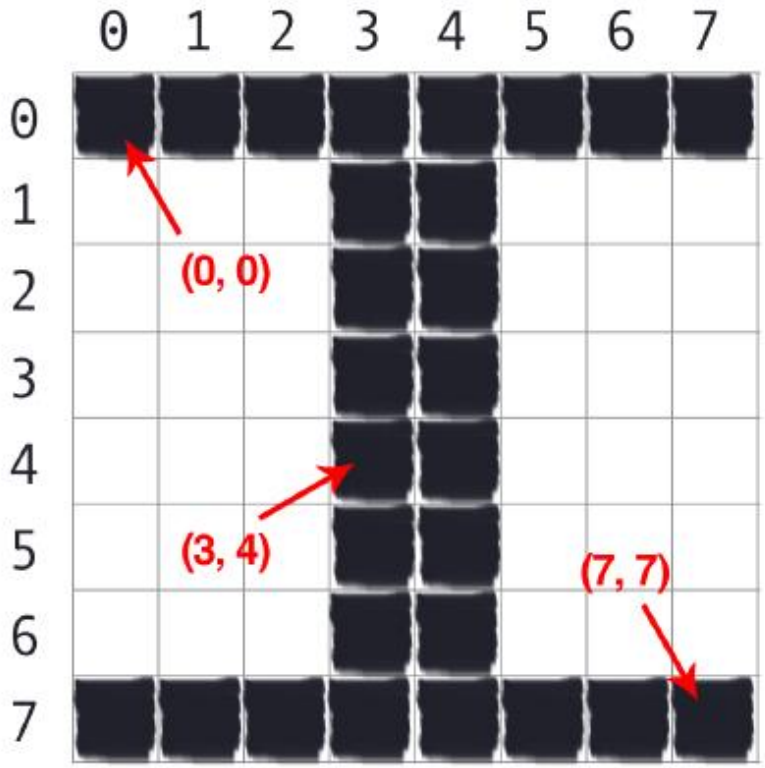
# Pixels: The Building Blocks of Images

## Forming an Image From Channels

As we now know, an RGB image is represented by three values, one for each of the Red, Green, and Blue components, respectively. We can conceptualize an RGB image as consisting of *three independent matrices* of width `W` and height `H`, one for each of the RGB components, as shown in Figure . We can combine these three matrices to obtain a multi-dimensional array with shape $W \times H \times D$ where `D` is the **depth** or **number of channels** (for the RGB color space, `D=3`):



# The Image Coordinate System




The letter “I” placed on a piece of graph paper. Pixels are accessed by their (x;y)- coordinates, where we go **x columns to the right and y rows down**

As an example of zero-indexing, consider the pixel **4 columns to the right** and **5 rows down is indexed by the point (3;4)**

# Images as NumPy Arrays

Image processing libraries such as OpenCV and scikit-image represent RGB images as multi-dimensional NumPy arrays with shape `(height, width, depth)`. Readers who are using image processing libraries for the first time are often confused by this representation – why does the *height* come before the *width* when we normally think of an image in terms of width *first* then height?

```
1 import cv2
2 image = cv2.imread("example.png")
3 print(image.shape)
4 cv2.imshow("Image", image)
5 cv2.waitKey(0)
```

```
$ python load_display.py
(248, 300, 3)
```

# Images as NumPy Arrays

```
$ python load_display.py
(248, 300, 3)
```

This image has a width of 300 pixels (the number of columns), a height of 248 pixels (the number of rows), and a depth of 3 (the number of channels). To access an individual pixel value from our `image` we use simple NumPy array indexing:

```
1 (b, g, r) = image[20, 100] # accesses pixel at x=100, y=20
2 (b, g, r) = image[75, 25] # accesses pixel at x=25, y=75
3 (b, g, r) = image[90, 85] # accesses pixel at x=85, y=90
```

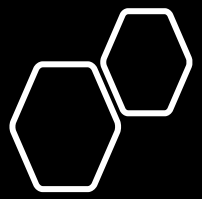
Again, notice how the `y` value is passed in *before* the `x` value – this syntax may feel uncomfortable at first, but it is consistent with how we access values in a matrix: first we specify the row number then the column number. From there, we are given a tuple representing the Red, Green, and Blue components of the image.

# RGB and BGR Ordering

It's important to note that OpenCV stores RGB channels in *reverse order*. While we normally think in terms of Red, Green, and Blue, OpenCV actually stores the pixel values in Blue, Green, Red order.

Why does OpenCV do this? The answer is simply historical reasons. Early developers of the OpenCV library chose the BGR color format because the BGR ordering was popular among camera manufacturers and other software developers at the time

Simply put – this BGR ordering was made for historical reasons and a choice that we now have to live with. It's a small caveat, but an important one to keep in mind when working with OpenCV.



# Scaling and Aspect Ratios

*Scaling*, or simply *resizing*, is the process of increasing or decreasing the size of an image in terms of width and height.

Ratio of the **width to the height of the image.**

**Ignoring the aspect ratio can lead to images that look compressed and distorted.**

# Scaling and Aspect Ratios

312x234; aspect ratio=1.33

236x86; aspect ratio=2.74

100x118; aspect ratio=0.84

Figure . **Left:** Original image. **Top and Bottom:** Resulting distorted images after resizing without preserving the aspect ratio (i.e., the ratio of the width to the height of the image).

# Scaling and Aspect Ratios

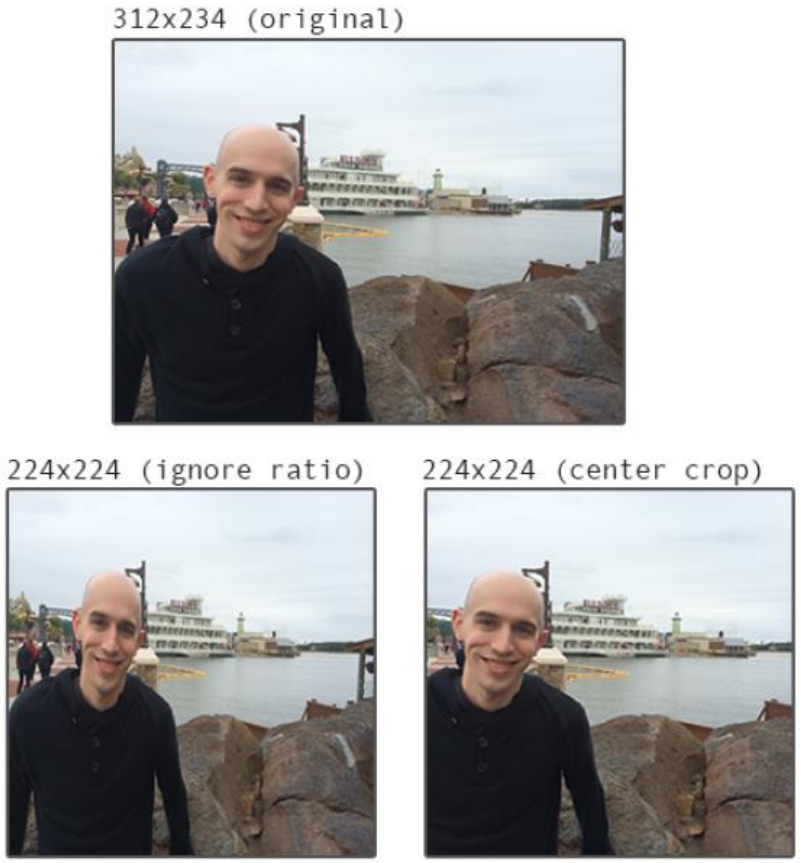



Figure . : **Top:** Our original input image. **Bottom left:** Resizing an image to $224 \times 224$ pixels by ignoring the aspect ratio. **Bottom right:** Resizing an image $224 \times 224$ pixels by first resizing along the shortest dimension and then taking the center crop.

# DEMO

- Implementing Backpropagation with Python

# DEMO

- Backpropagation with Python Example #1: Bitwise XOR

# DEMO

- Backpropagation with Python Example: MNIST Sample

# Thank You