

INTRODUCTION TO DEEP LEARNING: NEURAL NETWORKS

BY

SAIFUL BARI IFTU

LECTURER, DEPT. OF CSE, BRAC UNIVERSITY

CONTENTS

What Is Deep Learning (DL)?

Why DL?

Types of DL Models

Human Brain & Neurons

Neural Networks

Multi-Layer Perceptron (MLP)

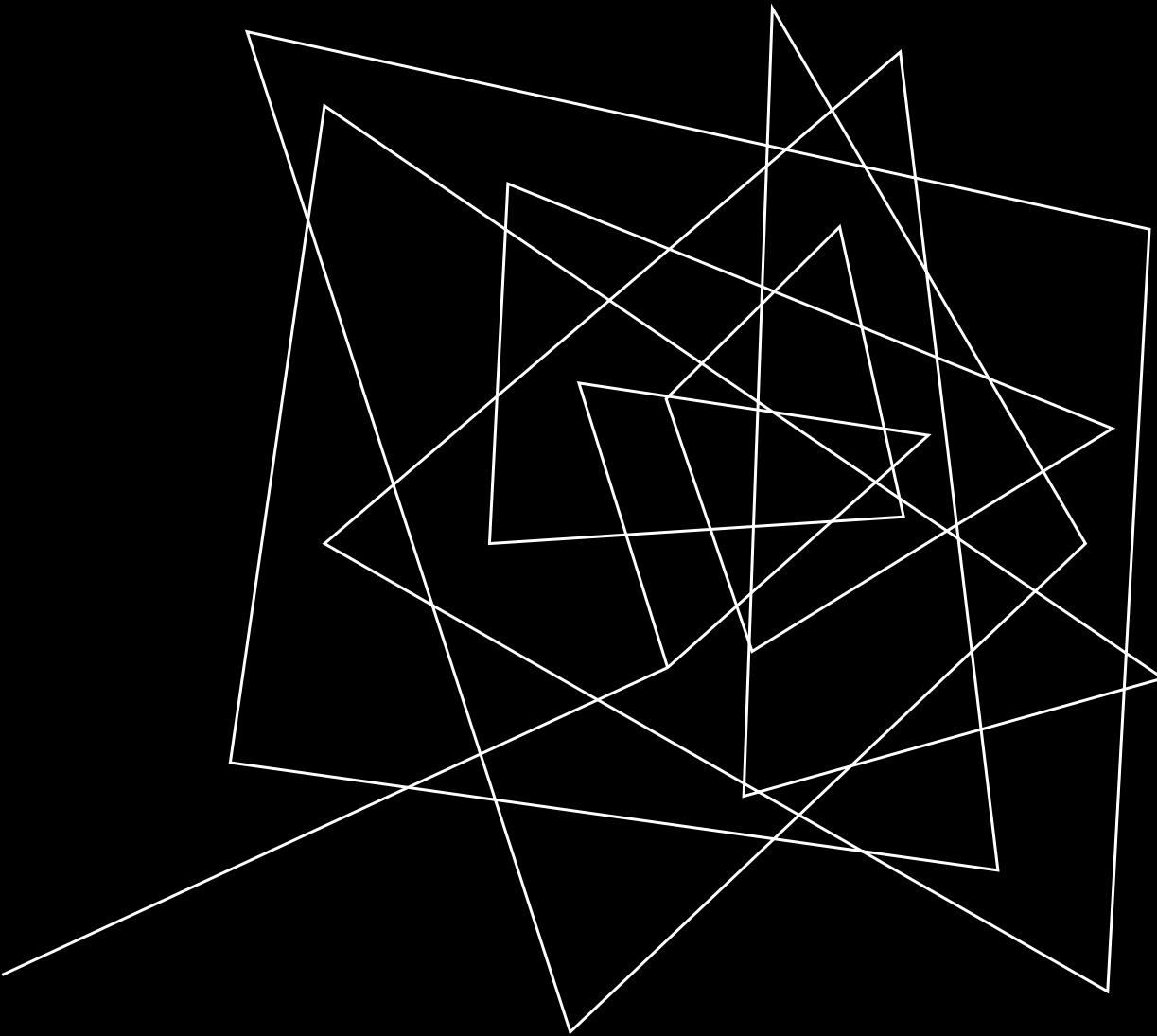
Forward Propagation

Activation Functions

Backpropagation

Challenges with DL

Dropout & Batch Normalization



WHAT IS
DEEP LEARNING?

WHAT IS DEEP LEARNING?

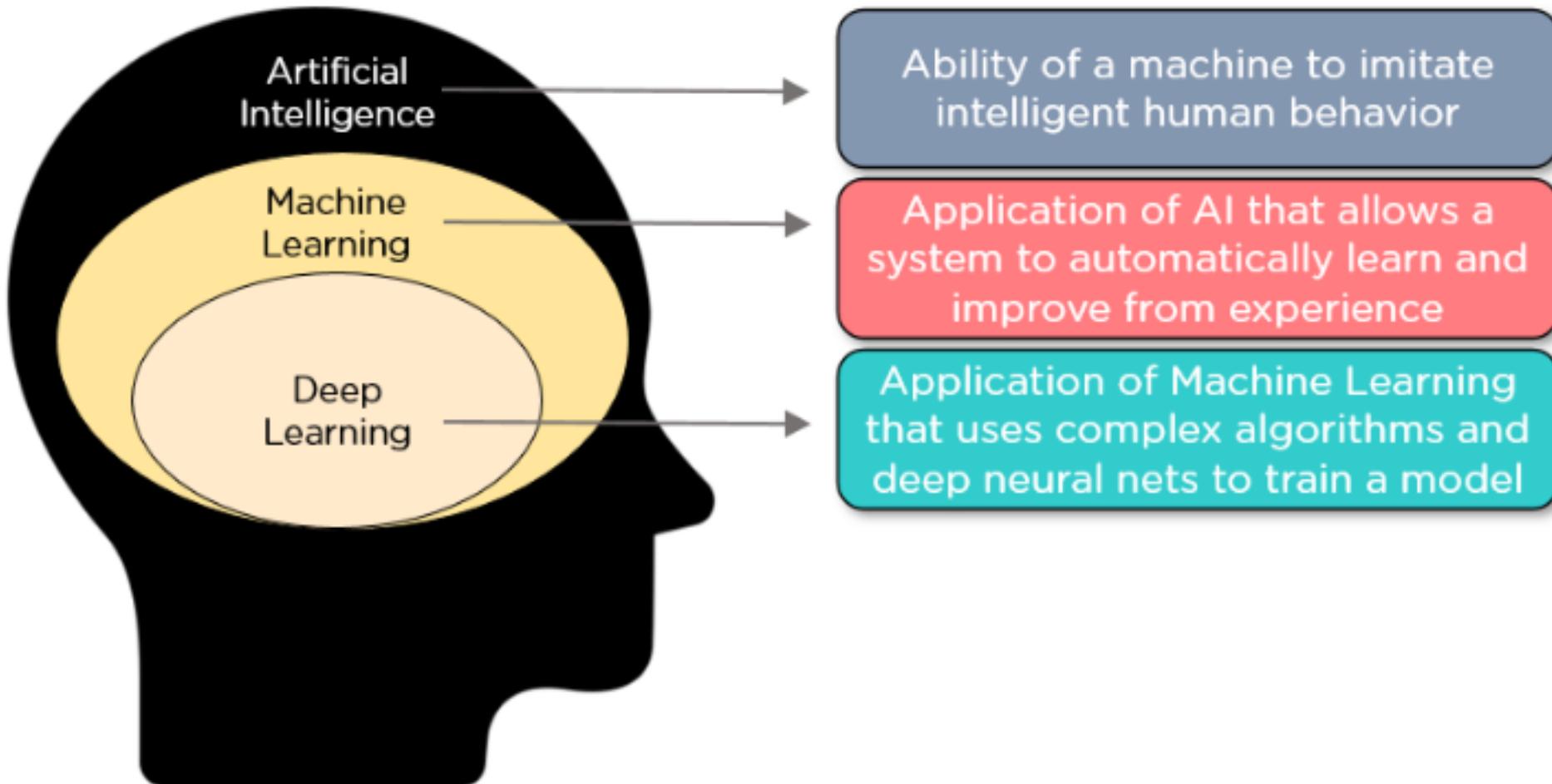
Formal Definition

- **Deep Learning (DL)** is a subset of machine learning that focuses on **neural networks** with **multiple layers**, known as deep neural networks. It is inspired by the human brain's structure and is designed to simulate the way humans learn from data. Deep learning is particularly well-suited for handling large, complex datasets and excels in tasks such as image recognition, natural language processing, and speech recognition.

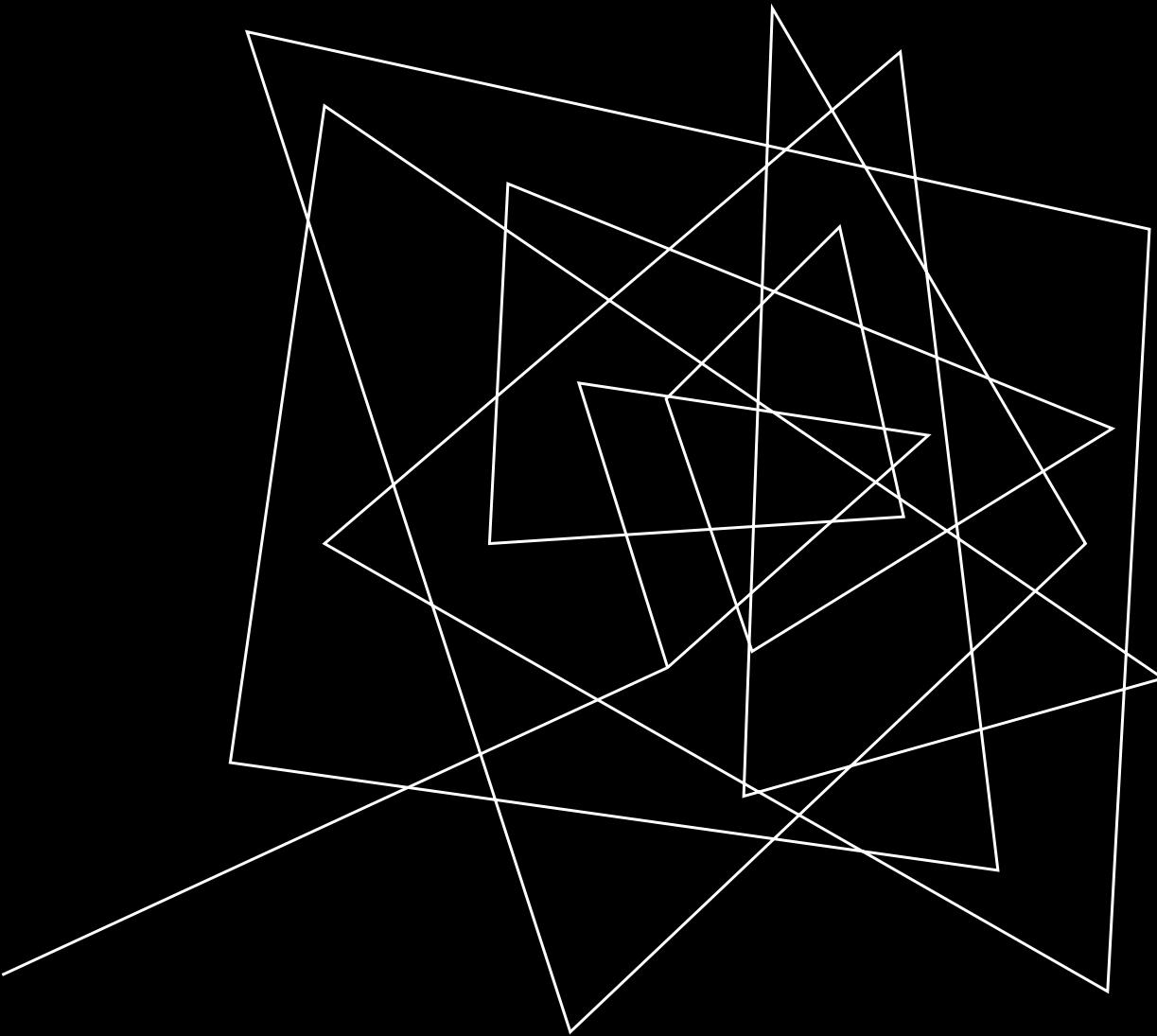
In Short

- Deep learning is a type of machine learning based on artificial neural networks in which **multiple layers of processing** are used to **extract progressively higher-level features from data**.

AI vs ML vs DL



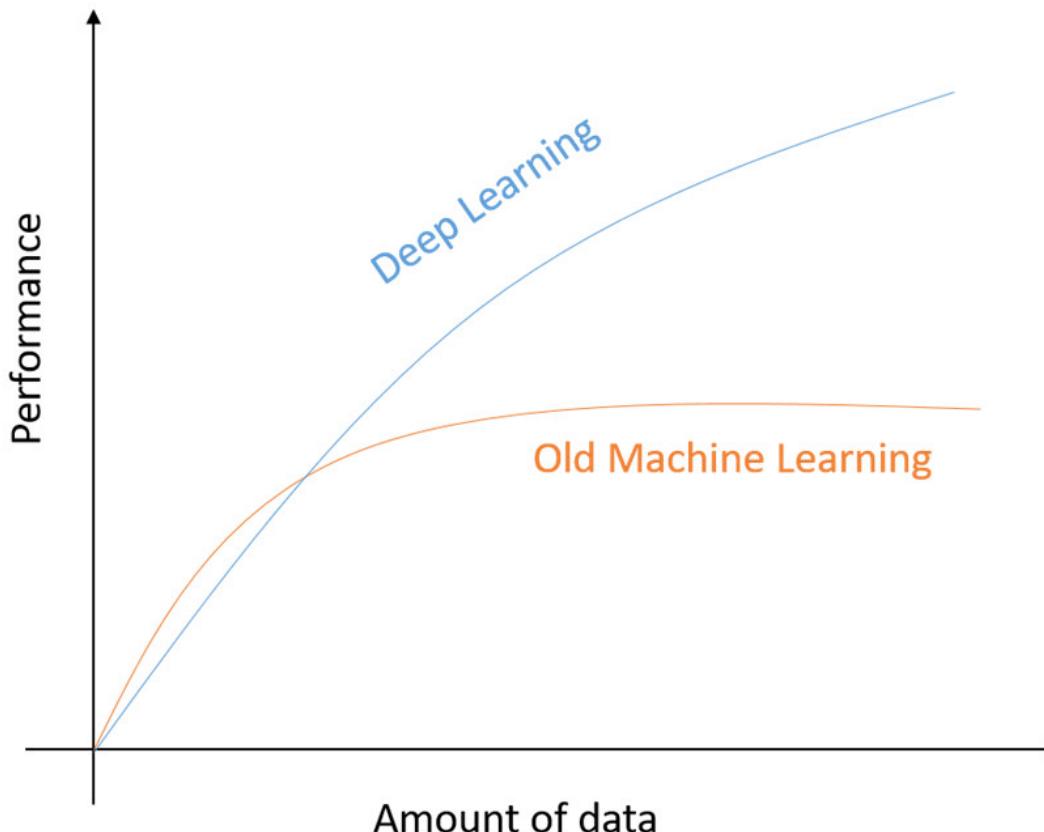
Source: <https://medium.com/@ramaleelamadishetti/regularization-in-deep-learning-6610207409e2>



WHY
DEEP LEARNING?

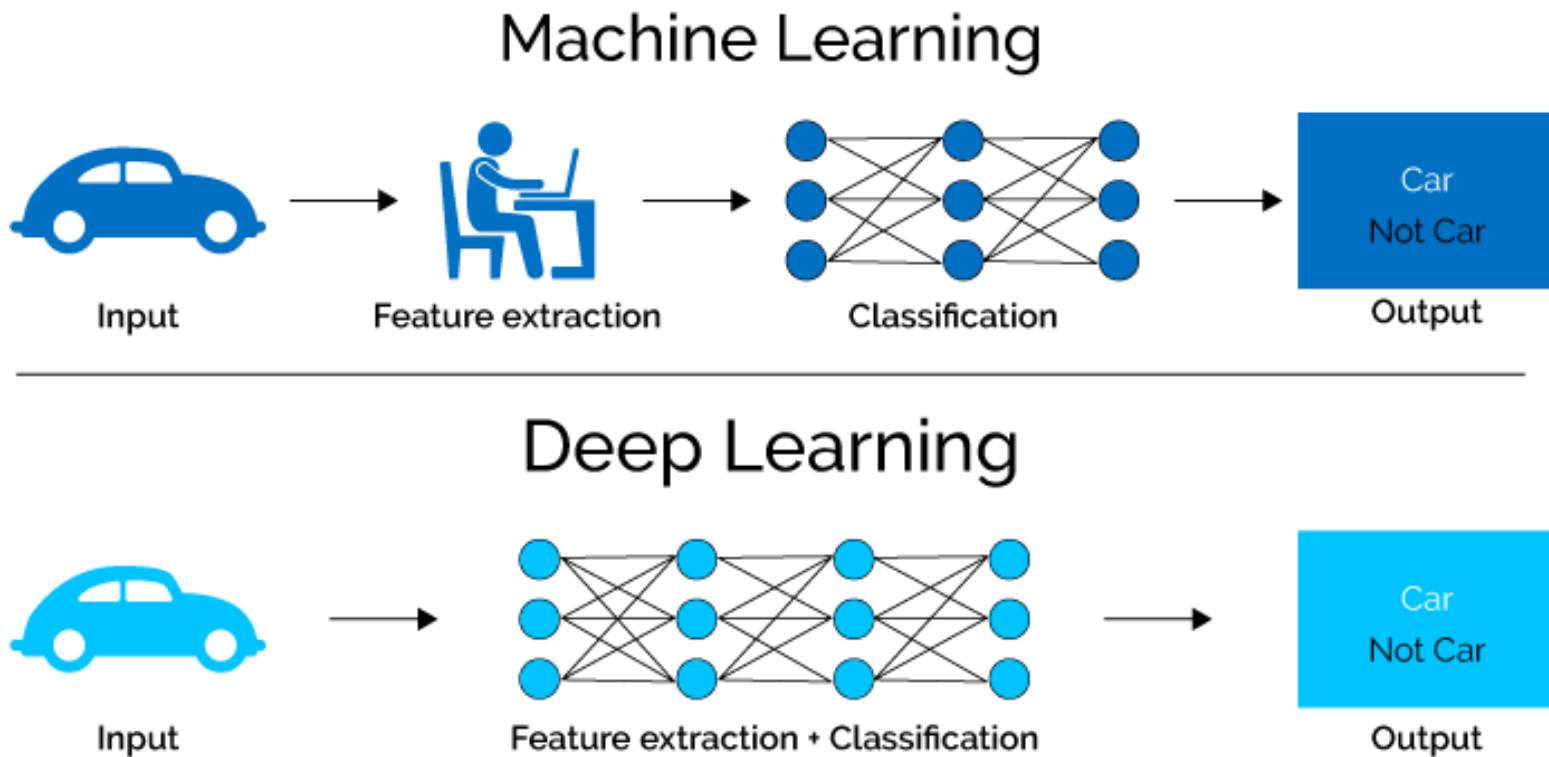
WHY DL: EXPLOITING LARGE DATASETS

Traditional ML algorithms (like decision trees, SVMs, or linear models) are effective for small to medium-sized datasets but struggle when the amount of data increases significantly. **Deep Learning** models, particularly deep neural networks, **thrive on large datasets**. In fact, their performance improves as the amount of data grows, making them suitable for tasks in domains with large datasets, like image recognition or natural language processing (NLP).



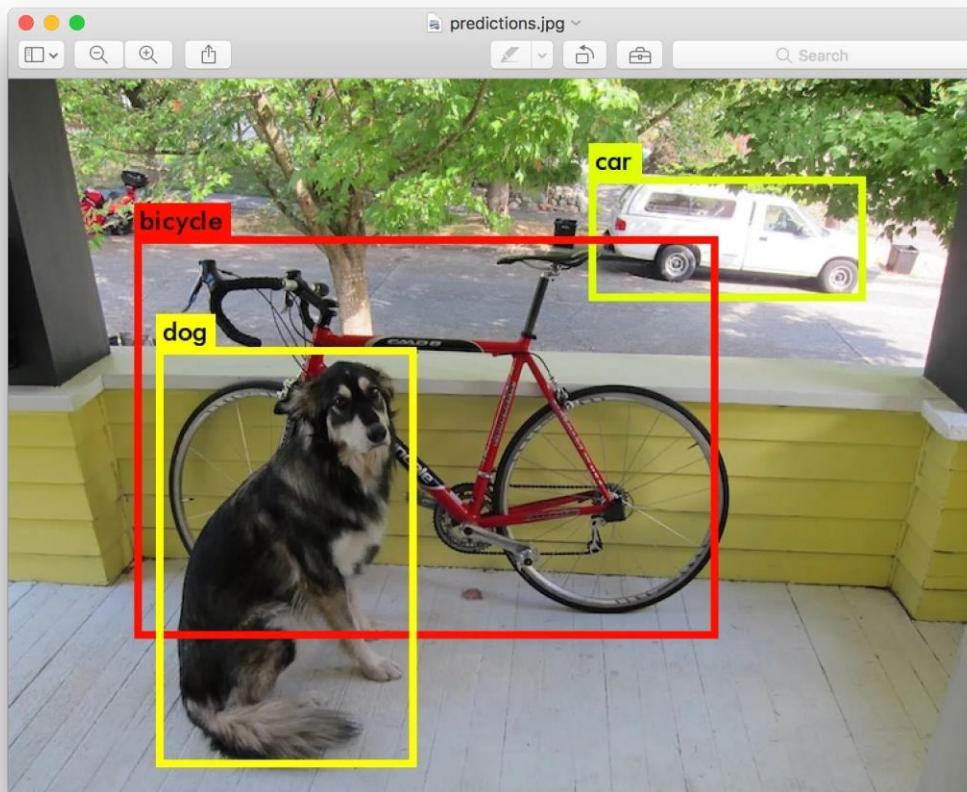
WHY DL: AUTOMATIC FEATURE EXTRACTION

Traditional ML often **requires manual feature engineering**, where experts need to extract relevant features from raw data for the model to work well. This process can be time-consuming and requires domain expertise. Deep Learning can **automatically learn features from raw data**, like pixels in images or words in text. Convolutional Neural Networks (CNNs), for example, learn different levels of abstraction (edges, textures, objects) directly from image data (in different layers of the network) without manual feature engineering.



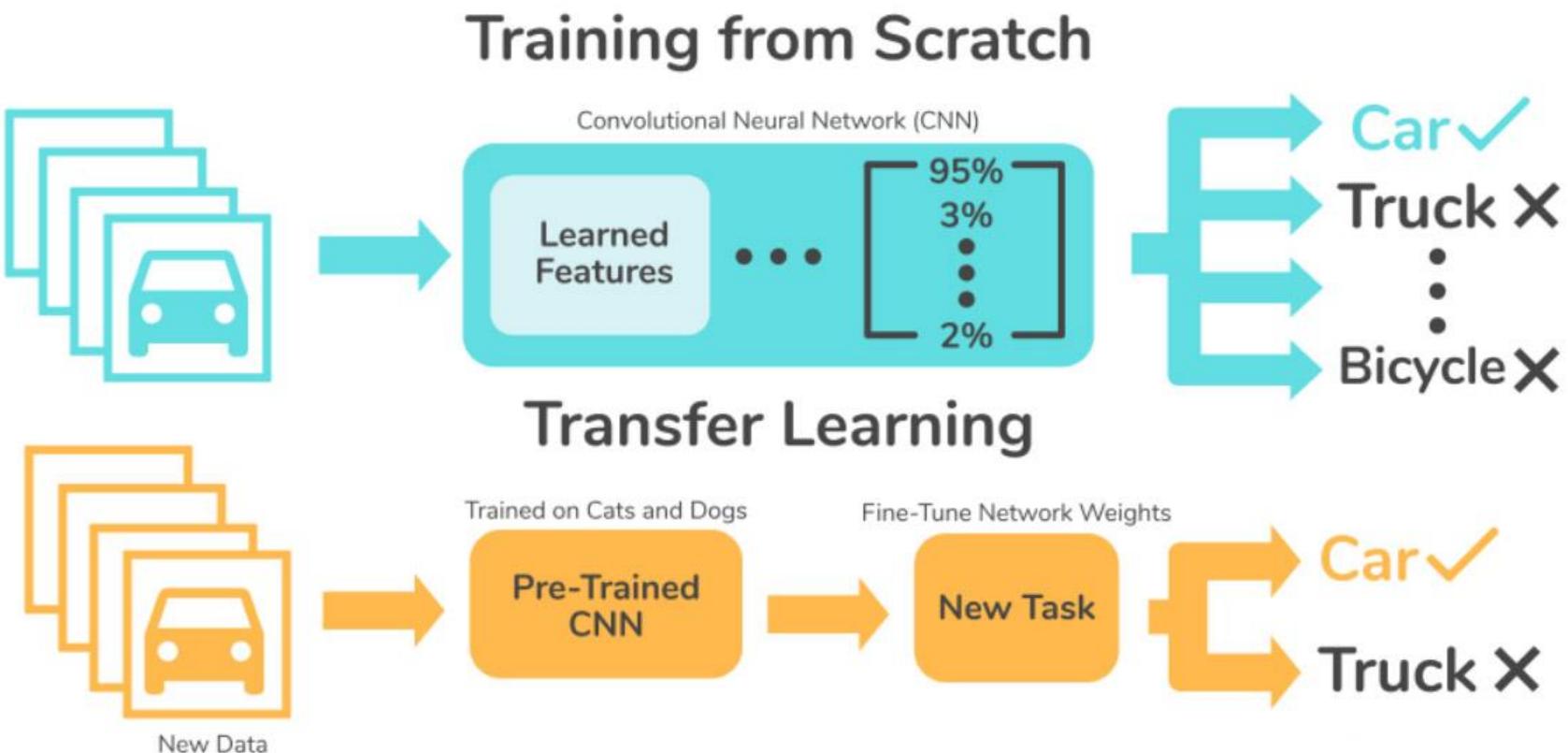
WHY DL: HANDLING UNSTRUCTURED DATA

Traditional ML is typically better suited for structured data (like tabular data in spreadsheets), but struggles with unstructured data (like images, audio, and text). **Deep Learning** excels at **handling unstructured data**, thanks to neural networks designed to process images (CNNs), sequences (Recurrent Neural Networks or RNNs), or both (transformers for NLP tasks).



WHY DL: TRANSFER LEARNING

Traditional ML models typically require extensive retraining when applied to new tasks or domains. Deep Learning offers techniques like **Transfer Learning**, where a pre-trained model on a large dataset (like ImageNet) can be fine-tuned for a specific task, saving time and computational resources.



DL IN PRACTICE



Source: CMU's Introduction to Machine Learning (10-601) Lectures

DL IN PRACTICE

Hey bro.. Can you generate an image for me that can be used in a presentation on Machine Learning?

Copilot

I'll try to create that.

1 of 30

"Machine Learning"

Designer Powered by DALL-E 3 14

My students tend to use Machine Learning to solve any kind of problem. In short, tell them when to use ML and when not to?

When to Use Machine Learning:

- **Complex Patterns:** When the problem involves detecting complex patterns, trends, or relationships in large datasets.
- **Unstructured Data:** When working with unstructured data like images, text, or audio.
- **Automation:** When manual rule-based systems are too rigid or impractical for the task.
- **Prediction:** When the goal is to make predictions based on historical data.

When Not to Use Machine Learning:

- **Simple Problems:** If the problem can be solved with basic rules or algorithms, ML may be overkill.
- **Limited Data:** When there isn't enough data to train a reliable model.
- **Interpretability Required:** When clear and understandable results are more important than complex predictions.
- **Resource Constraints:** If there are limitations in computational power, time, or expertise.

DL IN COMPUTER VISION

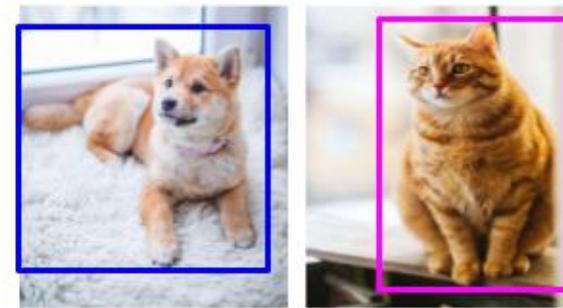
Classification



Dog

Cat

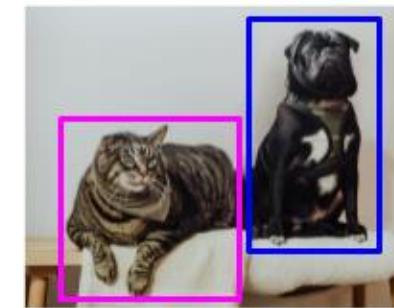
Classification + Localization



Dog

Cat

Object Detection



Cat, Dog

Semantic Segmentation



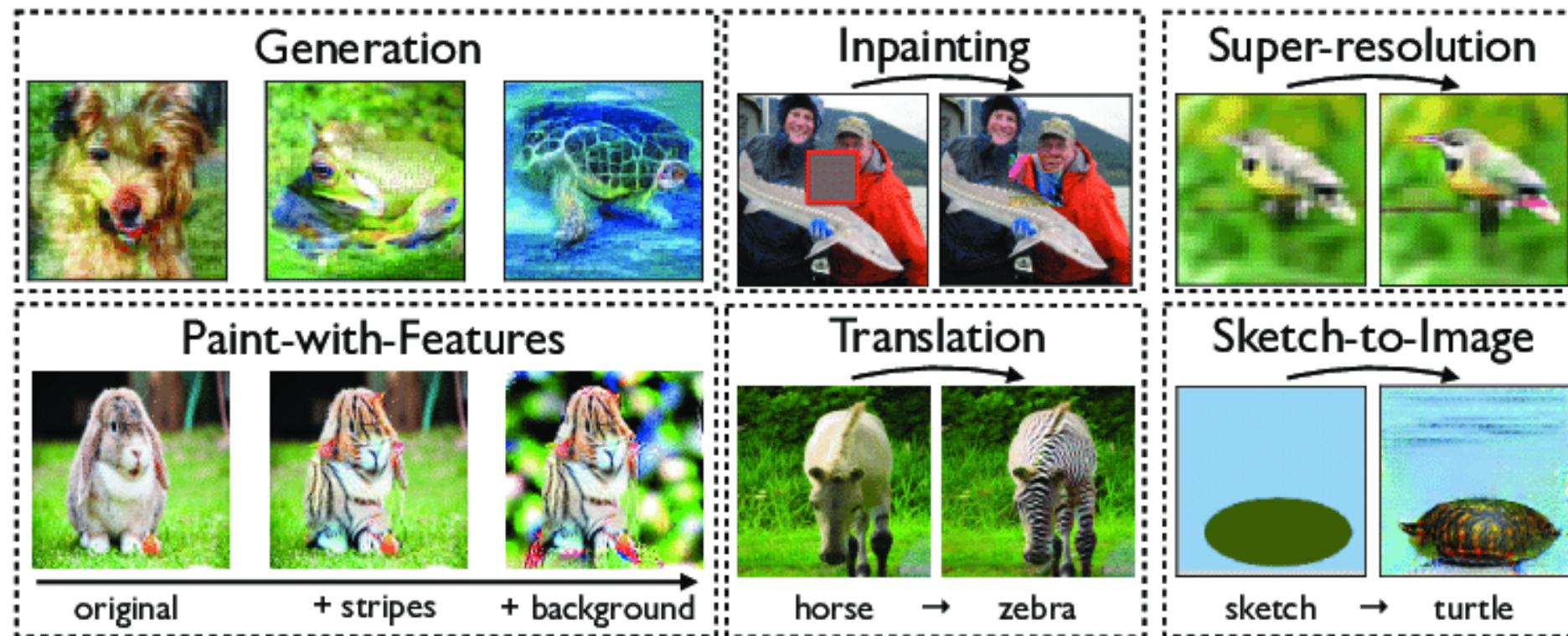
Grass, Cat

Instance Segmentation



Cat, Cat, Cat, Cat, Cat

DL IN COMPUTER VISION



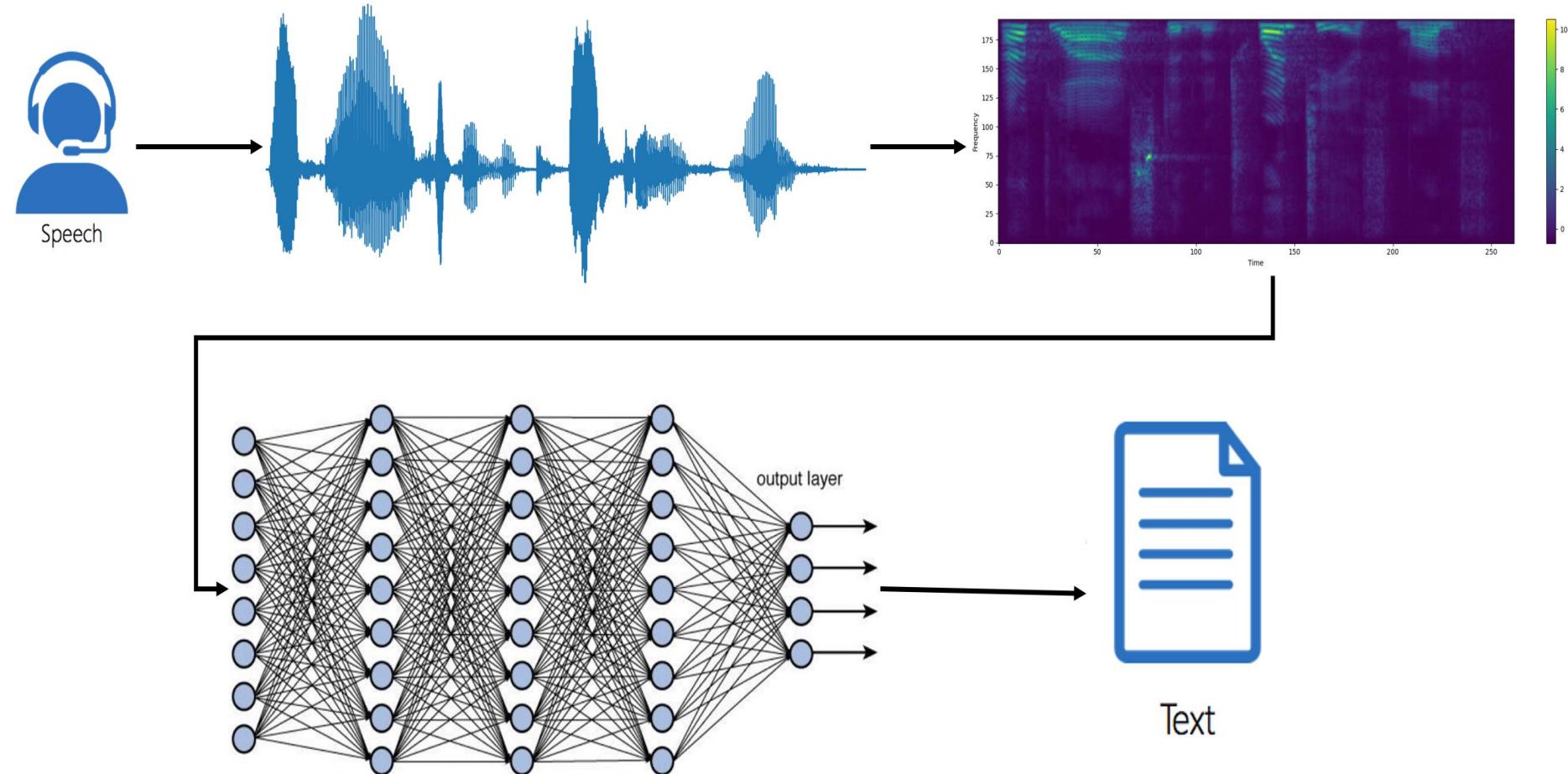
Source: S. Santurkar, A. Ilyas, D. Tsipras, L. Engstrom, B. Tran, and A. Madry, "Image synthesis with a single (robust) classifier," in Advances in Neural Information Processing Systems 33 (NeurIPS), Dec. 2019, pp. 1262–1273

DL IN NATURAL LANGUAGE PROCESSING

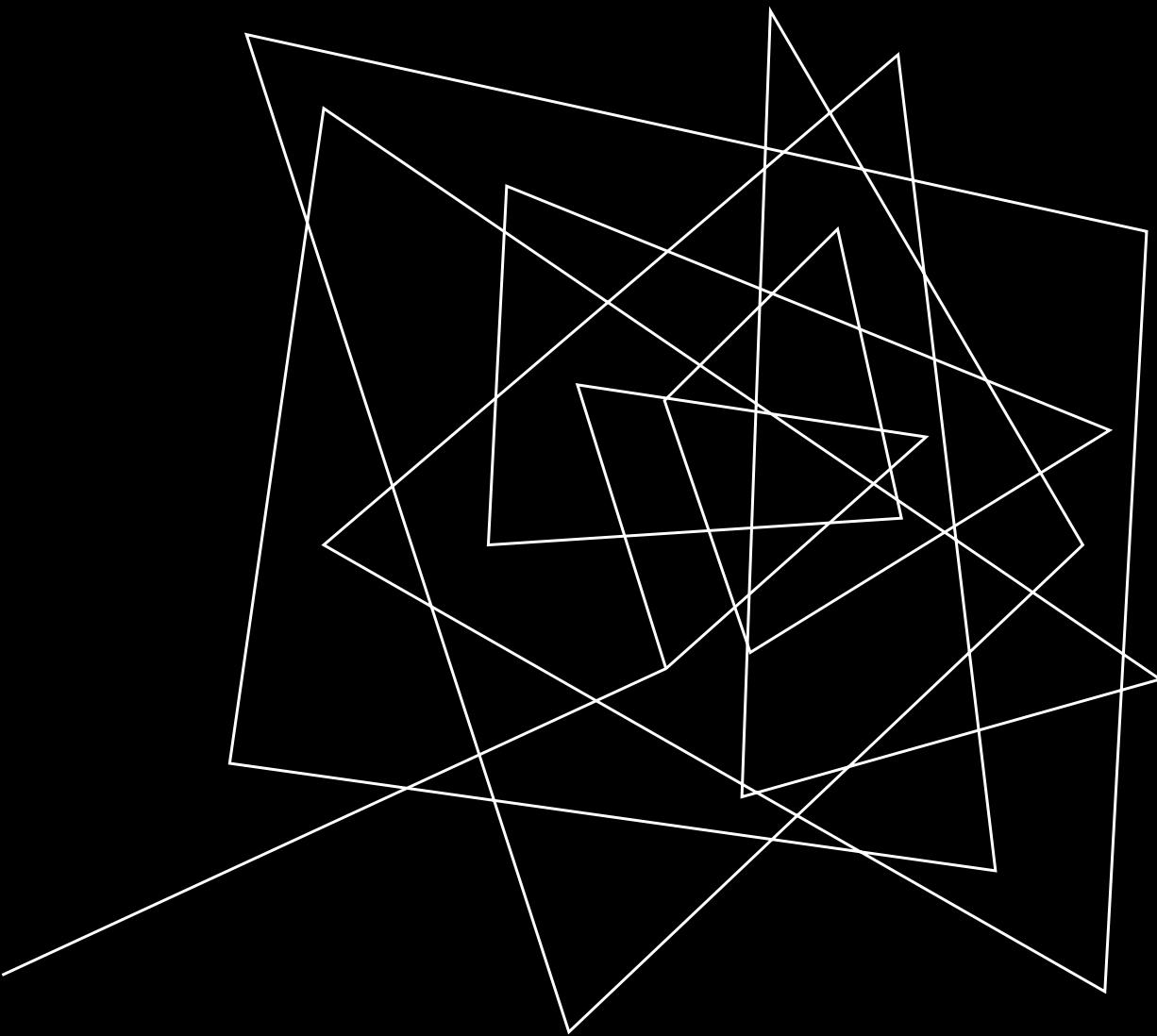
<p>Named Entity Recognition</p> <p>Frequency</p> <p>Latanoprost QHS OU Drug Name Route</p>	<p>Question-Answering and Summarization</p> <p>Q: Can COVID-19 affect the eyes?</p> <p>Source 1 Source 2 Source 3</p> <p>A: Yes, COVID can affect the eyes and cause...</p>	<p>Topic Modeling</p> <p>COVID-19 and the eye Machine Learning Health disparities</p>
<p>Word Embedding (i.e. Sentiment Analysis)</p> <p>Happy</p> <p>"Good outcomes" "Excellent bedside manner"</p> <p>Sad</p> <p>"Got a complication..." "Vision is worse" "I felt rushed"</p>	<p>Text Cleaning (i.e. Lemmatization, stemming)</p> <p>Foxes enjoy eating different foods.</p> <p>[‘Fox’, ‘enjoy’, ‘eat’, ‘differ’, ‘food’]</p>	<p>Relevance Ranking</p> <p>Query: AI systems in ophthalmology</p> <p>Results:</p> <ol style="list-style-type: none"> 1. Document 1 (score = 91.0) 2. Document 2 (score = 89.5) . . 100. Document 100 (score = 1.2)

Source: Chen, Jimmy & Baxter, Sally. (2022). Applications of natural language processing in ophthalmology: present and future. *Frontiers in Medicine*. 9. 906554. 10.3389/fmed.2022.906554.

DL IN SPEECH PROCESSING



Source: <https://pylessons.com/speech-recognition>



TYPES OF DL MODELS

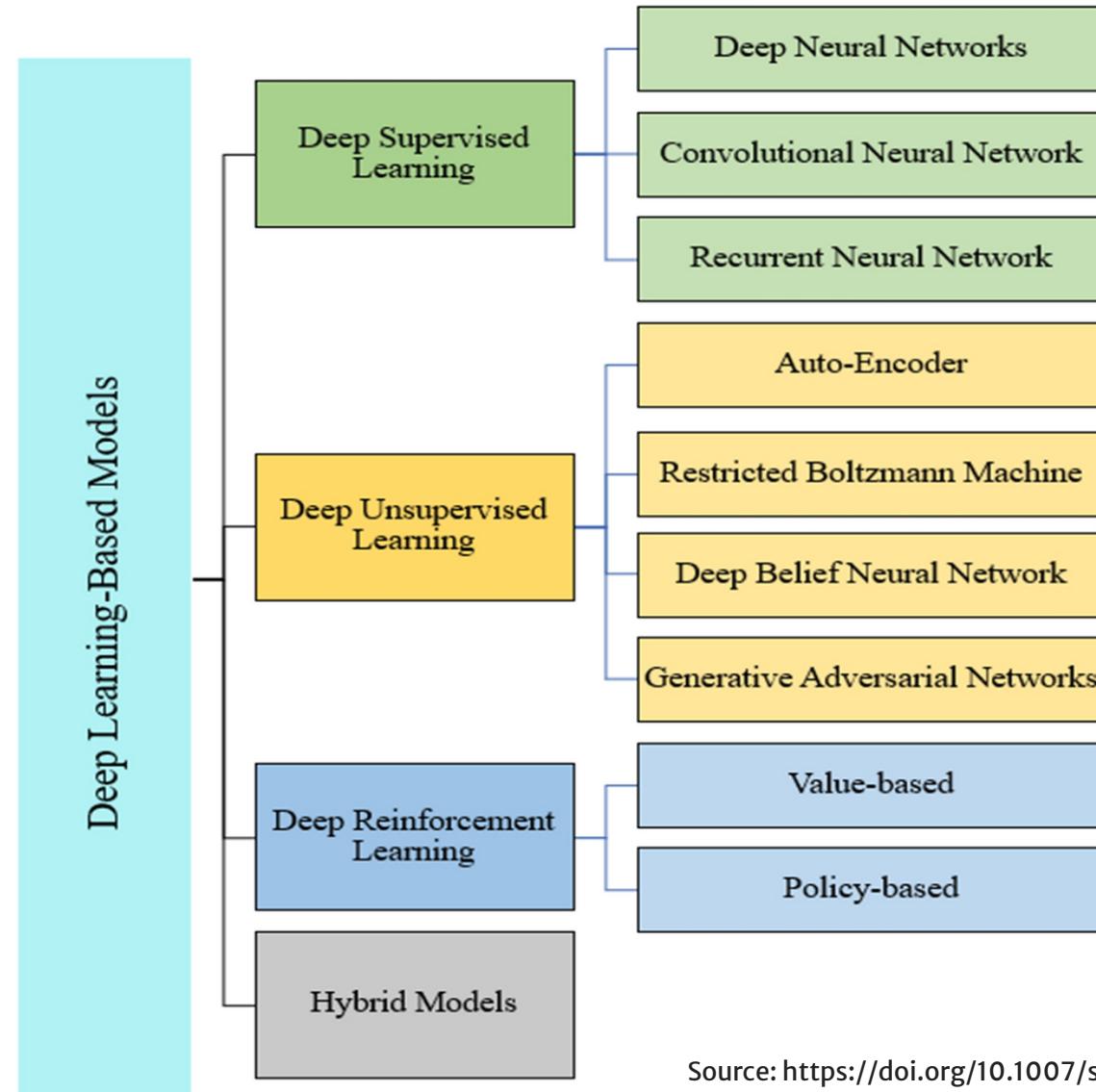
TYPES OF DL MODELS

At the core of deep learning are artificial neural networks (ANNs). The networks learn by adjusting the weights of the connections between the nodes in different layers. Based on the type of neural network there are several types of Deep Learning Models, such as:

- 1. Feedforward Neural Networks (FNN)**
- 2. Convolutional Neural Networks (CNN)**
- 3. Recurrent Neural Networks (RNN)**
- 4. Autoencoders**
- 5. Generative Adversarial Networks (GAN)**
- 6. Transformers**

There are other variants too, such as **Deep Reinforcement Learning** models, that are relevant in several different domains.

TYPES OF DL MODELS

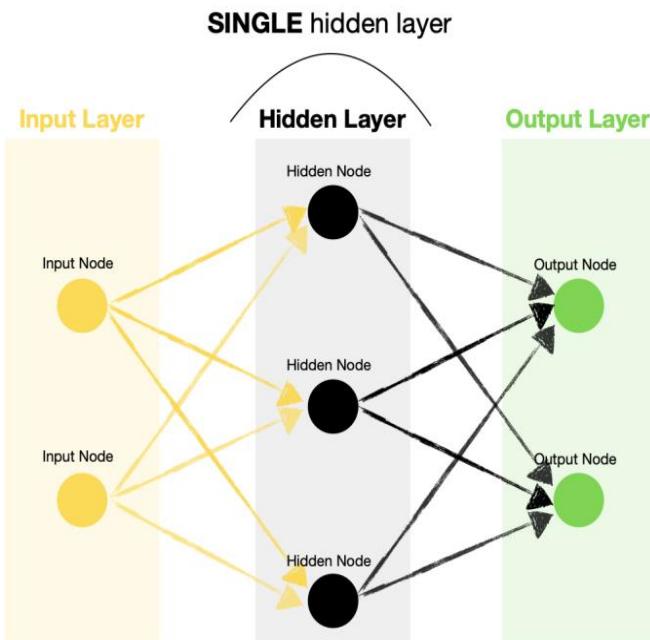


Source: <https://doi.org/10.1007/s00521-023-08957-4>

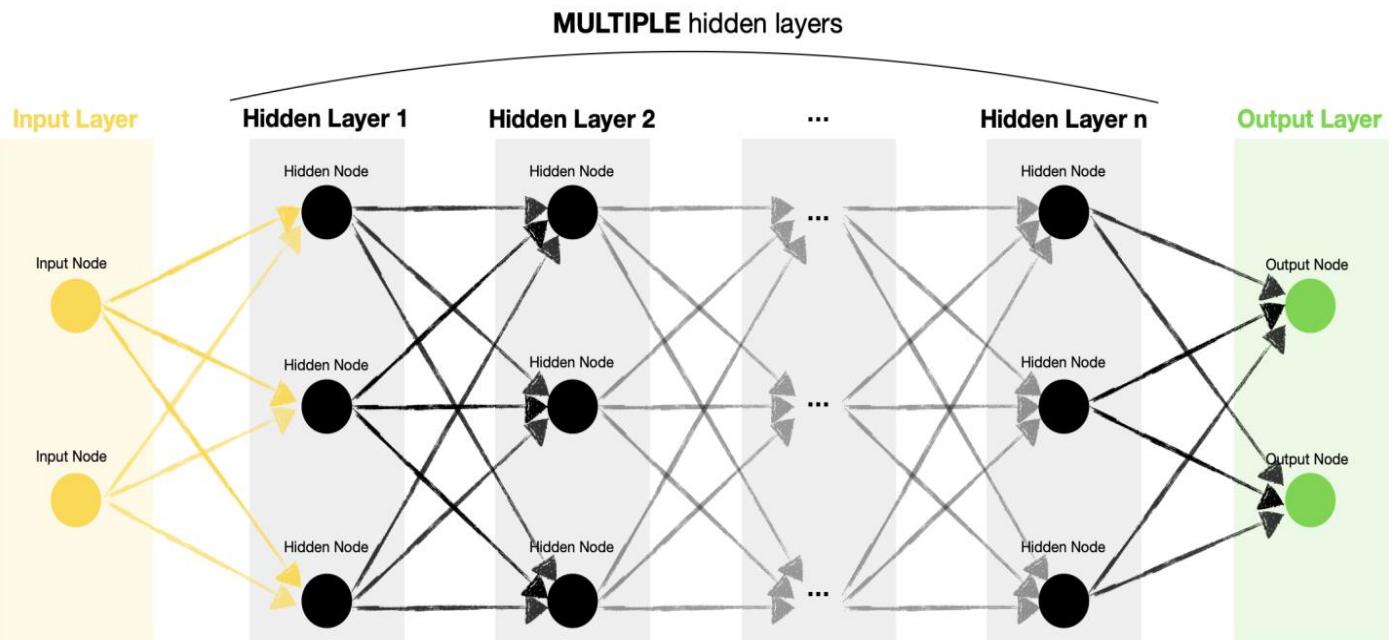
FEEDFORWARD NEURAL NETWORKS

Feedforward Neural Networks (FNN) are the simplest form of neural networks, where data moves in one direction, from input to output. They are often used for tasks such as High-dimensional tabular data classification, regression, and even image recognition.

Feed Forward Neural Network



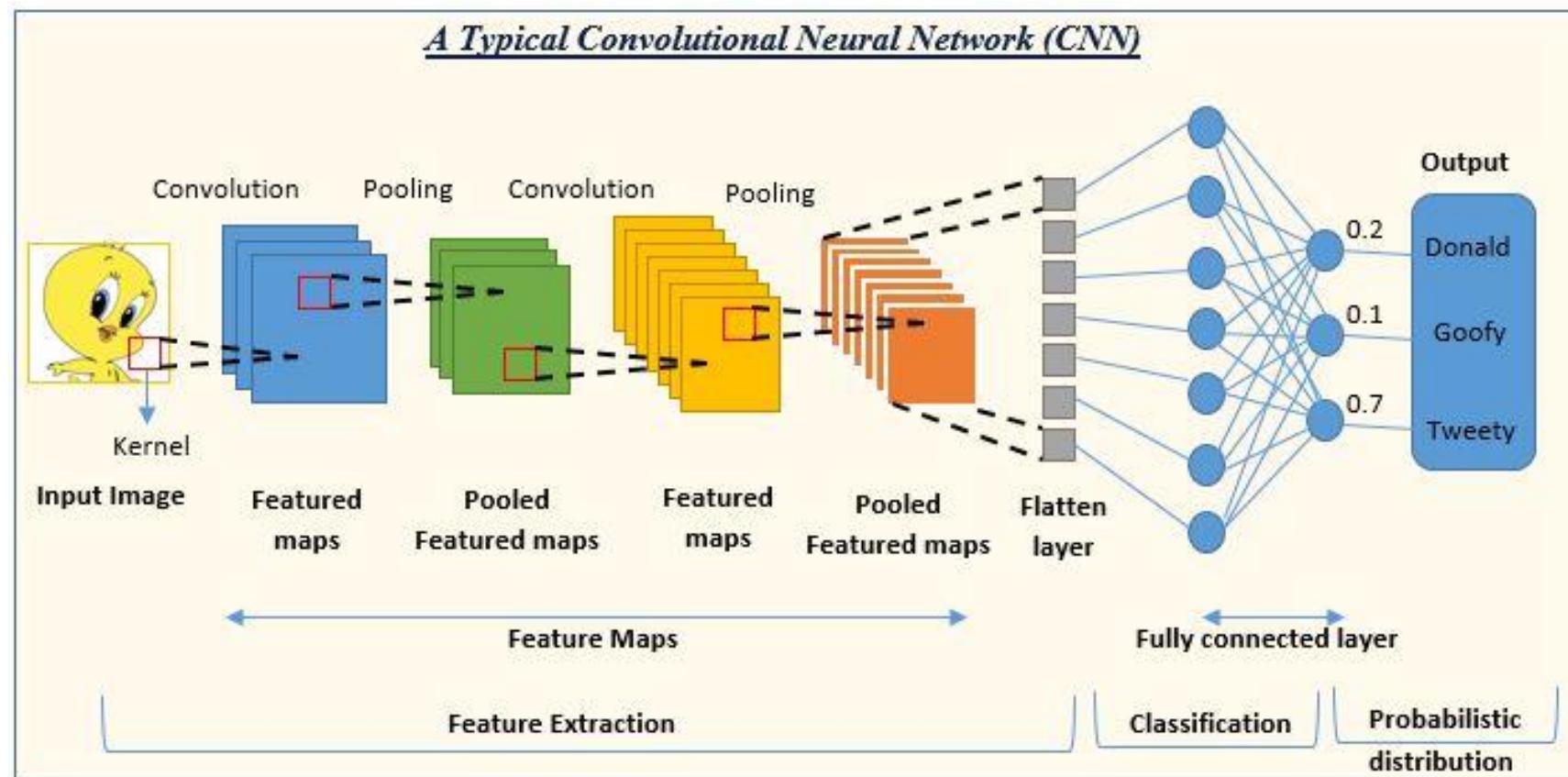
Deep Feed Forward Neural Network



Source: <https://www.linkedin.com/pulse/one-minute-overview-deep-feed-forward-neural-networks-saulius-dobilas/>

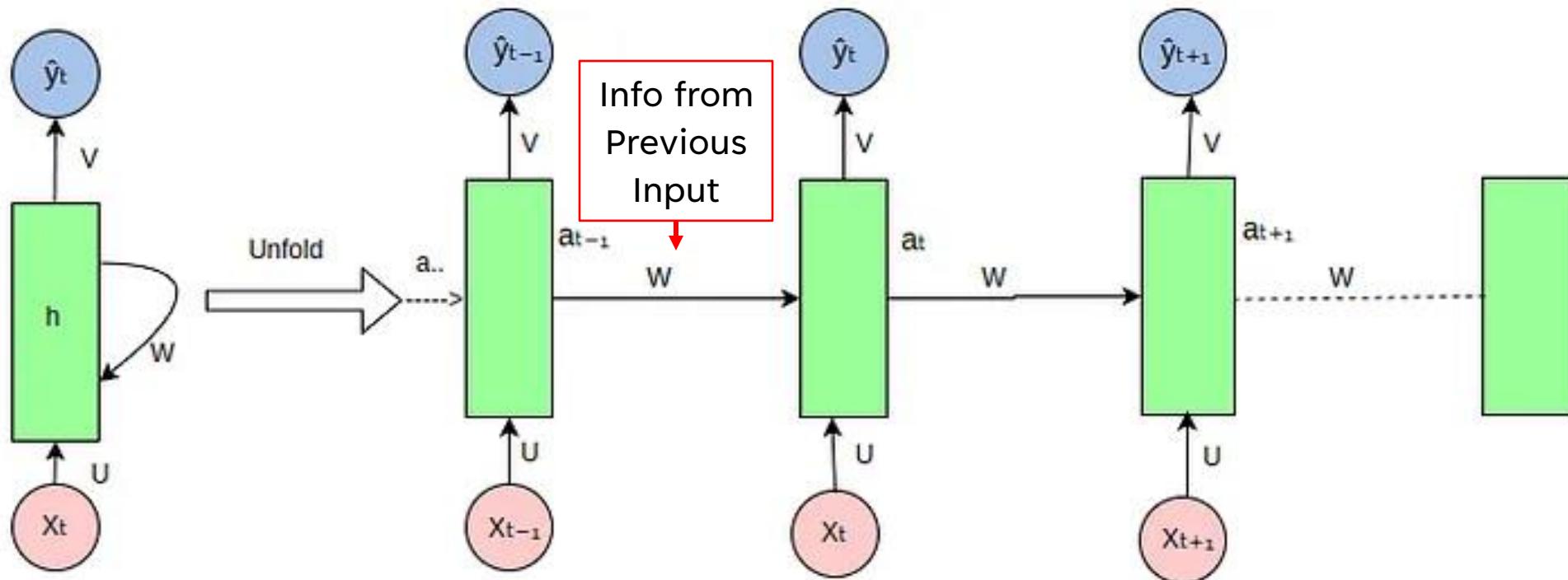
CONVOLUTIONAL NEURAL NETWORKS

CNNs (Convolutional Neural Networks) use convolutional layers to automatically detect features from images such as edges, textures, and shapes. These layers reduce the dimensionality of images without losing important information. CNNs are primarily used for image-related tasks such as image classification, object detection, and video analysis.



RECURRENT NEURAL NETWORKS

RNNs (Recurrent Neural Networks) are designed for sequential data, such as time series or natural language processing. RNNs maintain a hidden state that captures information from previous inputs, allowing them to learn from data sequences.

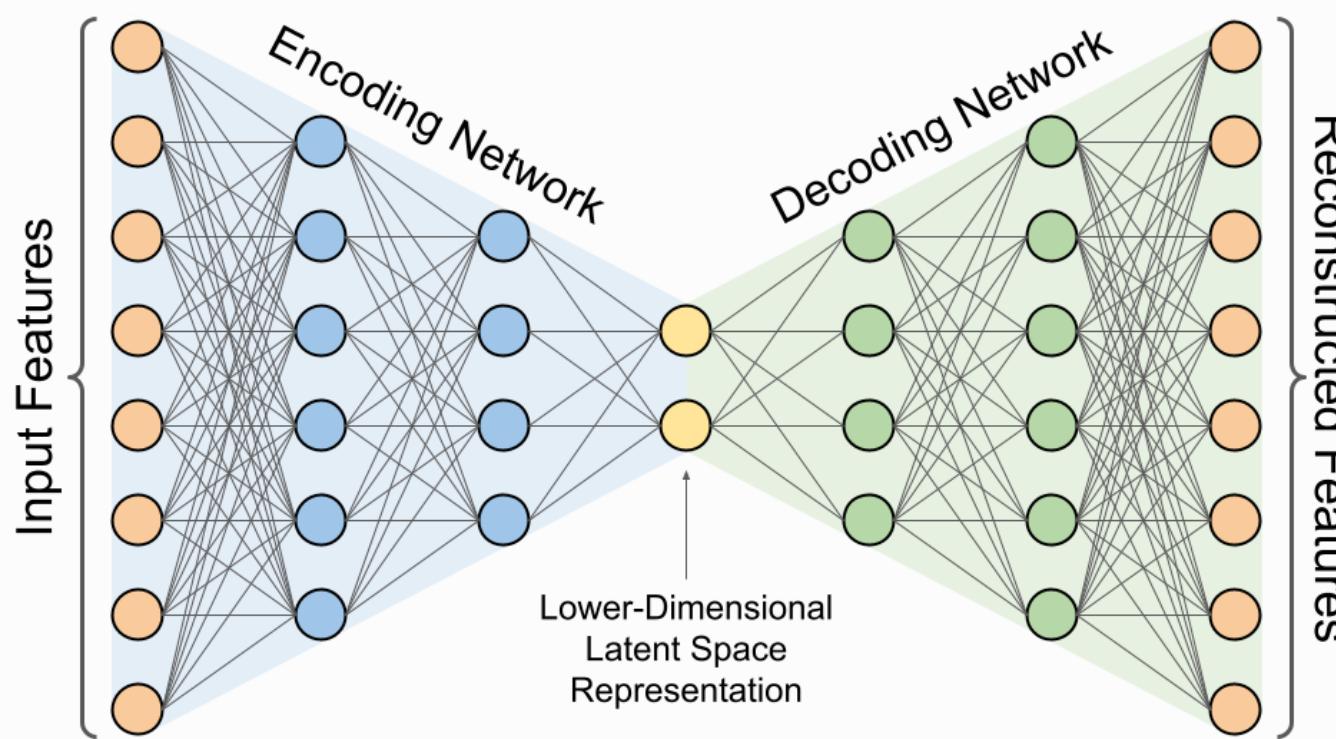


Source: <https://medium.com/@poudelsushmita878/recurrent-neural-network-rnn-architecture-explained-1d69560541ef>

A Typical RNN Architecture

AUTOENCODERS

Autoencoders are used for unsupervised learning tasks, such as dimensionality reduction or anomaly detection. They consist of two main parts: an **encoder** that compresses the input data and a **decoder** that attempts to reconstruct the original input from the compressed representation.

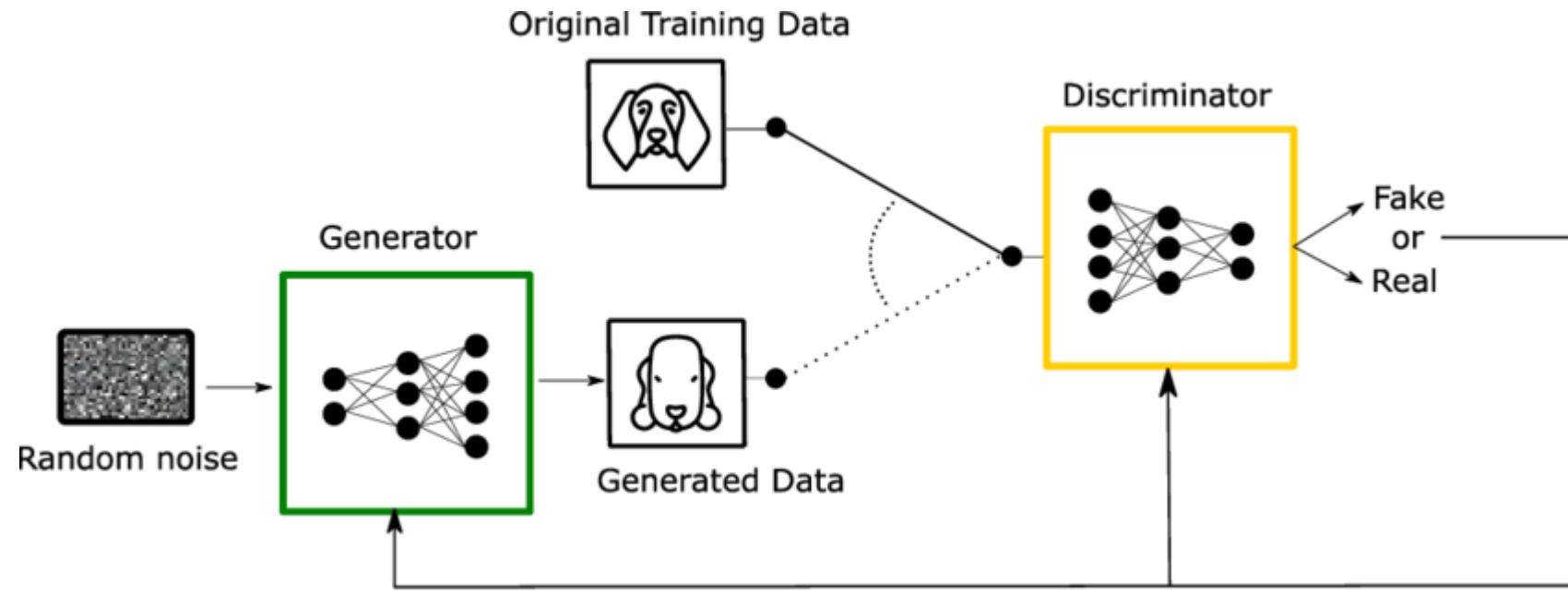


Source: <https://www.assemblyai.com/blog/introduction-to-variational-autoencoders-using-keras/>

Network Architecture for an Ordinary Autoencoder

GENERATIVE ADVERSARIAL NETWORK

GANs are a type of deep learning model used for generating new data similar to a given dataset. They consist of two networks, a **Generator** that creates fake data, and a **Discriminator** that evaluates the authenticity of the data. These two networks compete with each other, improving over time. GANs are used in applications like image generation, video generation, and deepfakes.

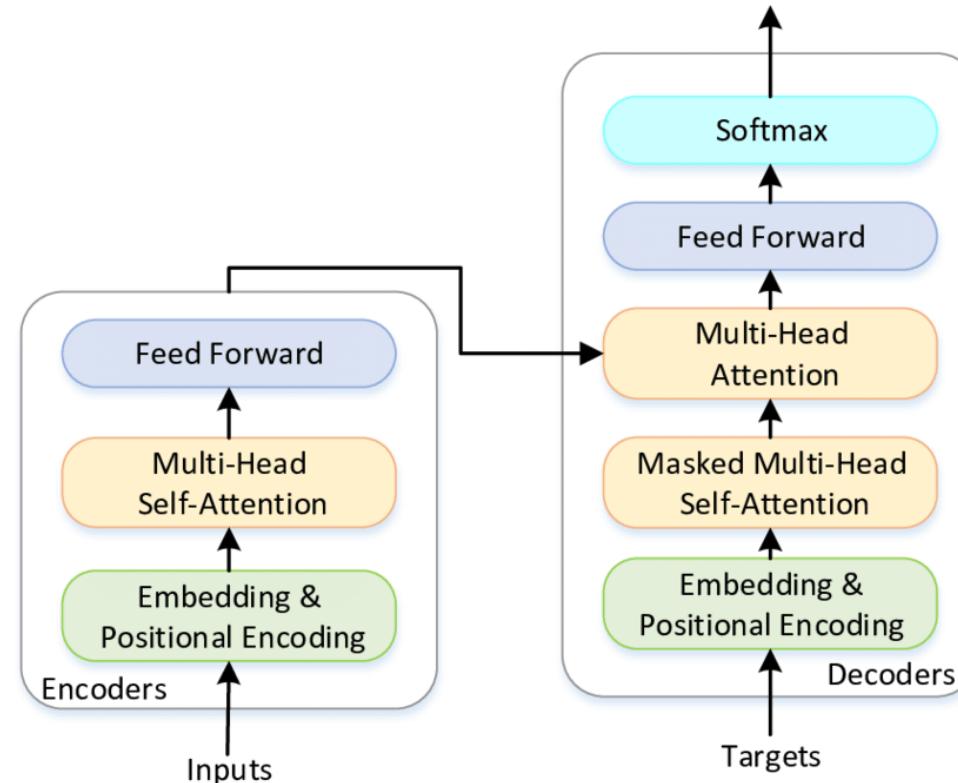


Source: <https://doi.org/10.3390/en12132485>

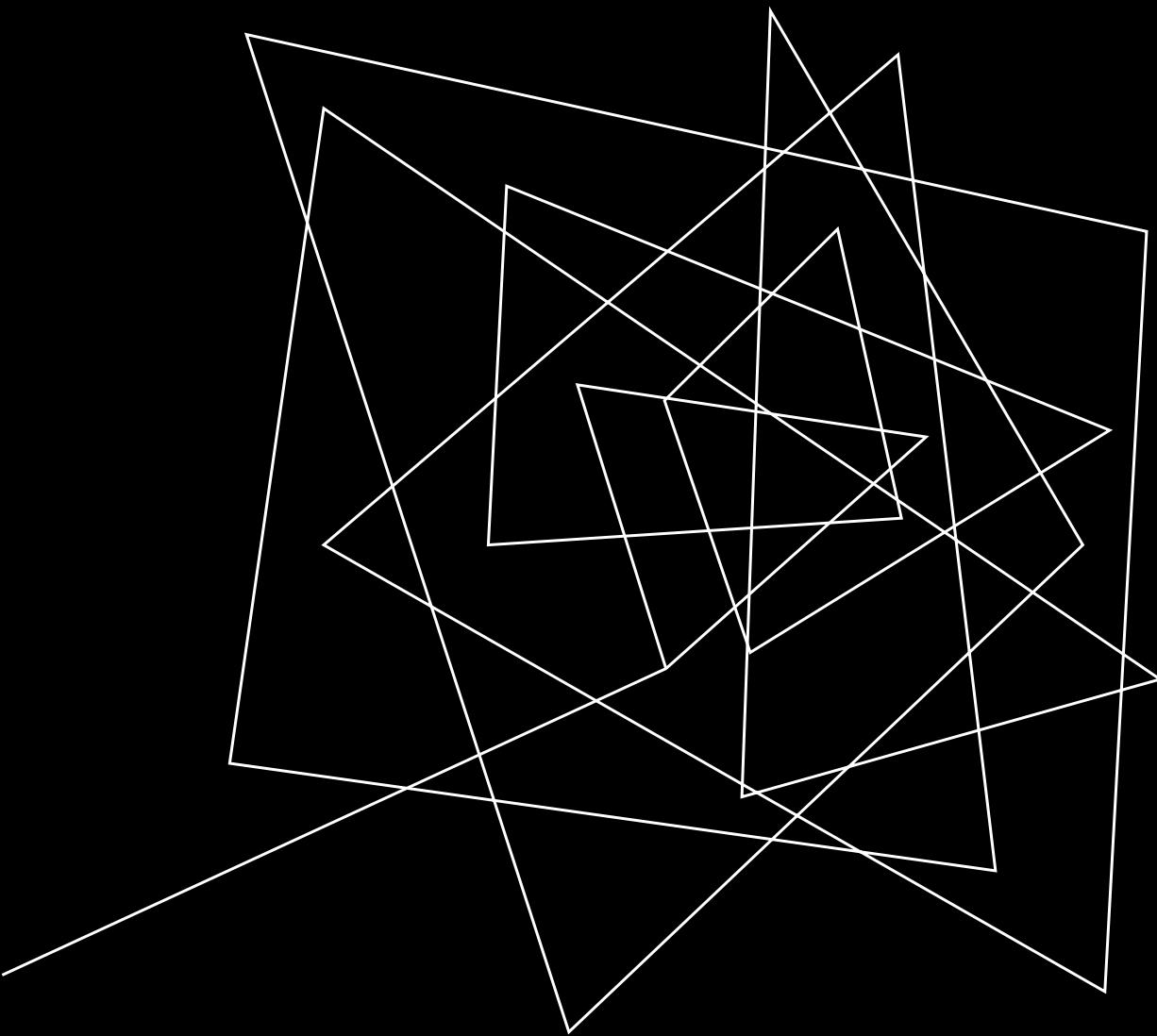
GAN Workflow

TRANSFORMERS

Transformers use self-attention mechanisms to focus on different parts of input sequences, making them highly effective for tasks such as language translation, text generation, and question-answering. They process data in parallel, making them highly efficient for tasks like natural language processing (NLP).



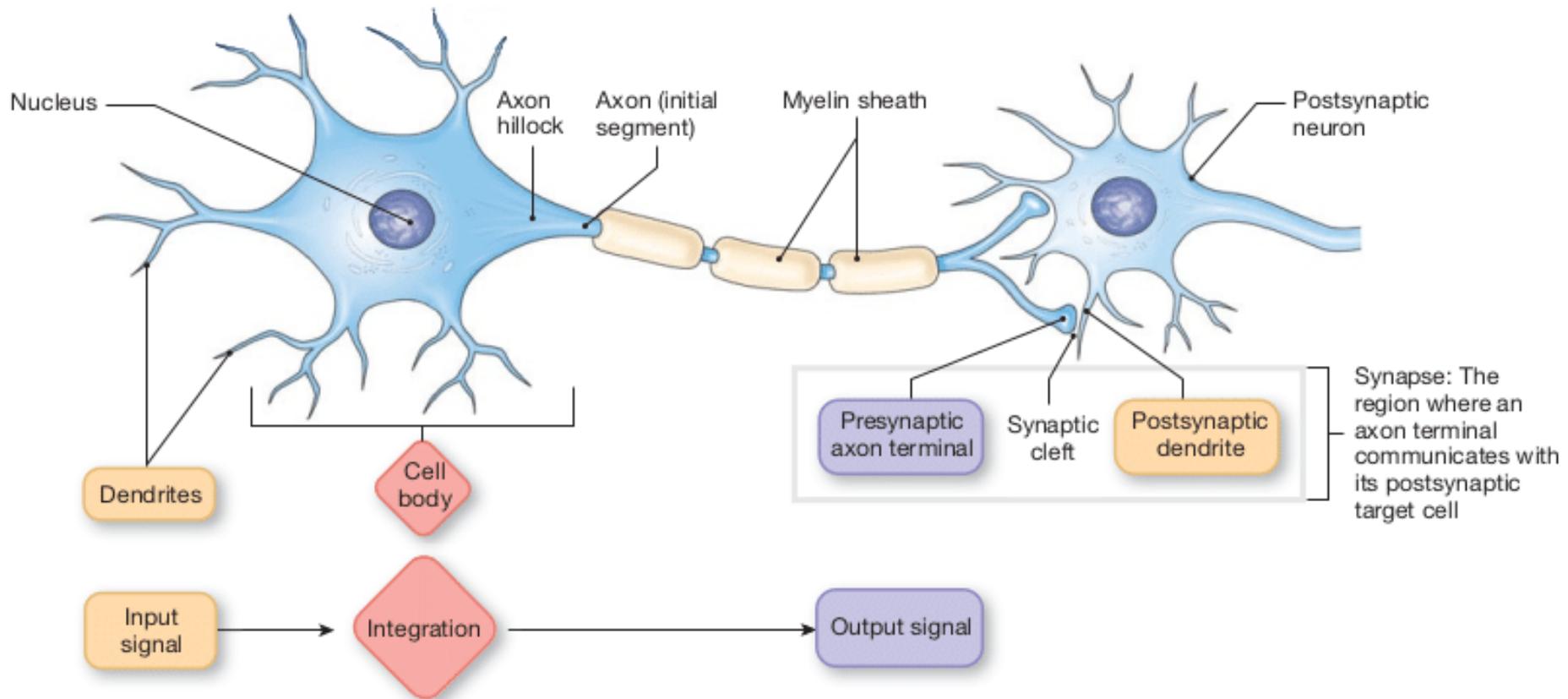
Simplified Diagram of a Transformer Model



HUMAN BRAIN &
NEURONS

HUMAN BRAIN: NEURONS

Neural Networks in machine learning were inspired by how **human brains** process information. While not an exact replica, certain key aspects of **biological neurons** have been mimicked to create artificial neural networks (ANNs).



Source: <https://doi.org/10.46298/mna.7284>

ARTIFICIAL NEURONS: THE MOTIVATION

In the context of neural networks, certain parts of the human brain's neurons serve as inspiration for their structure and functionality. In essence, neurons in artificial neural networks **mimic the communication and processing functions of biological neurons**, where dendrites, cell bodies, axons, and synapses work together to form complex connections and learning pathways. Here's how these parts relate to artificial neural networks:

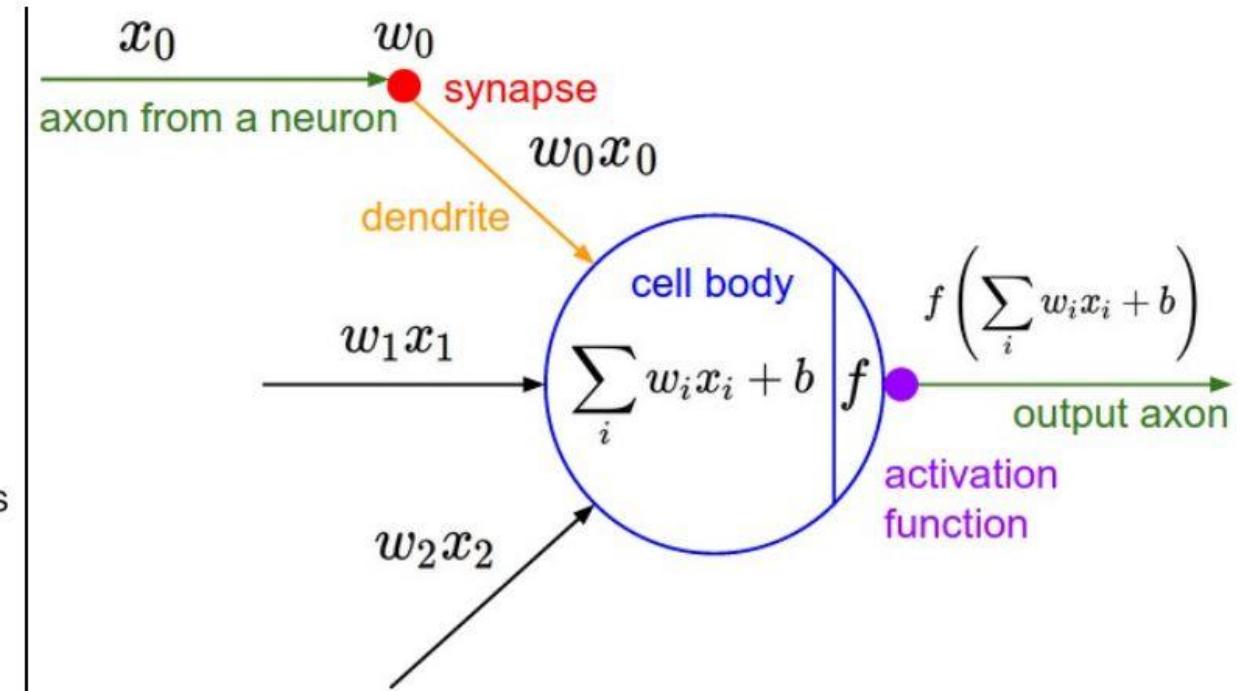
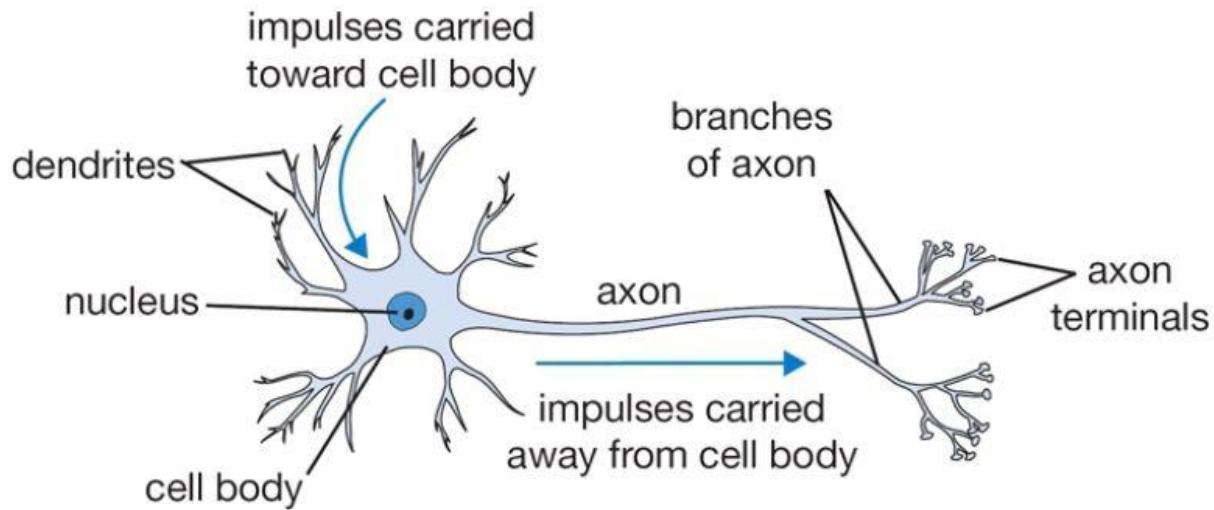
- **Dendrites:** Dendrites are the branched extensions of a neuron that receive signals from other neurons. Dendrites are analogous to the **input connections** in a neural network. They receive signals (data) from other neurons or input sources, similar to how input nodes in neural networks receive data.
- **Cell Body (Soma):** The **cell body** contains the nucleus and other organelles. It integrates the incoming signals and generates a response if the cumulative signal exceeds a certain threshold. The cell body is represented by the **neuron** in artificial neural networks. It sums the input signals (which are weighted), applies an activation function, and produces an output.

ARTIFICIAL NEURONS: THE MOTIVATION

- **Nucleus:** The **nucleus** contains the cell's genetic material and controls the neuron's activities. While not directly represented, the nucleus can be thought of as the **core logic** of a neuron, where the decision-making process occurs. In artificial neurons, this is akin to how weights and biases influence decision-making.
- **Axon:** The **axon** is a long, slender projection that transmits signals from the cell body to other neurons or muscles. The axon corresponds to **the output** of a neuron in an artificial network, which in turn acts as an **input** to the next layer of neurons. Once the neuron processes the input, it sends the output signal through the axon, analogous to forwarding the processed value to other neurons.
- **Synapse:** A **synapse** is the **junction between two neurons**. It allows signals to pass from one neuron to another, modulated by neurotransmitters. These connections can strengthen or weaken over time depending on how frequently they are used, a process known as synaptic plasticity. The synapse is represented by the **connection weights** between neurons. These weights determine the strength of the signal passed from one neuron to the next, similar to how synapses regulate the transmission strength in biological systems.

ARTIFICIAL NEURONS

In artificial neural networks, a "neuron" (also called a **node** or **unit**) receives inputs (which can be features from data), processes them using a mathematical function, and produces an output. Like biological neurons, artificial neurons are connected in layers, allowing them to process information in stages.



HUMAN BRAIN TO NEURAL NETWORKS

Apart from the structure of Neurons, Neural networks in machine learning were also inspired by how human brains process information. While not an exact replica, certain key aspects of this process was mimicked to create artificial neural networks (ANNs).

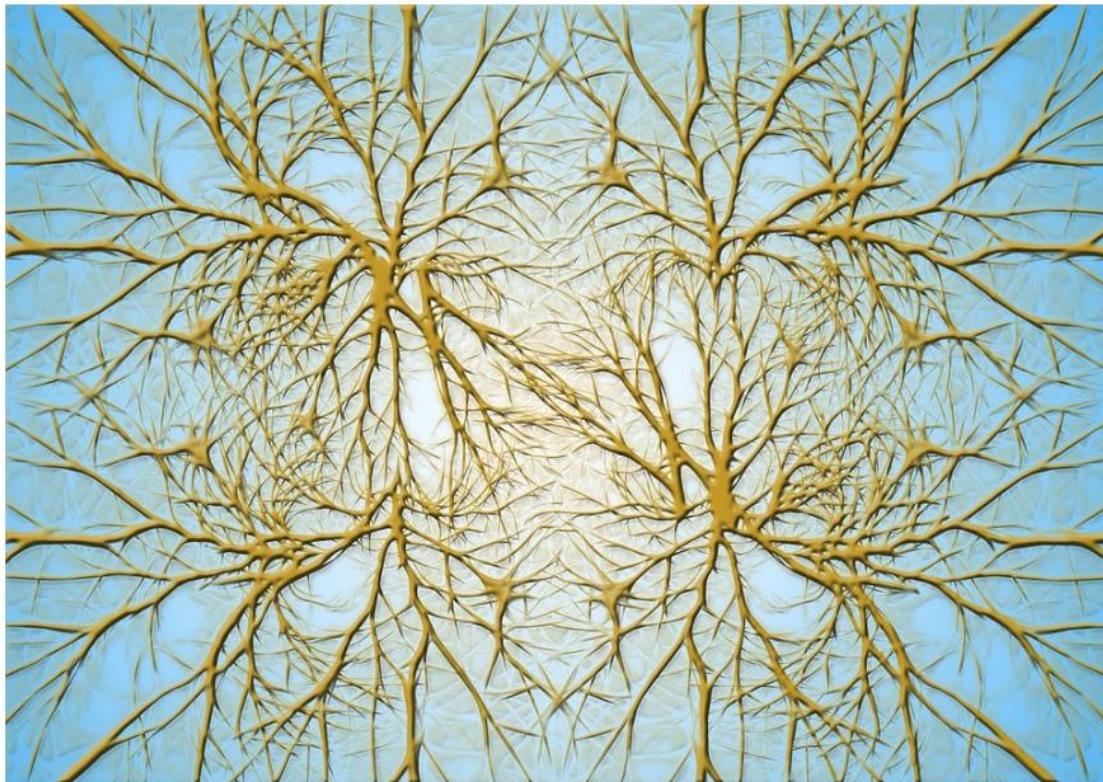
- **Layered Architecture (Hierarchical Learning):** The human brain processes information in a **hierarchical** manner. For example, in vision, the brain first detects edges, then shapes, and finally recognizes objects. This layered processing allows for more complex abstractions. Neural networks are also organized into **layers**. These multiple layers help the network learn increasingly abstract representations of the data, much like how the brain processes information hierarchically.
- **Activation Functions (Non-linear Behavior):** In biological neurons, the firing of a neuron is not a simple binary on/off process but involves complex, non-linear activation depending on the strength of the inputs. Artificial neurons also use activation functions to introduce non-linearity into the model. These non-linearities allow the network to learn and represent complex patterns beyond simple linear relationships.

HUMAN BRAIN TO NEURAL NETWORKS

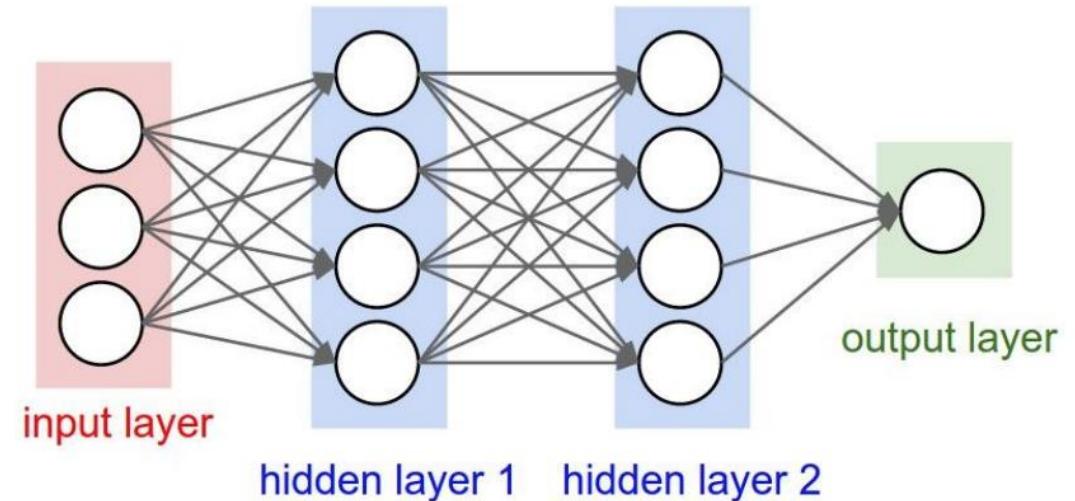
- **Learning through Experience:** The human brain learns through experience by **adjusting synaptic strengths** based on **sensory input and feedback**. When the brain recognizes patterns or makes decisions, neurons that are used frequently become more efficient, and those that aren't used weaken. Artificial neural networks also learn by adjusting weights based on training data. During the training process, the network is exposed to large amounts of data and uses a learning algorithm, such as backpropagation, to **update the weights**. In neural networks, the **backpropagation** algorithm works similarly to **feedback** in the brain. This allows the network to improve its ability to recognize patterns and make decisions over time.
- **Parallel Processing:** The brain can **process** multiple pieces of information **simultaneously** through interconnected networks of neurons. This allows humans to make sense of complex stimuli in real-time. Similarly, artificial neural networks **process information in parallel** across many layers of neurons. Each neuron works on its own small part of the input data simultaneously, contributing to the network's overall decision-making. This parallelism is key to the efficiency and power of neural networks in tasks like image recognition and speech processing.

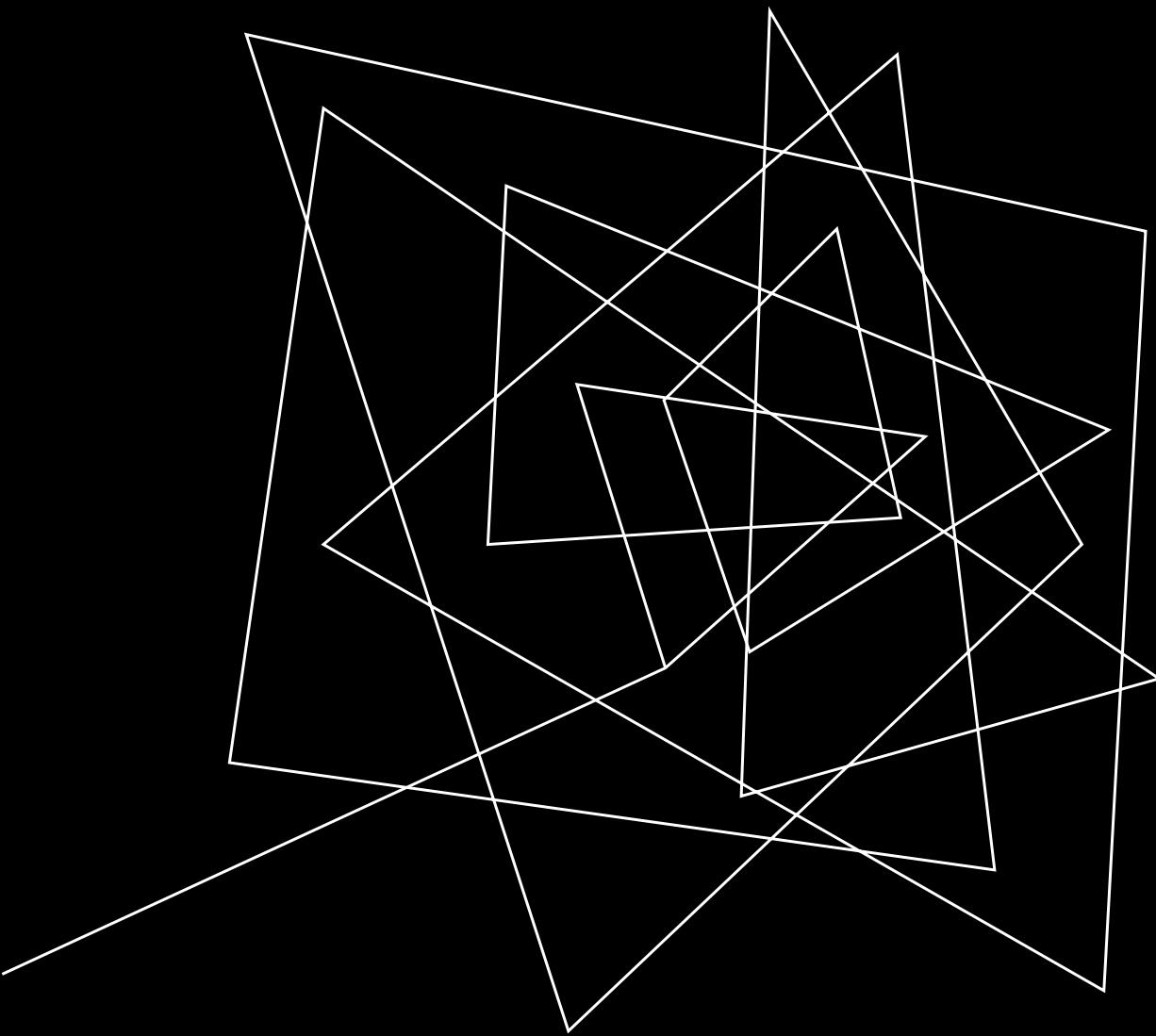
STILL JUST AN INSPIRATION!

Biological Neurons:
Complex connectivity patterns



Neurons in a neural network:
Organized into regular layers for
computational efficiency





NEURAL NETWORKS

NEURAL NETWORKS

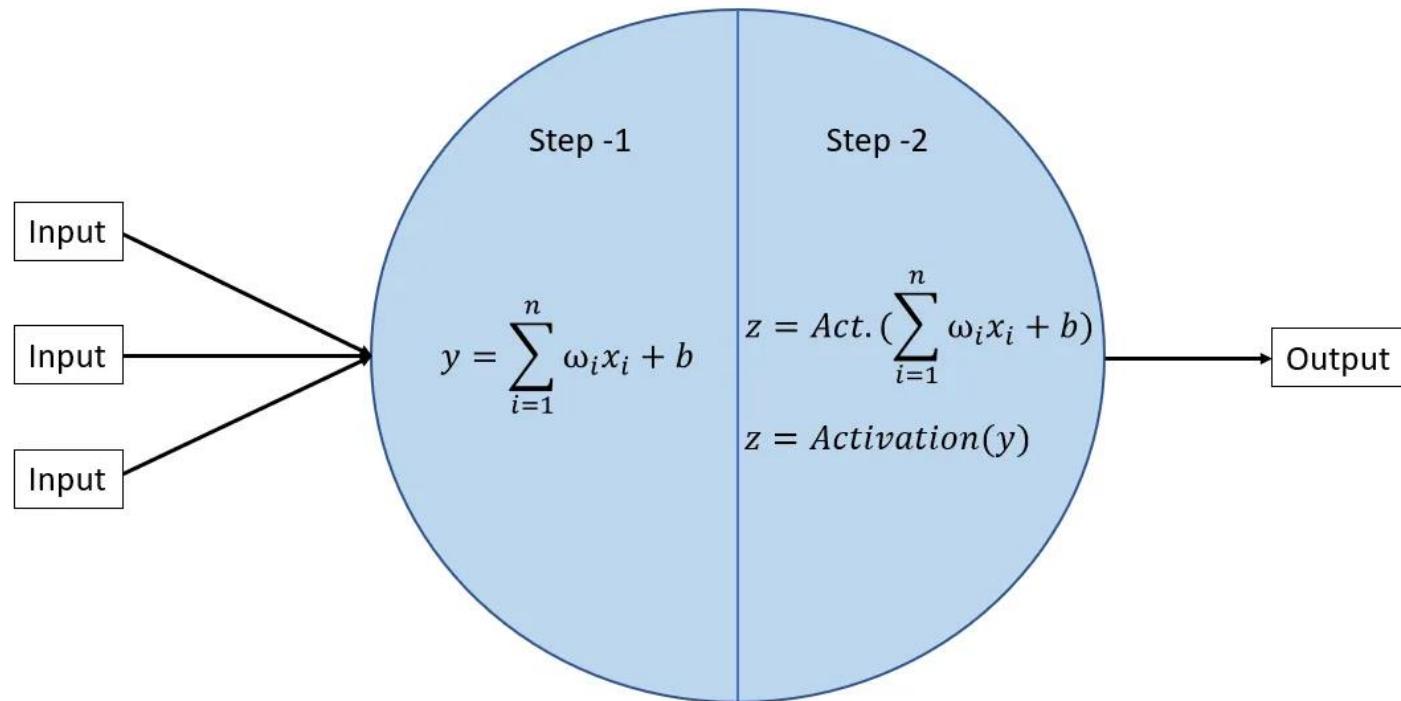
Neural Networks (NNs) are computational **models** inspired by the human brain's structure and functioning, which form the backbone of many machine learning (ML) and artificial intelligence (AI) applications. At the core, they consist of interconnected **layers of nodes** (also called **Neurons** or **Perceptrons**) that work together to recognize patterns and make decisions based on data. Neural networks are particularly effective at handling complex tasks such as image recognition, speech processing, and natural language understanding.

A **Neural Network Structure** consists of **three main types of layers**:

- ❑ **Input Layer:** The input layer receives raw data (**features**) from the outside world.
- ❑ **Hidden Layers:** These layers process the input data by applying mathematical transformations. Neural networks can have multiple hidden layers, which gives them the power to model complex relationships. Networks with more than one hidden layer are called deep neural networks.
- ❑ **Output Layer:** This layer produces the final result (**predictions**) of the network.

NEURON/UNIT STRUCTURE

A **Neuron** (or **Unit**) in a neural network is the fundamental building block that **processes input data**. It mimics a biological neuron and performs the following operations: **Taking an Input → Applying Weights → Computing Weighted Sum → Activation → Generating an Output**. In essence, a neuron processes information and helps in decision-making as part of the neural network's learning process, much like the nucleus in Biological Neurons.



Source: <https://medium.com/analytics-vidhya/what-do-you-mean-by-forward-propagation-in-ann-9a89c80dac1b>

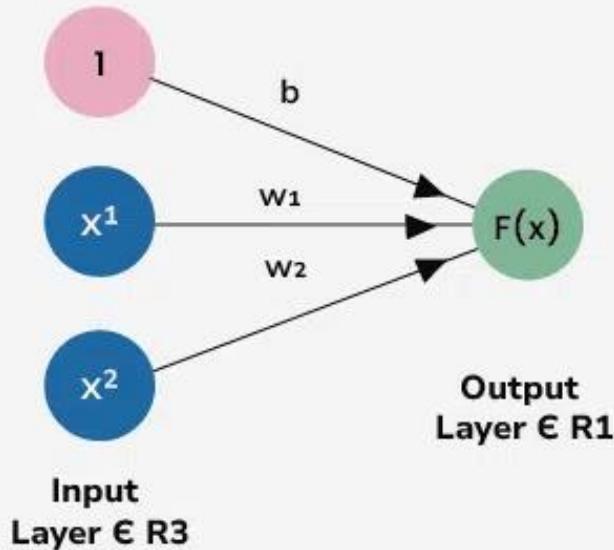
NEURAL NETWORKS

Let us clear out some facts about Neurons and Neural Networks first:

- **Neurons** are **arranged in layers** and layers are arranged **sequentially**.
- Neurons within the same layer do not transfer information with each other.
- All data/info enter via the **input layer** and information/output goes out by the **output layer**.
- Neurons in layer “**L**” are connected to neurons in layer “**L+1**”.
- Every **interconnection** in the neural network has a **weight associated** with it, and every **neuron** has a **bias associated** with it.
- All Neurons in a particular layer use the **same activation function**.

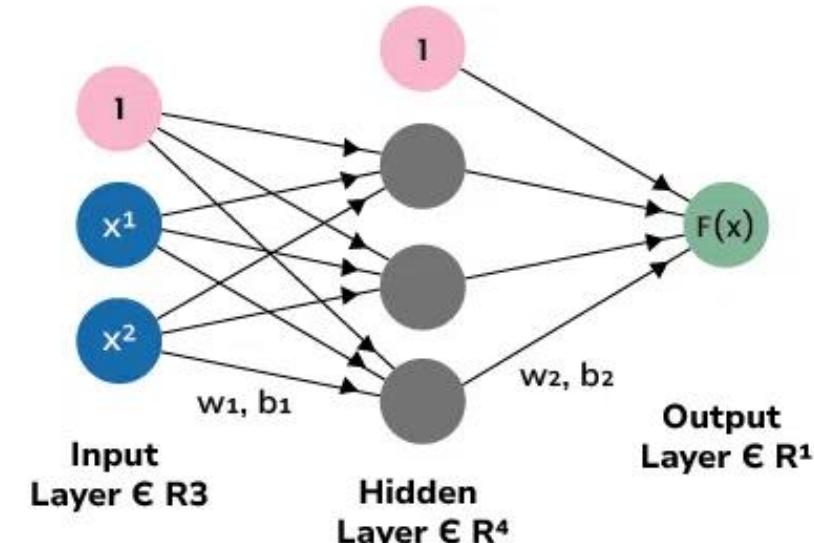
LINEAR REGRESSION vs NEURAL NETWORKS

Linear Regression



$$F(x) = w^T x + b$$

Neural Networks



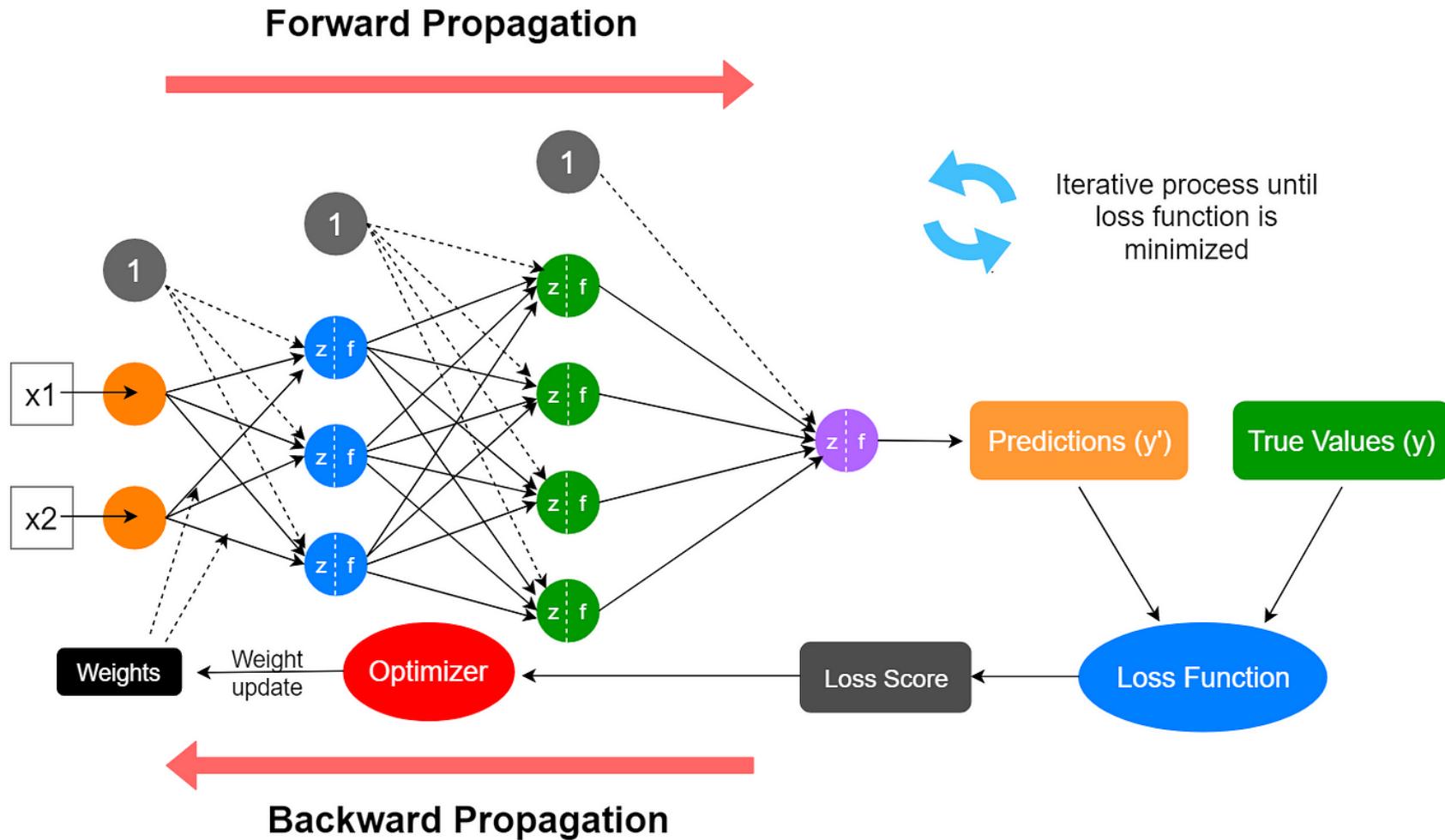
$$F(x) = w_2^T \sigma(w_1^T x + b_1) + b_2$$

Source: <https://www.geeksforgeeks.org/linear-regression-vs-neural-networks-understanding-key-differences/>

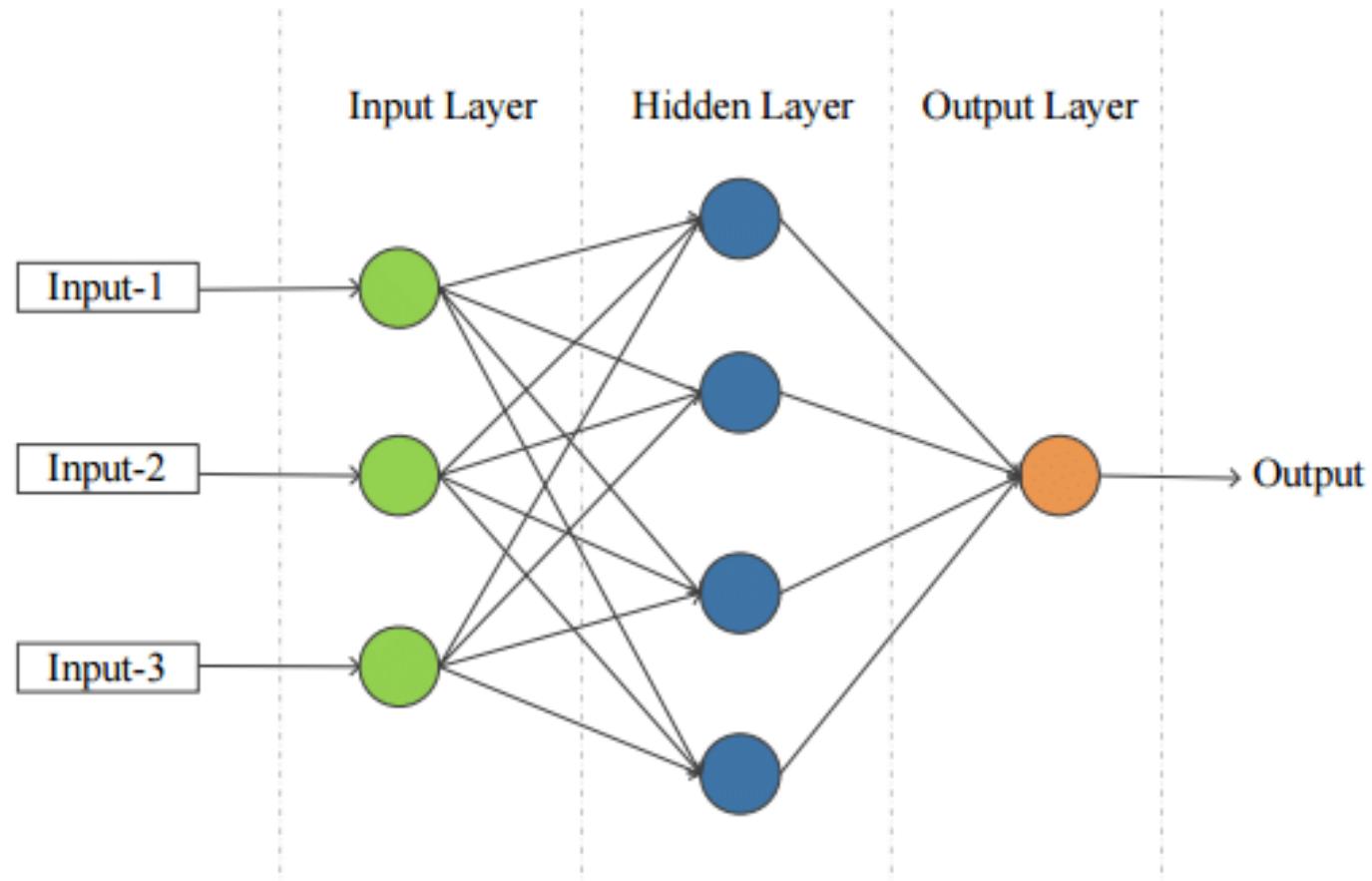
Linear Regression itself can be Portrayed as a Single-layer Neural Network

NEURAL NETWORKS

As we saw in the previous slide, **Neural Networks** apply functions over the functions of input features in multiple layers. The diagram below summarizes the **learning process of NNs**.

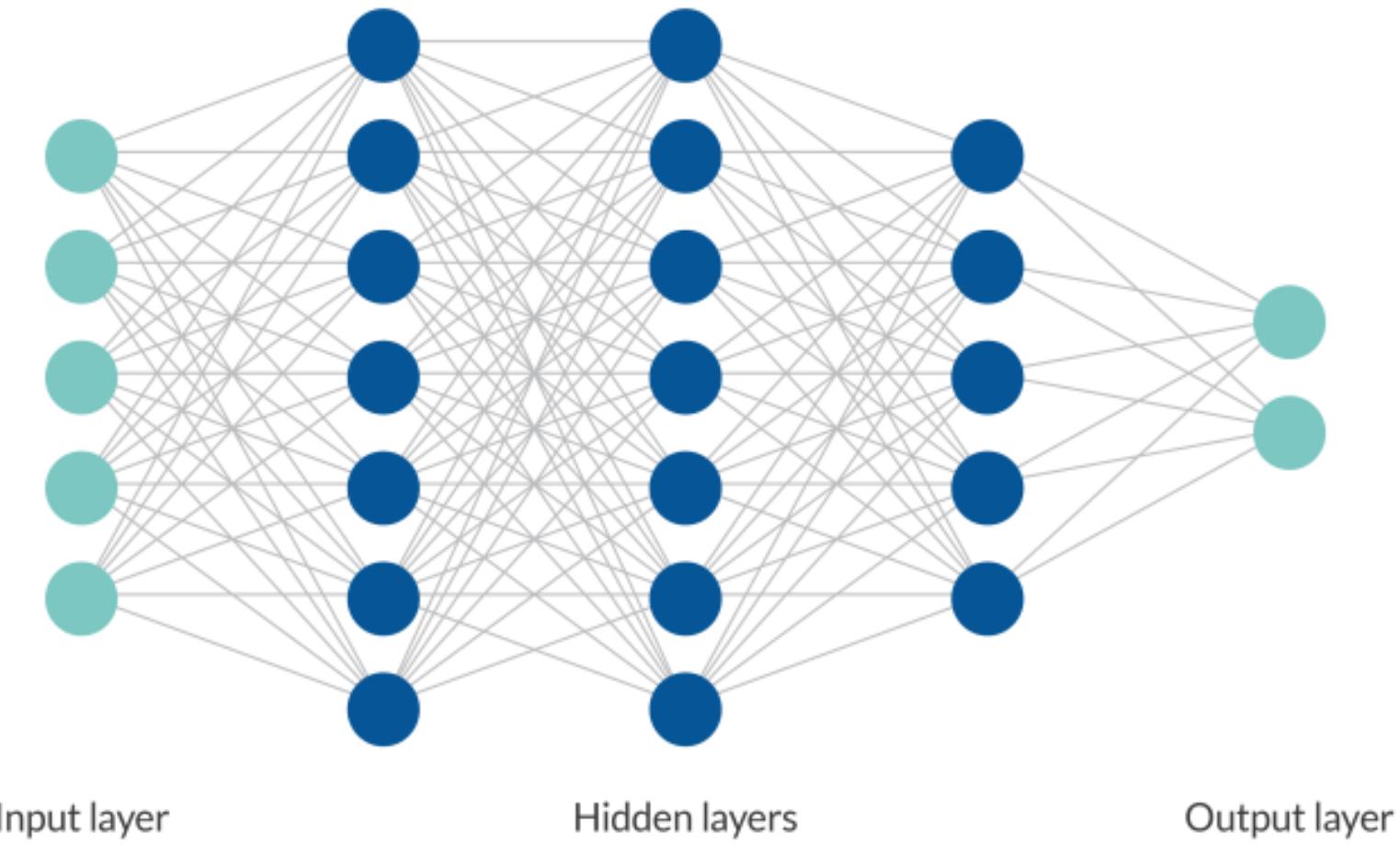


SHALLOW NEURAL NETWORKS

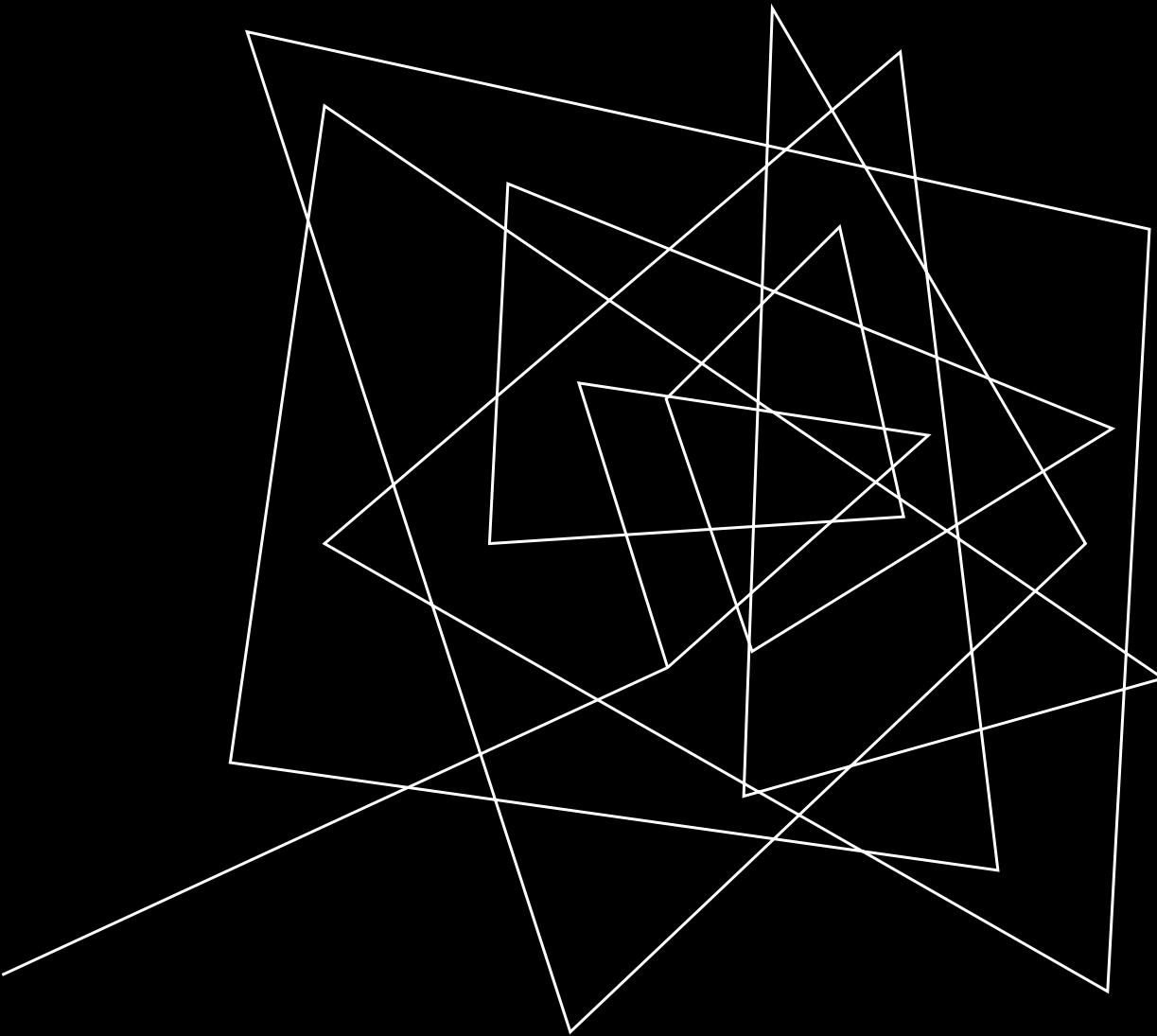


A Feedforward Neural Network with One Hidden Layer

DEEP NEURAL NETWORKS



A Deep Neural Network with Three Hidden Layers



MULTI-LAYER
PERCEPTRON (MLP)

PERCEPTRON

A **Perceptron** is the simplest form of an artificial neural network model, designed to mimic how a single neuron in the brain works. What happens around a perceptron is central to the learning process of a Neural Network. A single **Neuron/Unit** in a Neural Network works identically to this simplest form of NN.

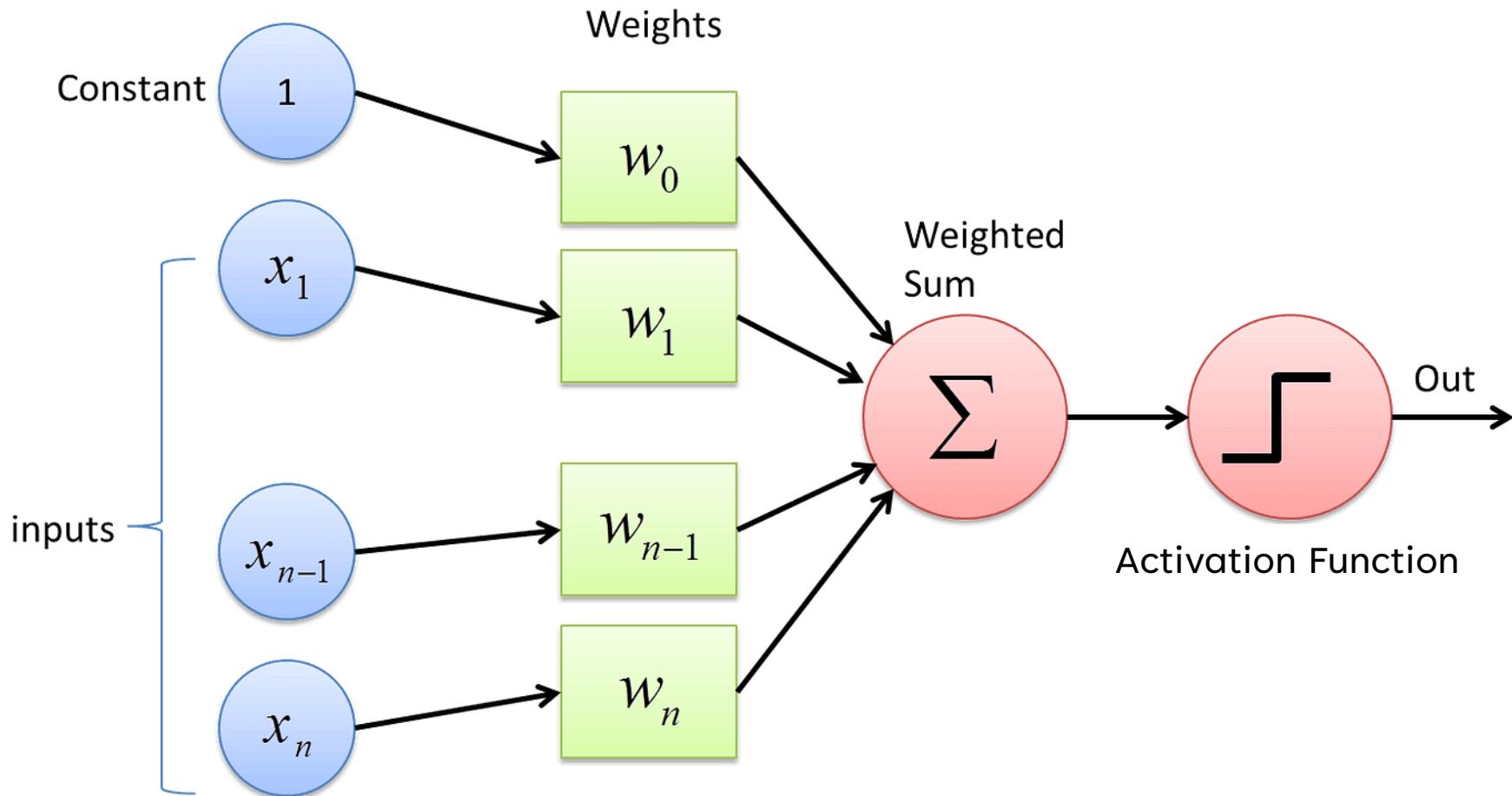
Steps Inside a Perceptron/Neuron

- **Input and Weights:** Each input x_i is multiplied by its corresponding weight w_i .
- **Weighted Sum Calculation:** The perceptron calculates the weighted sum of the inputs and adds the bias.

$$f_w(x) = \sum_{i=0}^n w_i x_i$$

- **Activation Function:** The weighted sum is passed through the **activation function**. This activation function is crucial in **introducing nonlinearity** to the learning process and generating outputs.
- **Output:** The perceptron outputs the activation function's result, either as an input to the next layer or as a final prediction.

PERCEPTRON

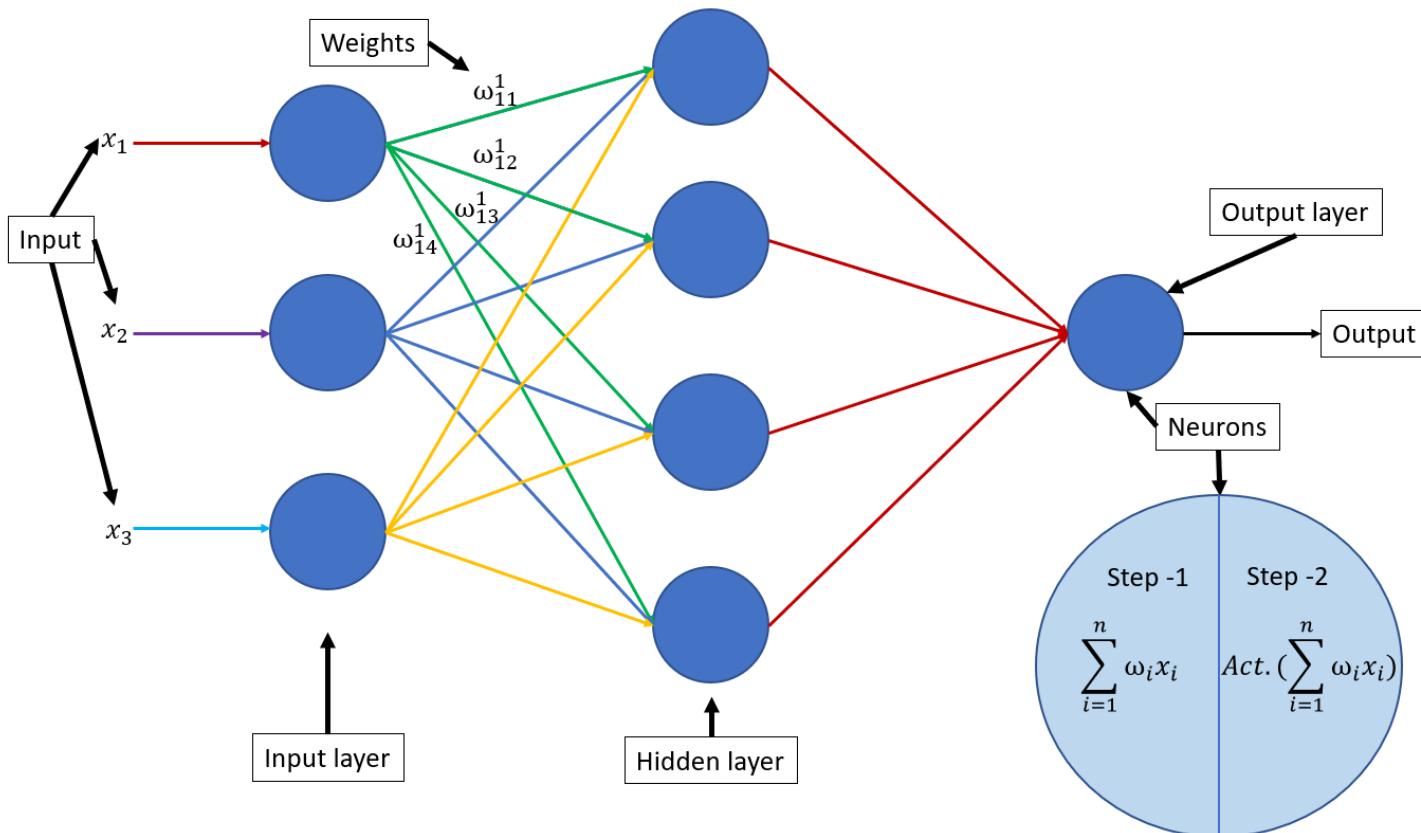


Source: <https://towardsdatascience.com/what-the-hell-is-perceptron-626217814f53>

Above is a diagram of a **Perceptron**. It performs **transformation** (in this case weighted sum) and **activation**, which are central to the modeling of Neural Network based algorithms.

MULTI-LAYER PERCEPTRONS (MLP)

A **Multi-Layer Perceptron (MLP)** is an extension of the basic perceptron model through multiple layers of neurons and is one of the most fundamental types of artificial neural networks used in machine learning. These types of NNs are also known as **Feedforward Neural Networks (FNN)** or **Artificial Neural Networks (ANN)**. Unlike a single perceptron, which is limited to solving linearly separable problems, an MLP can solve more complex, non-linear problems by incorporating multiple layers of neurons.



MULTI-LAYER PERCEPTRONS (MLP)

As we saw earlier, an MLP consists of three primary types of layers:

- Input Layer
- **Hidden Layer(s)**
- Output Layer

Hidden Layer(s) are what differentiates MLP or other Neural Networks from Perceptrons and Traditional ML algorithms. The **Neurons** from these layers serve as **inputs for the next layers** and also act as the **outputs of the previous layers**. These layers are responsible for transforming the inputs into more complex representations.

Key Features of MLP

- **Feedforward Network:** Information in an MLP flows in one direction—from the input layer, through the hidden layer(s), and finally to the output layer.
- **Fully Connected Layers:** Each neuron in one layer is connected to every neuron in the next layer, with each connection having its own weight. When every neuron in one layer is connected to every neuron in the previous and next layers, they are often called **Fully Connected (FC) Layers** or **Dense Layers**.
- **Non-linear Activation Functions:** MLPs use non-linear activation functions in the hidden layers. This allows the network to learn non-linear patterns in the data.

MLP ALGORITHM

As we know, Any ML algorithm has three major aspects. The **Model**, the **Cost Function**, and the **Optimization Algorithm**. In the case of a generic **MLP**, let us take a look at all three:

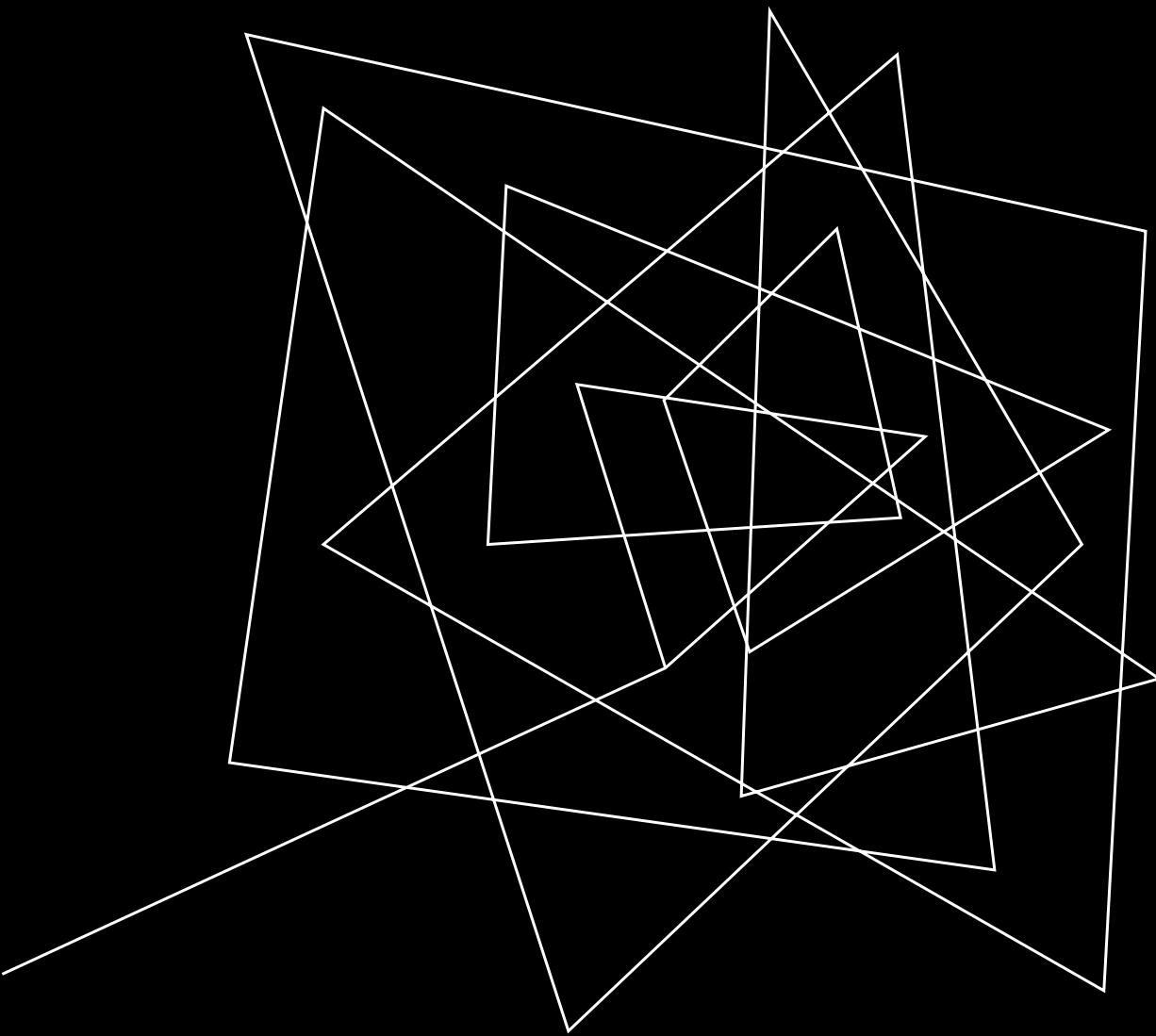
- **Model:** The mathematical relationship (Model) between the input features and the output labels is formed through **Forward Propagation** in MLP. It is the process of passing input data from the input layer, through the hidden layer(s), and finally to the output layer.
- **Cost Function:** The **Cost Functions** used in MLP are identical to what we saw in linear models. For Regression tasks, **MSE** is the most commonly used Cost Function. And for Classification tasks, **Cross-Entropy Loss** is usually the go-to method. Both these Cost functions work just as well in the case of Neural Networks.

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n [t^{(i)} - f_{\theta}(x^{(i)})]^2$$

$$\text{Cross-Entropy Loss} = - [t \log(P) + (1-t)\log(1-P)]$$

- **Optimization:** MLPs optimize the parameters through a process called **Backpropagation**. Backpropagation is basically the extension of **Gradient Descent** through the chain rule.

We will discuss **Forward Propagation** and **Backpropagation** in details in the next slides.



FORWARD
PROPAGATION

FORWARD PROPAGATION

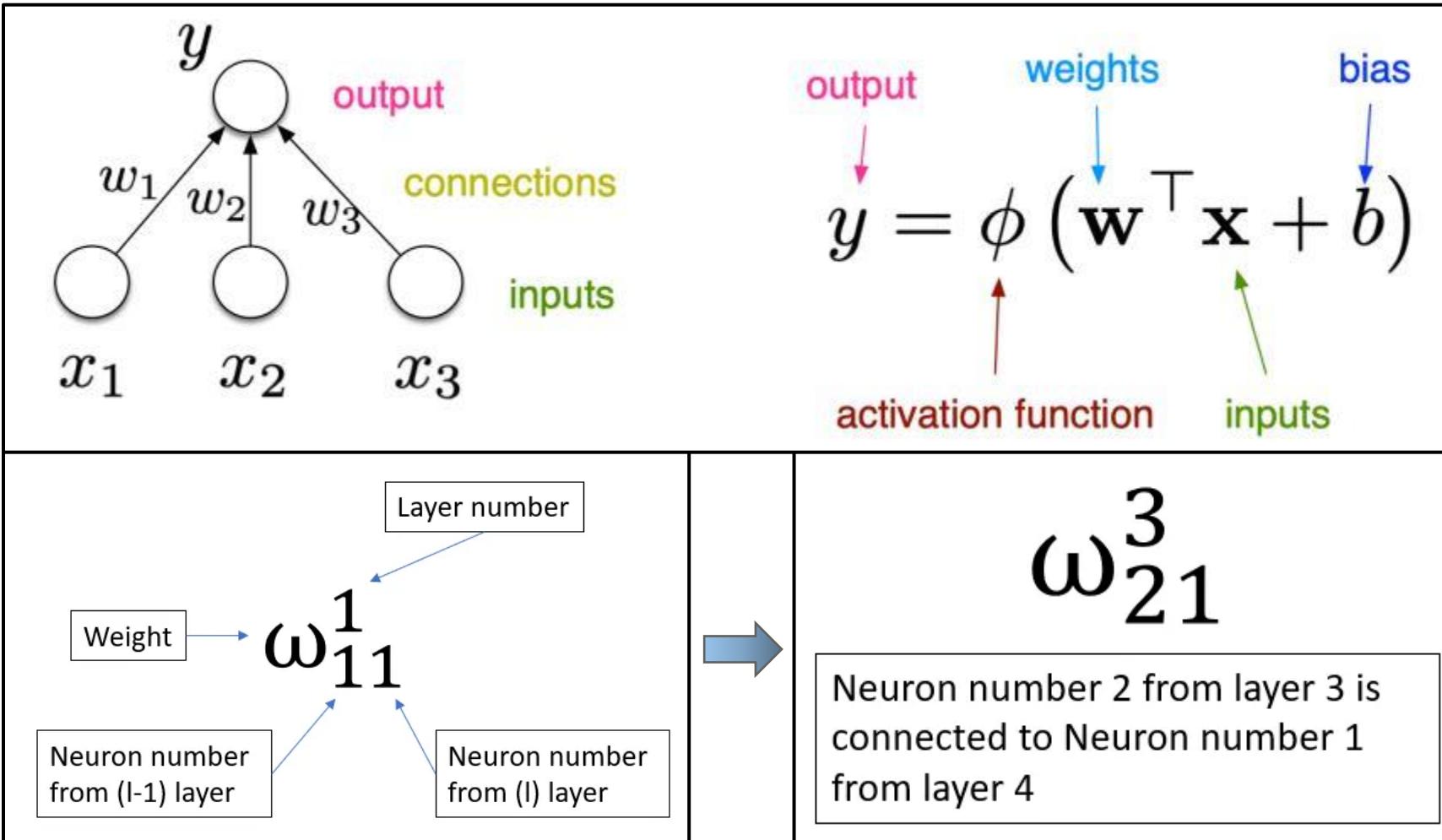
The process of passing input data through the layers of the network is known as **Forward Propagation**. Each neuron computes a weighted sum of its inputs, adds a bias, and passes the result through an activation function. This process continues through all layers until the output layer produces the final prediction.

Forward Propagation Steps

- **Input:** The input features are fed into the neural network.
- **Transformation:** Each neuron in the next layer takes a weighted sum of inputs from the previous layer and adds a bias term.
- **Activation:** The weighted sum is then passed through an activation function (like ReLU, sigmoid, tanh, or SoftMax), which introduces non-linearity to the model.
- **Propagation through Hidden Layers:** The activated output from each layer becomes the input for the next layer. This process continues through the hidden layers.
- **Output:** Finally, the processed data reaches the output layer, where a decision or prediction is made (classification, regression, etc.).

FORWARD PROPAGATION: TERMINOLOGIES

Before going into the Mathematics, let us take a look at the **notations** and **terminologies** we will be using in our discussions.



FORWARD PROPAGATION

Let us assume that we have an L layer Neural Network ($L = L-1$ Hidden Layers + 1 Output Layer). $\mathbf{h}^{(i)}$ is the vector of the values of neurons (called **activations/outputs**) in the i^{th} hidden layer. The function $\mathbf{f}^{(i)}$ represents the overall transformation from $(i-1)^{\text{th}}$ layer to i^{th} layer. Finally, $\Phi(z)$ represents the **activation function**.

Forward Propagation Through the Layers

Each layer computes a function in the neural network, so, overall, the network computes **a composition of functions**.

$$\mathbf{h}^{(1)} = \mathbf{f}^{(1)}(\mathbf{x}) = \Phi(\mathbf{w}^{(1)T}\mathbf{x} + \mathbf{b}^{(1)})$$

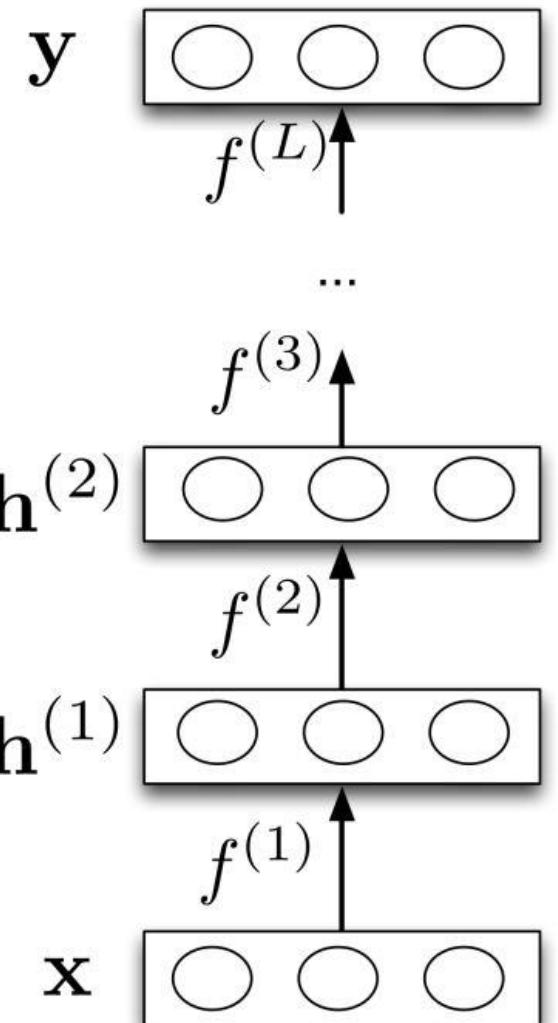
$$\mathbf{h}^{(2)} = \mathbf{f}^{(2)}(\mathbf{h}^{(1)}) = \Phi(\mathbf{w}^{(2)T}\mathbf{h}^{(1)} + \mathbf{b}^{(2)})$$

⋮

$$\mathbf{h}^{(i)} = \mathbf{f}^{(i)}(\mathbf{h}^{(i-1)}) = \Phi(\mathbf{w}^{(i)T}\mathbf{h}^{(i-1)} + \mathbf{b}^{(i)})$$

⋮

$$\mathbf{y} = \mathbf{f}^{(L)}(\mathbf{h}^{(L-1)})$$



FORWARD PROPAGATION

Forward Propagation Through the Layers

So, Overall, we can represent the input to output mapping as:

$$y = f^{(L)} \cdot f^{(L-1)} \cdot \dots \cdot f^{(2)} \cdot f^{(1)}(x)$$

In the **output layer**,

- If the task is **Regression**:

$$y = f^{(L)}(h^{(L-1)}) = w^{(L)T} h^{(L-1)} + b^{(L)}$$

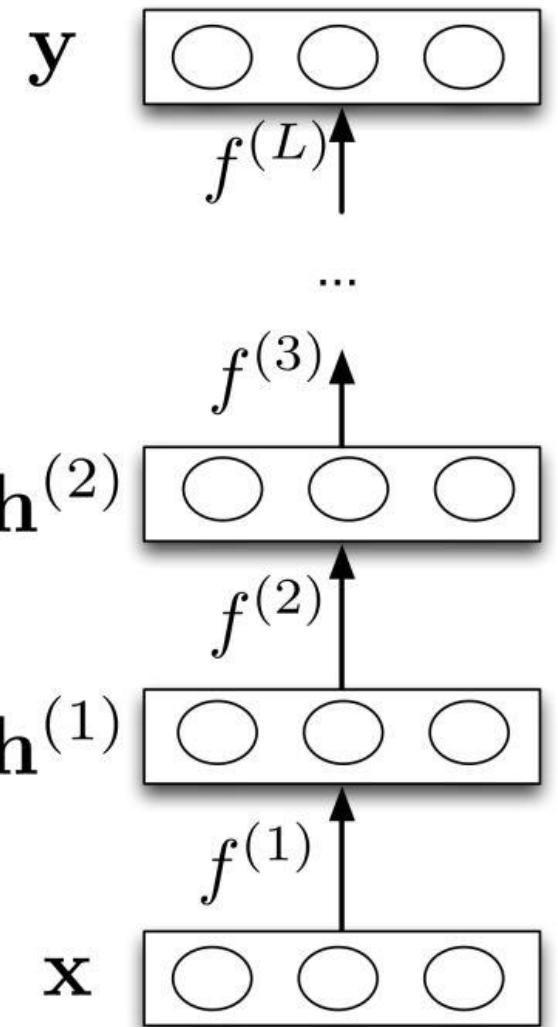
- If the task is **Binary Classification**:

$$y = f^{(L)}(h^{(L-1)}) = \sigma(w^{(L)T} h^{(L-1)} + b^{(L)})$$

- If the task is **Multinomial Classification** :

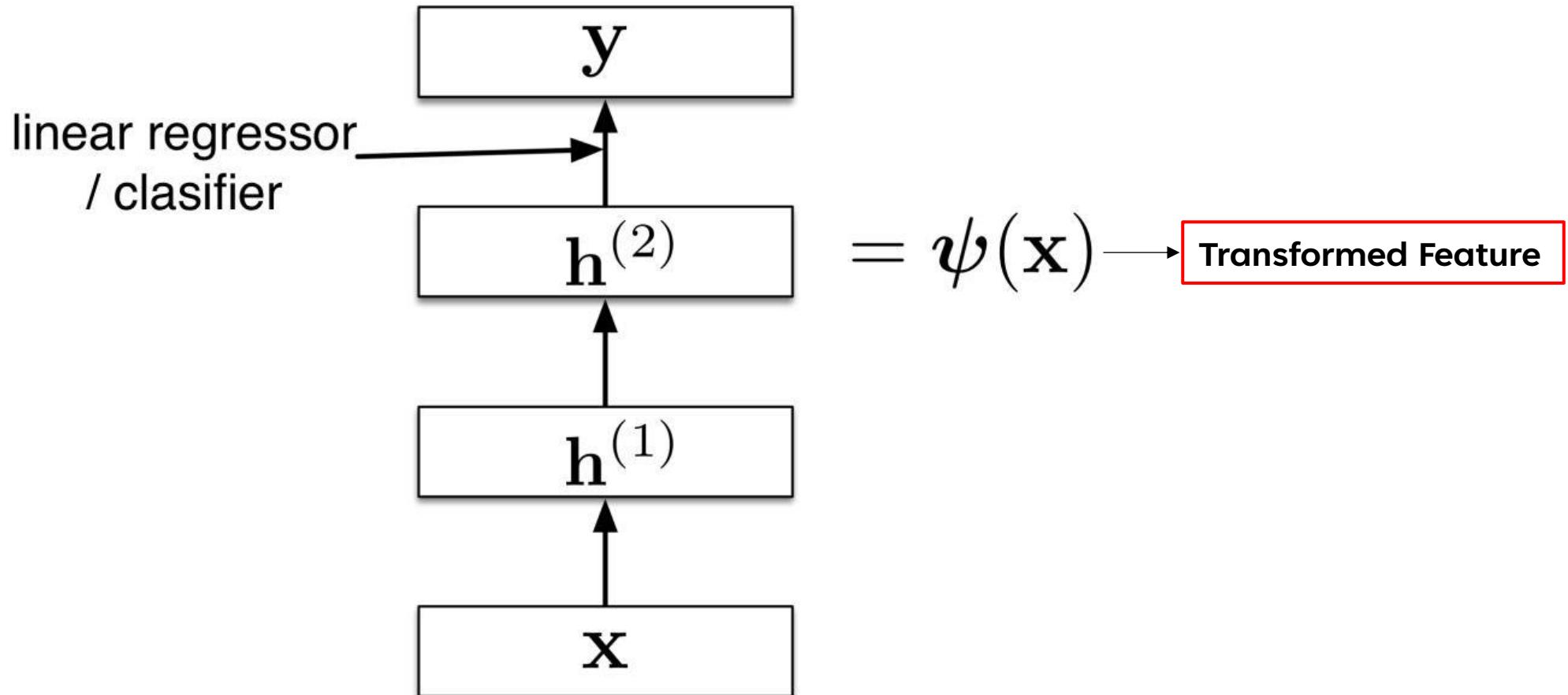
$$y = f^{(L)}(h^{(L-1)}) = \text{SoftMax}(w^{(L)T} h^{(L-1)} + b^{(L)})$$

Finally, if it's a classification task, then **thresholding** or **argmax** is applied to generate the predicted label.



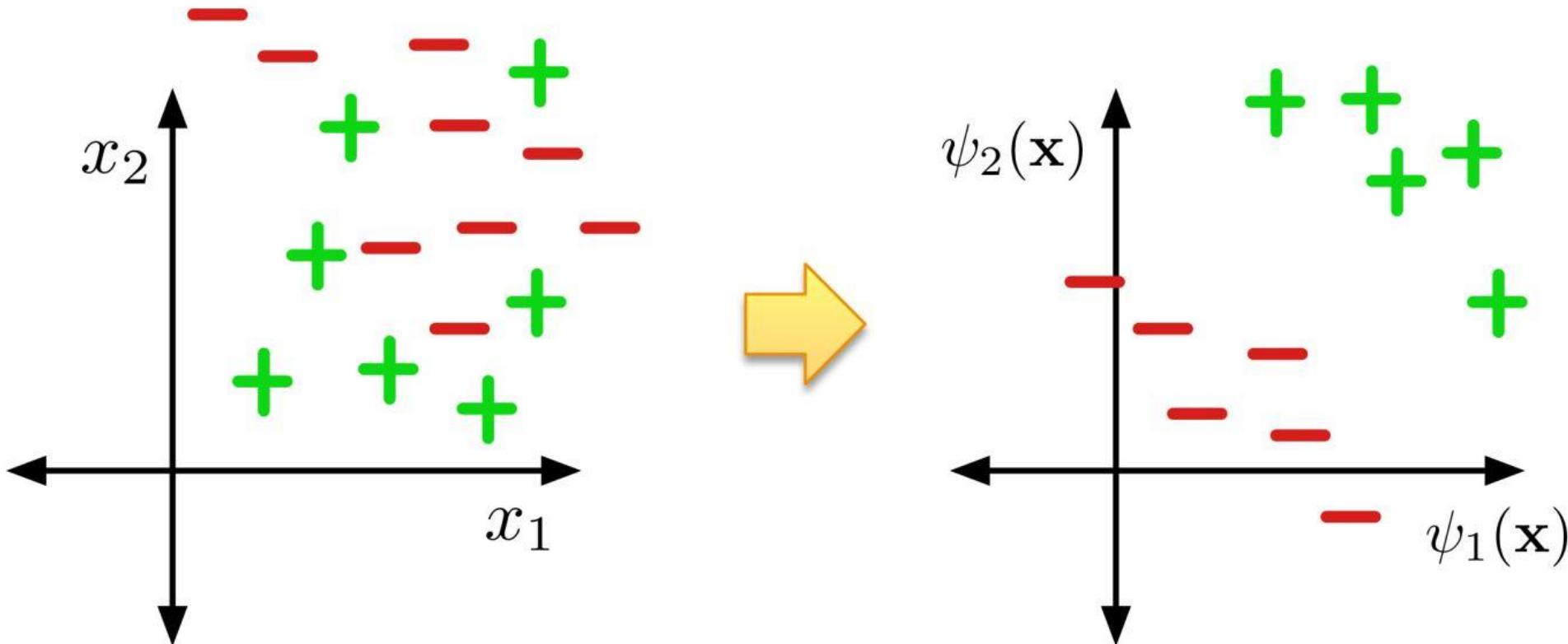
FEATURE LEARNING

As we can see from the output layers, Neural Networks can be viewed as a way of **learning Features**:



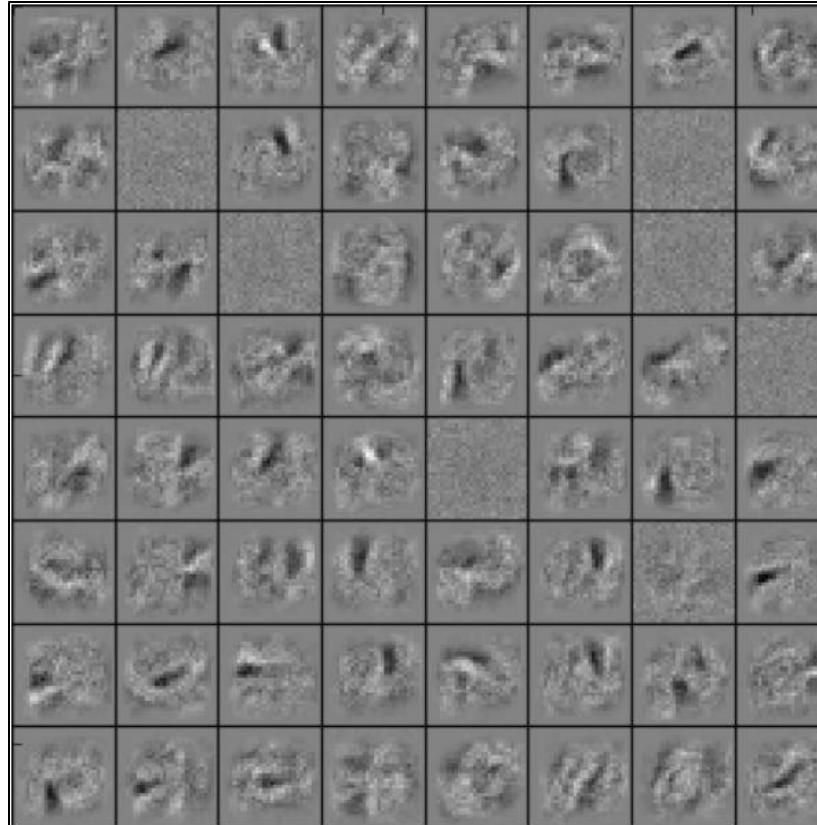
FEATURE LEARNING

The Goal: Separation of Classes in the Transformed Feature Space

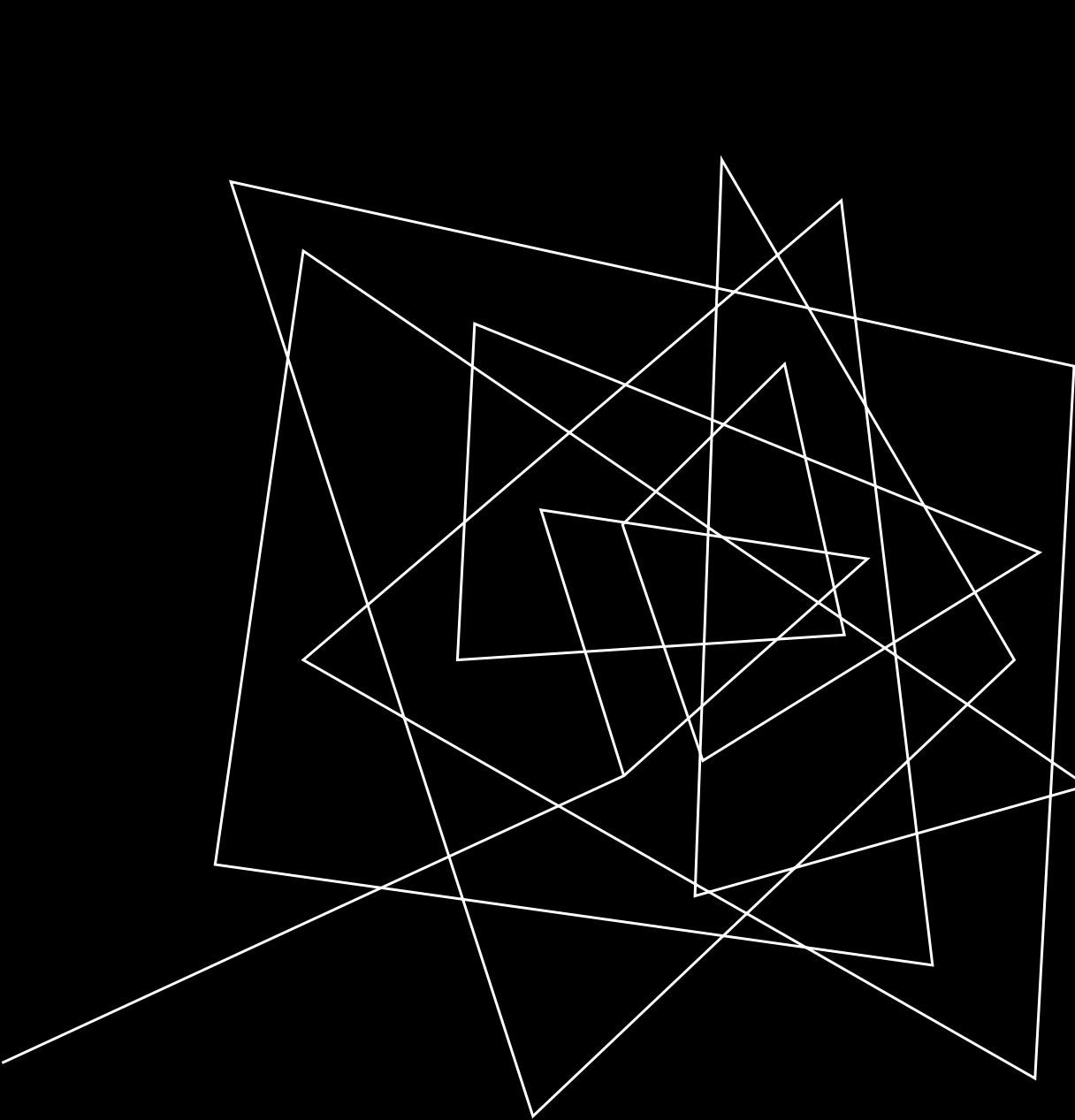


FEATURE LEARNING

Here are some of the features learned by the first hidden layer of a handwritten digit classifier (after rearranging the **activations** as images):



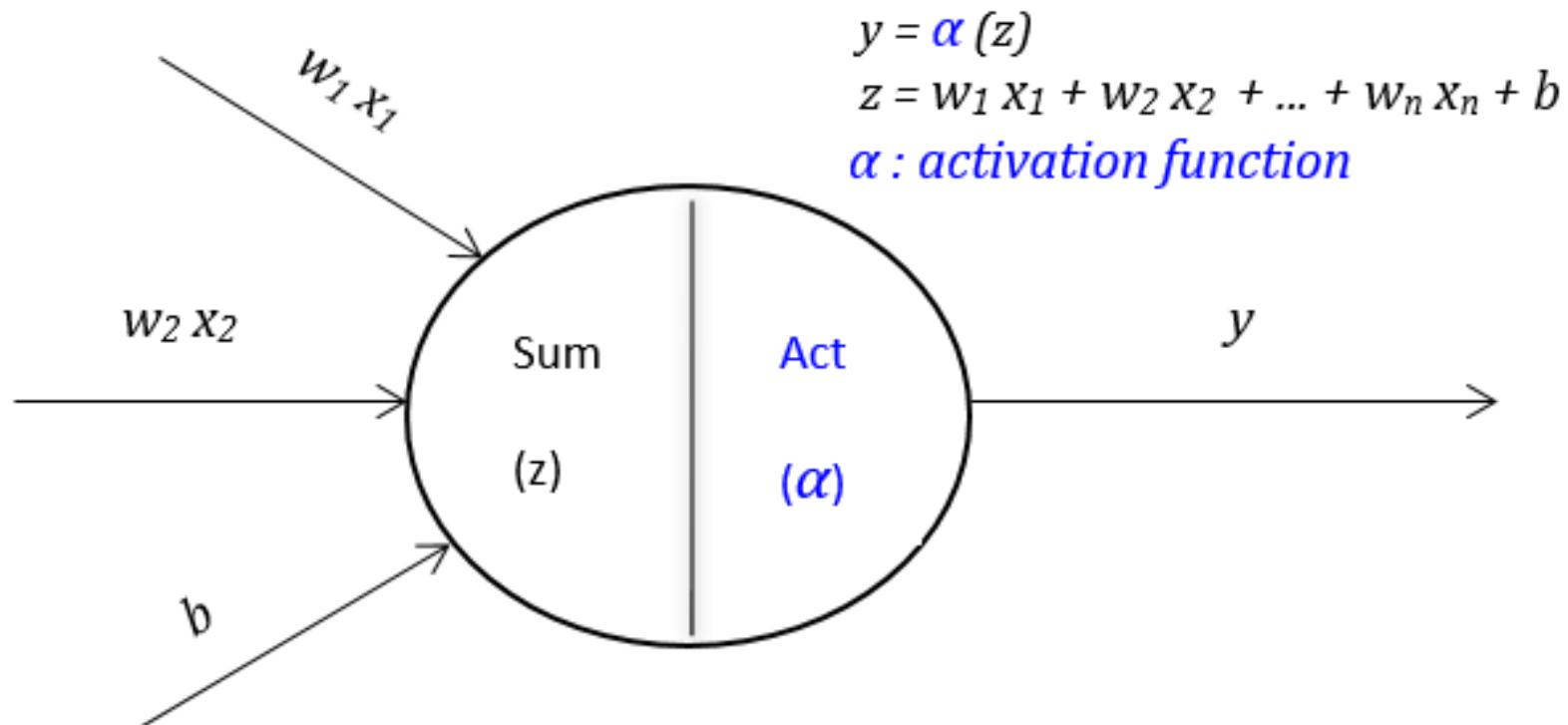
Unlike hard-coded feature maps (e.g., in polynomial regression), features learned by neural networks **adapt to patterns** in the data.



ACTIVATION FUNCTIONS

ACTIVATION FUNCTIONS

An **Activation Function** in a neural network determines whether a **neuron** should be activated or not based on the weighted sum of inputs. It applies **non-linear transformations** to the linear outputs, introducing non-linearity into the model, which is crucial for the network to learn complex patterns and relationships in the data.



ACTIVATION FUNCTIONS

Why Activation Functions Are Needed

- ❑ **Non-linearity:** Let us examine a sequence of **linear layers** (*Linear Activation such as Identity or Scaling*).

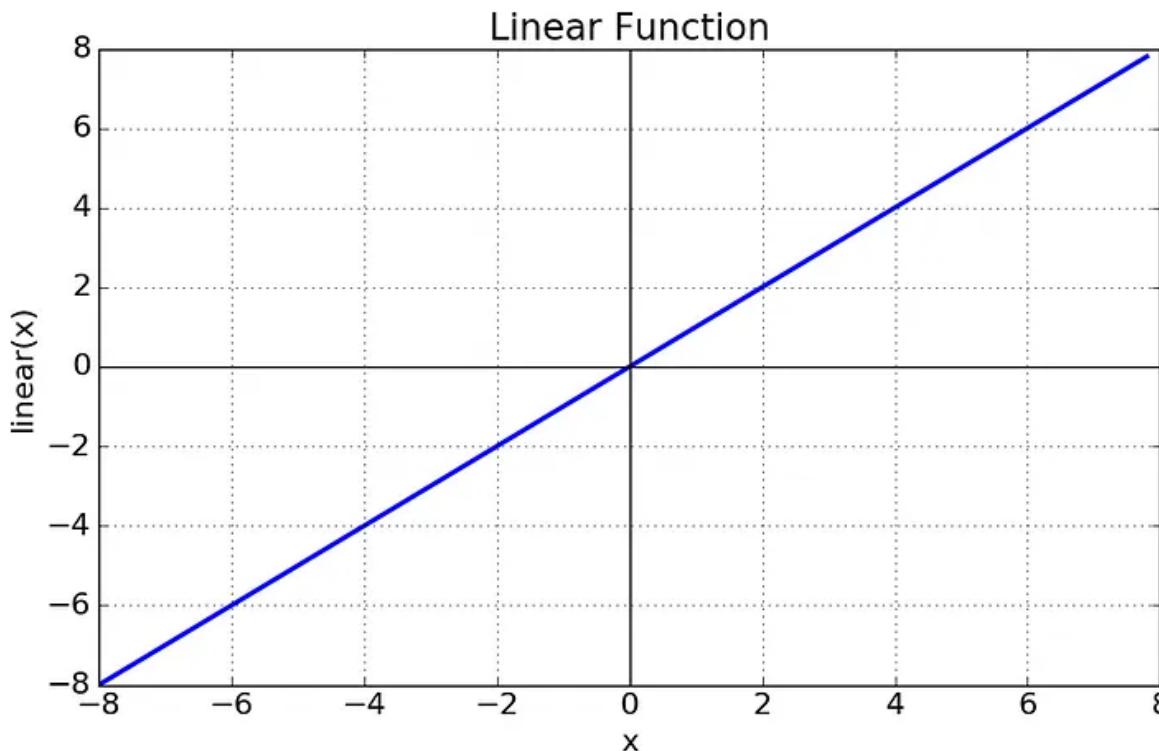
$$y = \underbrace{\mathbf{W}^{(3)} \mathbf{W}^{(2)} \mathbf{W}^{(1)}}_{\triangleq \mathbf{W}'} \mathbf{x}$$

- As we can see, the sequence of layers can be equivalently represented with a single linear layer. So, without non-linear activation functions, a neural network would behave like a linear model regardless of how many layers it has, which limits its ability to solve complex problems. Non-linear activation functions allow the network to **model complex, non-linear patterns**.
- ❑ **Flow of Information:** Activation functions help neurons decide whether to pass information forward or not, mimicking biological neurons.
- ❑ **Output:** They help map the output of neurons to a range of values in the output layer (e.g., probabilities or binary decisions) based on the specific problem, such as classification or regression.

ACTIVATION FUNCTIONS: IDENTITY

The **Identity** or **Linear Activation Function** is one of the simplest activation functions used in neural networks where **the output equals the input**. It is typically used in the output layer of neural networks for regression tasks to predict continuous values, but it lacks the ability to introduce non-linearity, limiting its use in hidden layers.

$$\text{Identity, } \Phi(z) = z$$

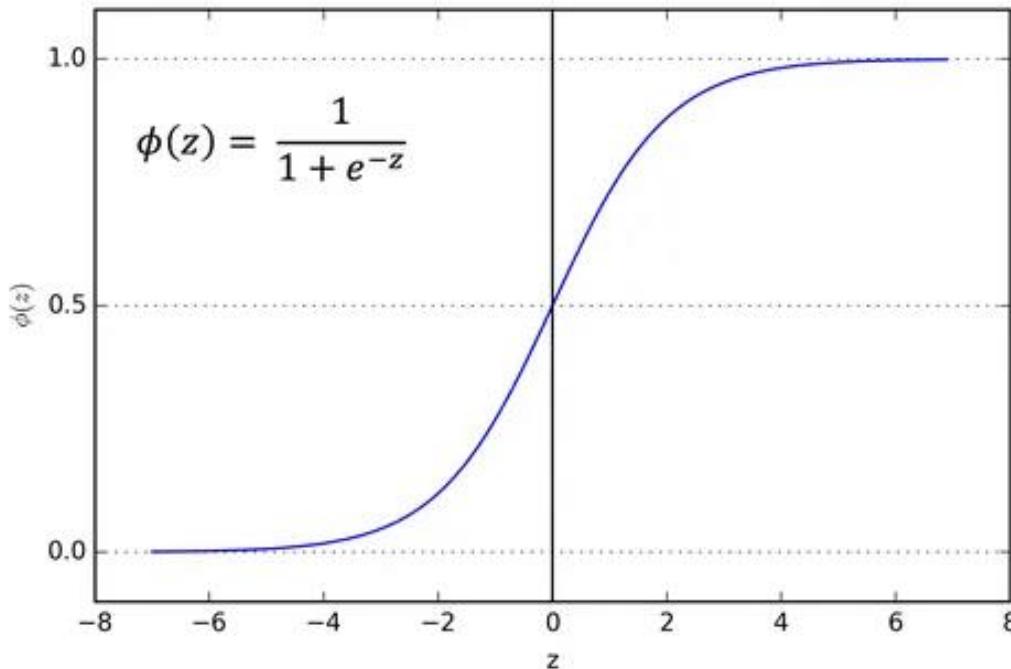


Source: <https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6>

ACTIVATION FUNCTIONS: SIGMOID

The **Sigmoid Activation** Function transforms any real-valued number into a value between **0** and **1**. This characteristic makes it ideal for binary classification problems, where the output can be interpreted as a probability. As x approaches infinity, $f(x)$ approaches 1, and as x approaches negative infinity, $f(x)$ approaches 0. However, it can suffer from **Vanishing Gradient** problems when the input value is far from zero.

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

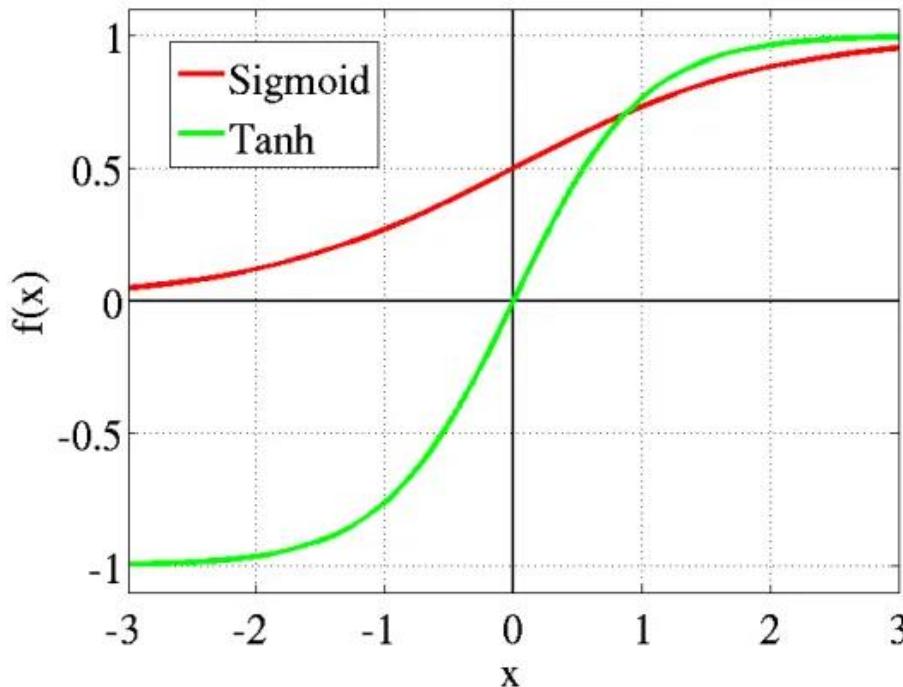


Source: <https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6>

ACTIVATION FUNCTIONS: TANH

The **Tanh Activation Function (Hyperbolic Tangent)** outputs values between **-1** and **1**. This makes the tanh function useful when the neural network needs to output both negative and positive values. Tanh is similar to the sigmoid but generally performs better in hidden layers because its **outputs are centered around zero**, which helps with optimization. However, like sigmoid, it also suffers from **Vanishing Gradient** problems for large input values.

$$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

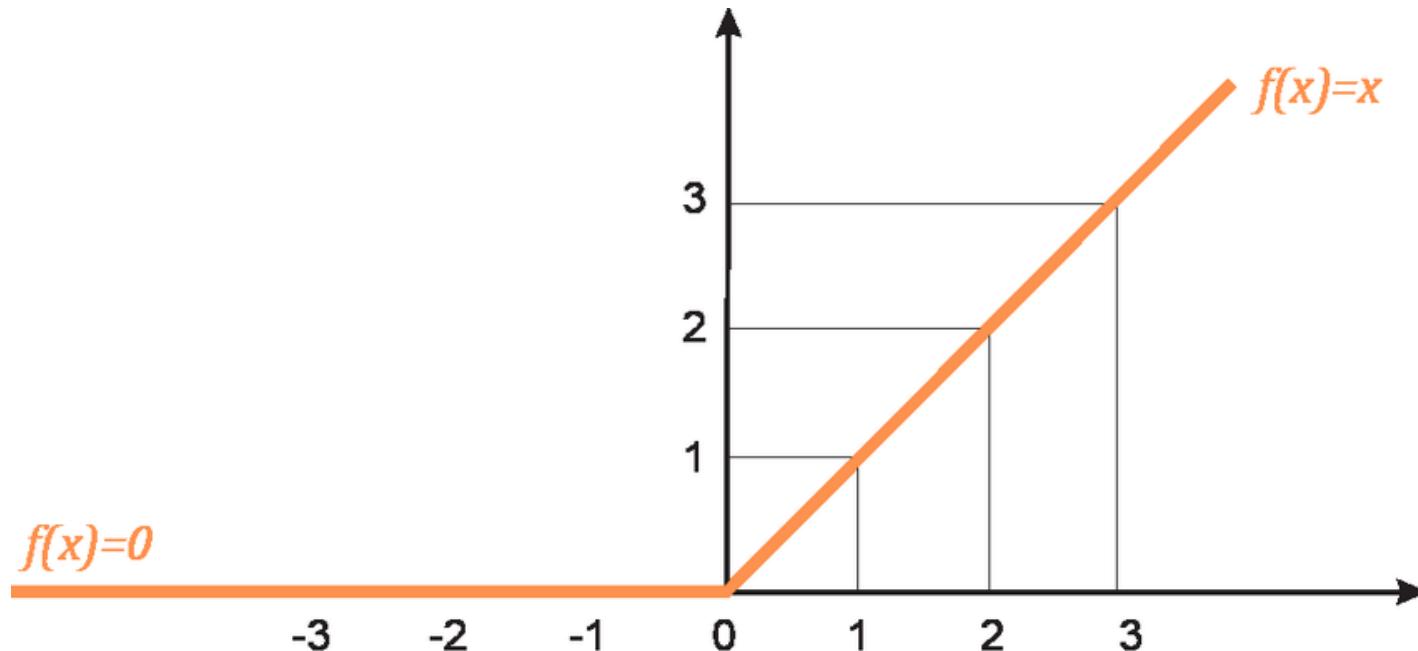


Source: <https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6>

ACTIVATION FUNCTIONS: RELU

The **ReLU Activation Function (Rectified Linear Unit)** outputs zero for any negative input and outputs the input itself if it's positive. ReLU is widely used in deep learning due to its simplicity and effectiveness. It helps **mitigate the Vanishing Gradient** problem, enabling faster training of deep neural networks. However, ReLU can suffer from the "**dying ReLU**" problem, where neurons can become inactive if their inputs are negative, causing gradients to be zero and preventing further learning.

$$\text{ReLU}(z) = \max(0, z)$$

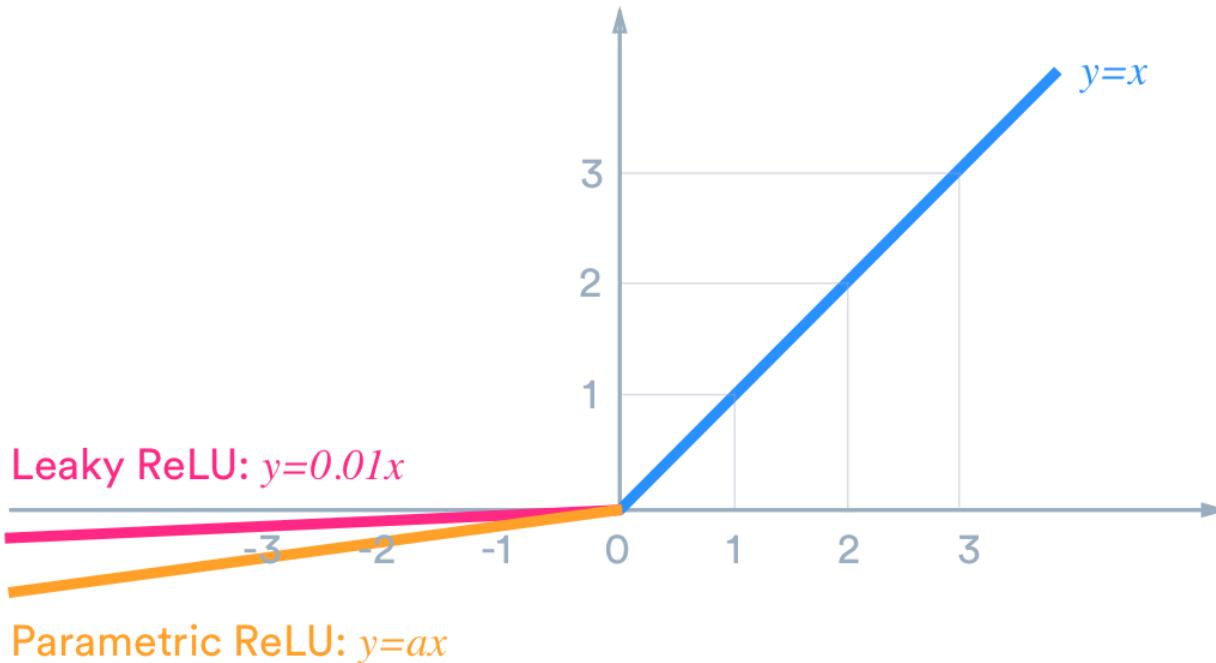


ACTIVATION FUNCTIONS: LEAKY RELU

The **Leaky ReLU** Activation Function is a variation of the ReLU function. It is defined as:

$$f(z) = \max(\alpha z, z)$$

Here, α is a small constant (typically **0.01**), which allows a small, non-zero gradient when the input is negative. This prevents the "**dying ReLU**" problem by ensuring that neurons with negative inputs still have a small output, allowing them to continue learning even for negative values. Leaky ReLU is often used as an alternative to ReLU in neural networks to avoid inactive neurons.



ACTIVATION FUNCTIONS: SOFTMAX

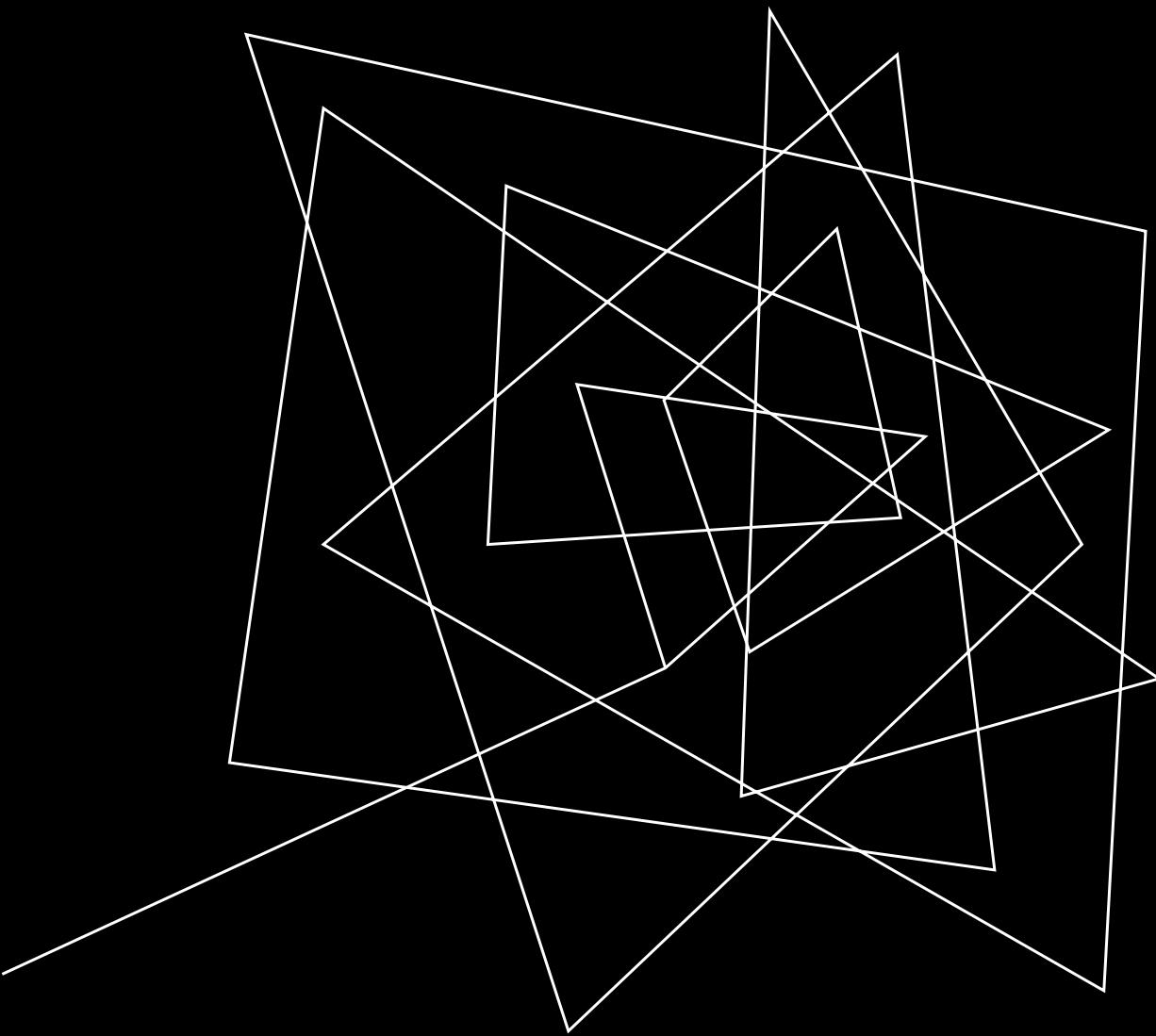
The **SoftMax** Activation Function is used in the output layer of neural networks, particularly for multi-class classification problems. It converts the raw output scores (also called **logits**) into **probabilities**. The SoftMax function is defined as:

$$\text{SoftMax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^K e^{z_j}} = \frac{\exp(z_i)}{\exp(z_1) + \exp(z_2) + \dots + \exp(z_K)}$$

Where:

- z_i is the raw output (logit) of class i ,
- K is the number of classes,
- The numerator is the exponent of the raw output for class i ,
- The denominator is the sum of the exponentials of the logits for all classes.

The output values from **SoftMax** lie **between 0 and 1**, and they **sum up to 1**, which can be interpreted as probabilities over the possible classes.



BACKPROPAGATION

CALCULUS REVIEW: GRADIENTS

Definition: The Gradient is a vector that contains all the partial derivatives of a function with respect to its variables. If $f(x,y)$ is a function of two variables, the gradient is:

$$\nabla f(x,y) = \left(\frac{\partial f(x,y)}{\partial x}, \frac{\partial f(x,y)}{\partial y} \right)$$

The Gradient vector points in the direction of the steepest Ascent (the direction in which the function increases most rapidly).

Example: If $f(x, y) = x^2 + y^2$, then the gradient is:

$$\nabla f(x,y) = (2x, 2y)$$

Significance: In optimization, particularly in Gradient Descent, the negative of the gradient points in the direction of the steepest descent (the direction in which the function decreases the most). This is why the gradient is used to update the parameters in optimization algorithms. It provides the most efficient direction to reduce the cost function.

CALCULUS REVIEW: CHAIN RULE

Definition: The **Chain Rule** is a fundamental concept in calculus used to compute the derivative of a **composite function** (like the one we saw in MLP model). When a function is composed of multiple functions, the chain rule allows us to differentiate it by breaking it down into its constituent parts. If we have two functions, say: $f(x)$ and $g(x)$, and we create a composite function $h(x)=f(g(x))$, where f is applied to $g(x)$, then the chain rule helps to differentiate $h(x)$ with respect to x (denoted by $h'(x)$).

For the composite function $h(x)=f(g(x))$, the derivative of $h(x)$ with respect to x is:

$$h'(x) = f'(g(x)) \cdot g'(x)$$

So, for each neuron in the network, the **derivative of the loss function $\mathcal{L}(w)$** with respect to a weight w is calculated by:

$$\frac{\partial \mathcal{L}}{\partial w} = \frac{\partial \mathcal{L}}{\partial z} \cdot \frac{\partial z}{\partial w}$$

Where, z is the output of the Neuron.

Significance: The chain rule is essential in multivariable calculus, neural networks (backpropagation), and many other fields where complex functions are composed of simpler ones. It allows us to tackle differentiation in layers and systematically break down how changes in one variable propagate through other dependent functions.

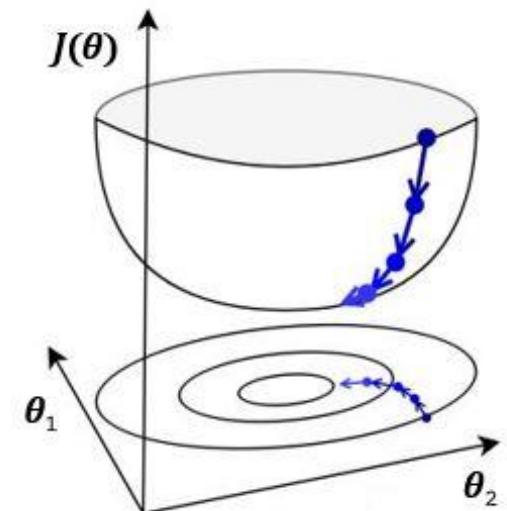
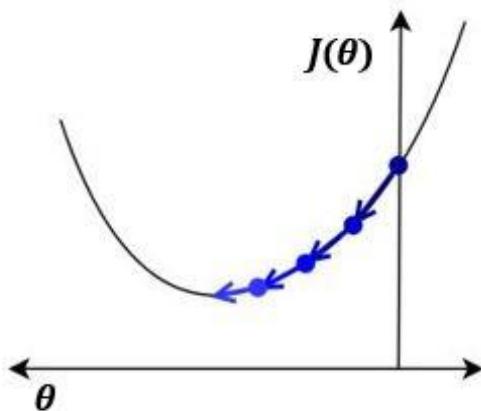
GRADIENT DESCENT (REVIEW)

Gradient Descent is an iterative Optimization Algorithm that aims to *find the minimum of a Cost Function*. The cost function measures how well a machine learning model's predictions match the actual data. The objective is to *adjust the model parameters/weights to minimize this Cost Function*.

The Process consists of **two stages**:

- Iteratively computing the **Gradient** to determine the direction in which the function decreases most steeply.
- Taking a step in that direction.

Steps Towards the Minima (Where the value of $J(\theta)$ is minimum).



Source: https://www.cs.toronto.edu/~rgrosse/courses/csc311_f20/

GRADIENT DESCENT (REVIEW)

Observe:

- If $\frac{\partial J(w)}{\partial w_j} > 0$ (Positive slope), then increasing w_j increases $J(w)$.
- If $\frac{\partial J(w)}{\partial w_j} < 0$ (Negative slope), then increasing w_j decreases $J(w)$.
- The following update (**Gradient Descent**) always decreases the cost function for small enough α (unless $\frac{\partial J(w)}{\partial w_j} = 0$, where it remains unchanged at the minima/maxima):

$$w_{j(\text{new})} \leftarrow w_{j(\text{old})} - \alpha \frac{\partial J(w)}{\partial w_j}$$

Learning Rate, α : It is a *hyperparameter* that controls the size of the steps taken to reach the minimum. The larger it is, the faster w changes. α is always greater than 0.

- Tuning the learning rate is usually necessary, but the values are typically small, e.g., 0.01 or 0.0001.

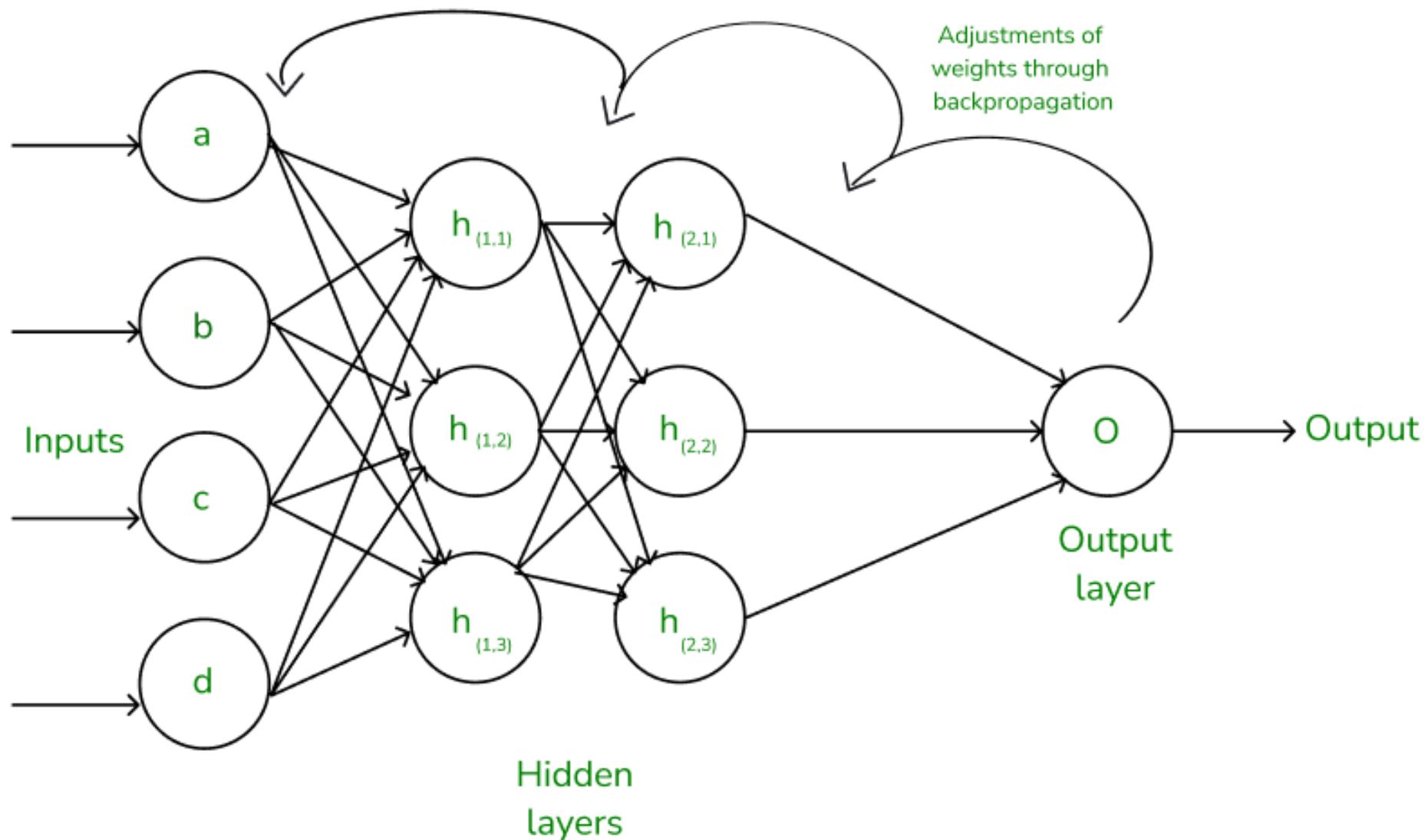
BACKPROPAGATION

Backpropagation (short for “Backward Propagation of Errors”) is a fundamental optimization algorithm used for training artificial neural networks, particularly deep neural networks (DNNs). Backpropagation is the core algorithm used to optimize the network through **Gradient Descent**. The idea is to **propagate** the error (the difference between the predicted and actual values) **backward** through the network, starting **from the output layer to the input layer**, while **updating the weights** to minimize this error.

Optimization Steps

1. **Forward Propagation:** Calculating activations and the cost.
2. **Gradient Computation:** Starting from last layer (Output) & computing the derivatives of the activations and linear transformation outputs to find out the gradients of the weights and bias of that layer.
3. **Update Weights:** Use gradient descent to update weights and biases of that layer.
4. **Backpropagation:** Propagate back to previous layer to compute the gradients.
5. **Repeat:** Repeat the **Gradient Computation**, **Weight Update** & **Backpropagation** Steps all the way to the input layer.

BACKPROPAGATION

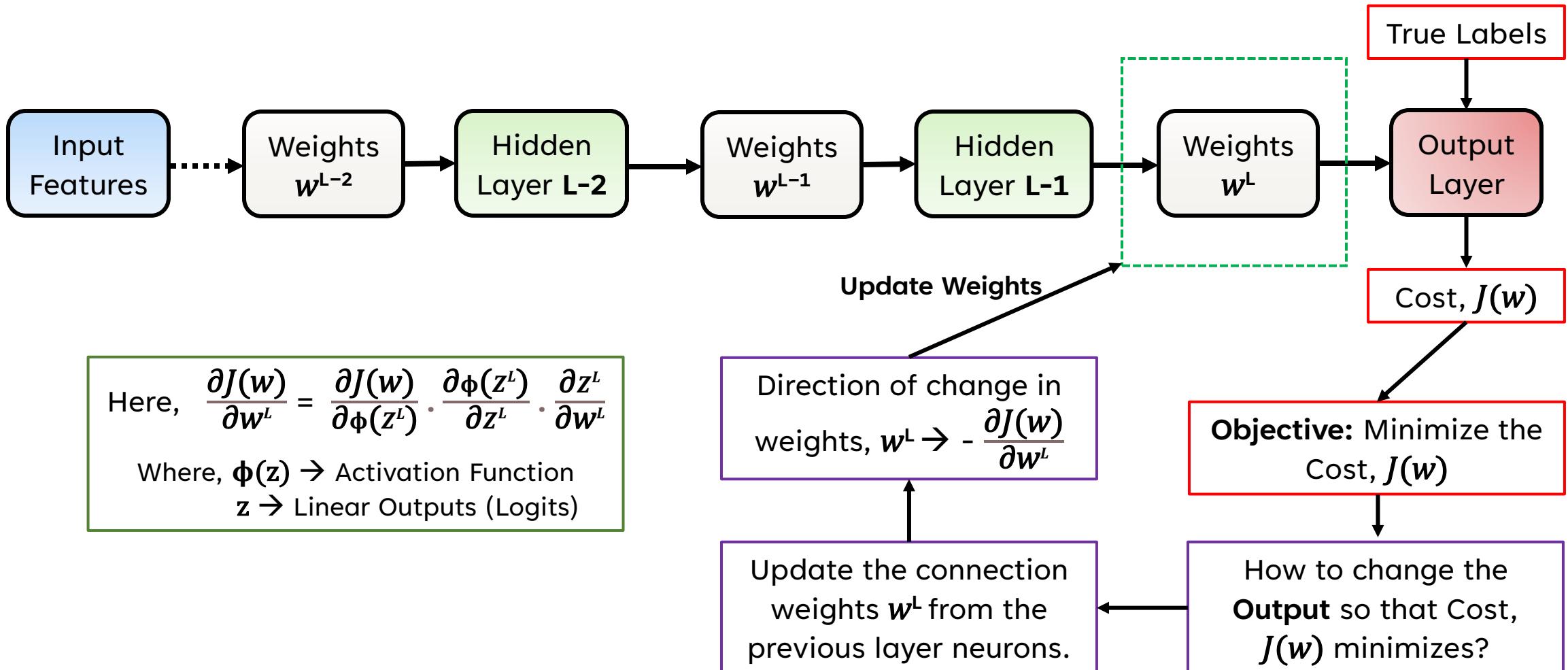


Source: <https://www.geeksforgeeks.org/backpropagation-in-neural-network/>

BACKPROPAGATION: THE IDEA

Step 1: Final Layer (L)

Let us assume an L layer Neural Network, The **Forward Propagation** has just been finished.



$$\text{Here, } \frac{\partial J(w)}{\partial w^L} = \frac{\partial J(w)}{\partial \phi(z^L)} \cdot \frac{\partial \phi(z^L)}{\partial z^L} \cdot \frac{\partial z^L}{\partial w^L}$$

Where, $\phi(z) \rightarrow$ Activation Function
 $z \rightarrow$ Linear Outputs (Logits)

Direction of change in
weights, $w^L \rightarrow -\frac{\partial J(w)}{\partial w^L}$

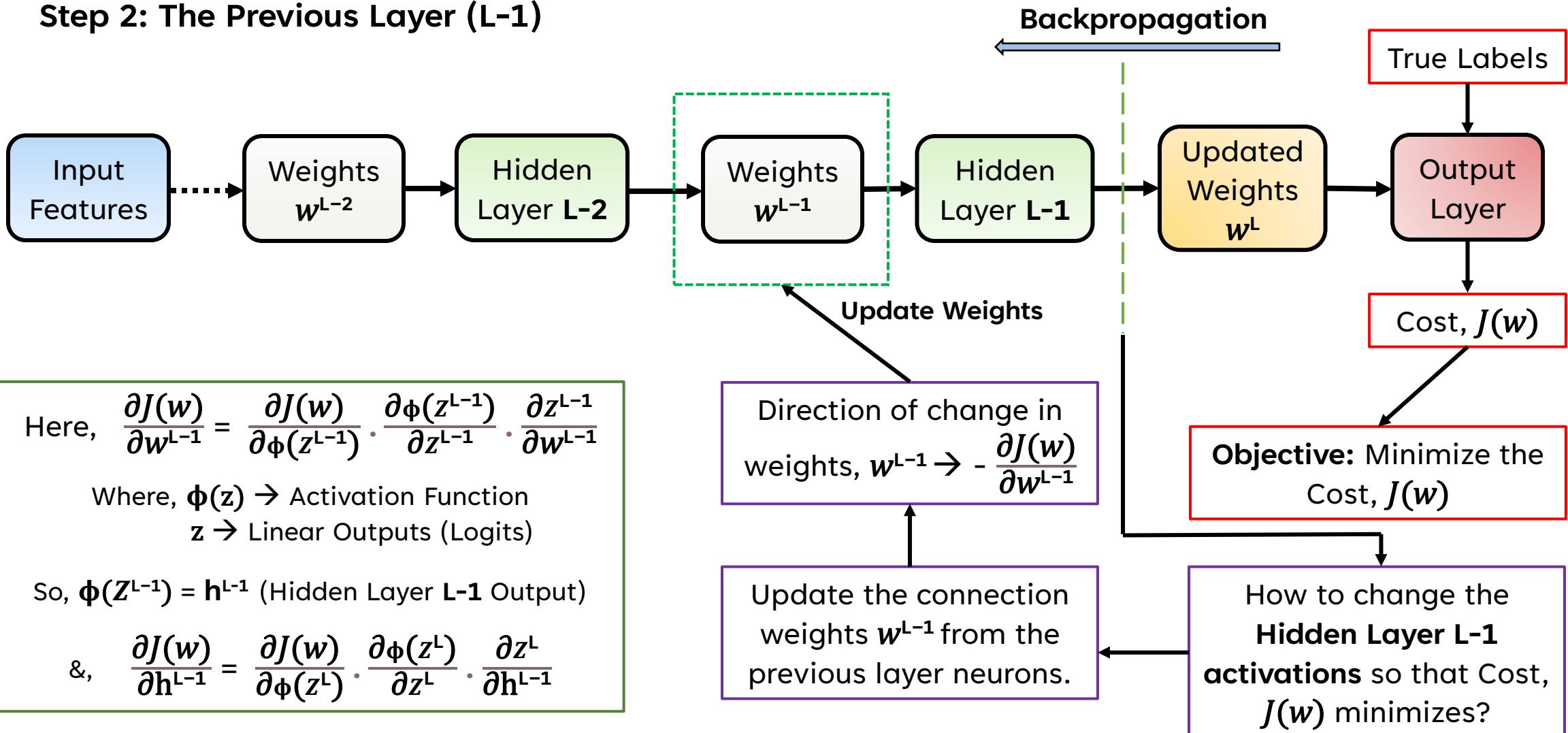
Update the connection
weights w^L from the
previous layer neurons.

Objective: Minimize the
Cost, $J(w)$

How to change the
Output so that Cost,
 $J(w)$ minimizes?

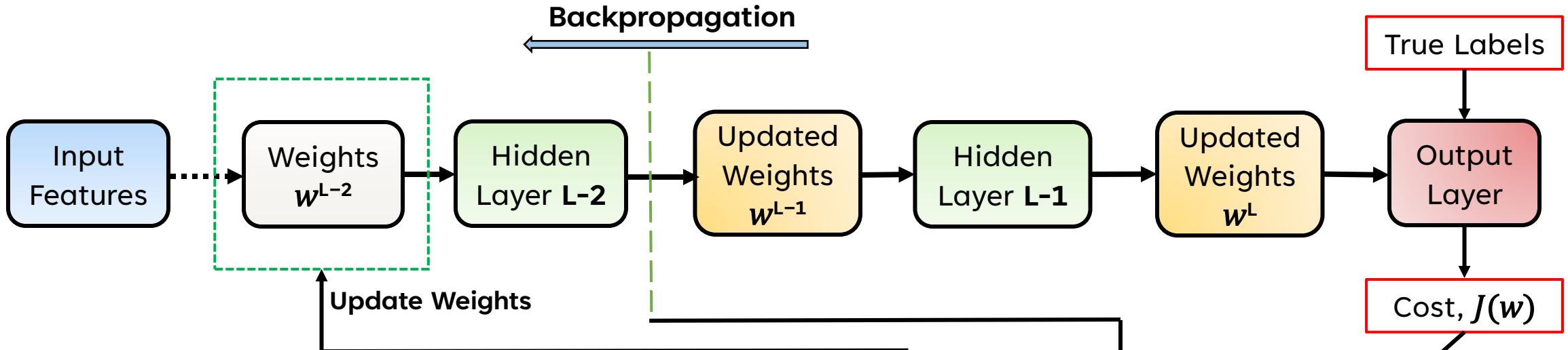
BACKPROPAGATION: THE IDEA

Step 2: The Previous Layer (L-1)



BACKPROPAGATION: THE IDEA

Step 3: The Previous Layer (L-2)



Here, $\frac{\partial J(w)}{\partial w^{L-2}} = \frac{\partial J(w)}{\partial \phi(z^{L-2})} \cdot \frac{\partial \phi(z^{L-2})}{\partial z^{L-2}} \cdot \frac{\partial z^{L-2}}{\partial w^{L-2}}$

Where, $\phi(z) \rightarrow$ Activation Function
 $z \rightarrow$ Linear Outputs (Logits)

Note that, for any Layer $\ell+1$ with updated weights,

$$\phi(Z^\ell) = h^\ell \text{ (Hidden Layer } \ell \text{ Output)}$$

$$\&, \quad \frac{\partial J(w)}{\partial h^\ell} = \frac{\partial J(w)}{\partial h^{\ell+1}} \cdot \frac{\partial h^{\ell+1}}{\partial Z^{\ell+1}} \cdot \frac{\partial Z^{\ell+1}}{\partial h^\ell}$$

Direction of change in weights, $w^{L-2} \rightarrow -\frac{\partial J(w)}{\partial w^{L-2}}$

Update the connection weights w^{L-2} from the previous layer neurons.

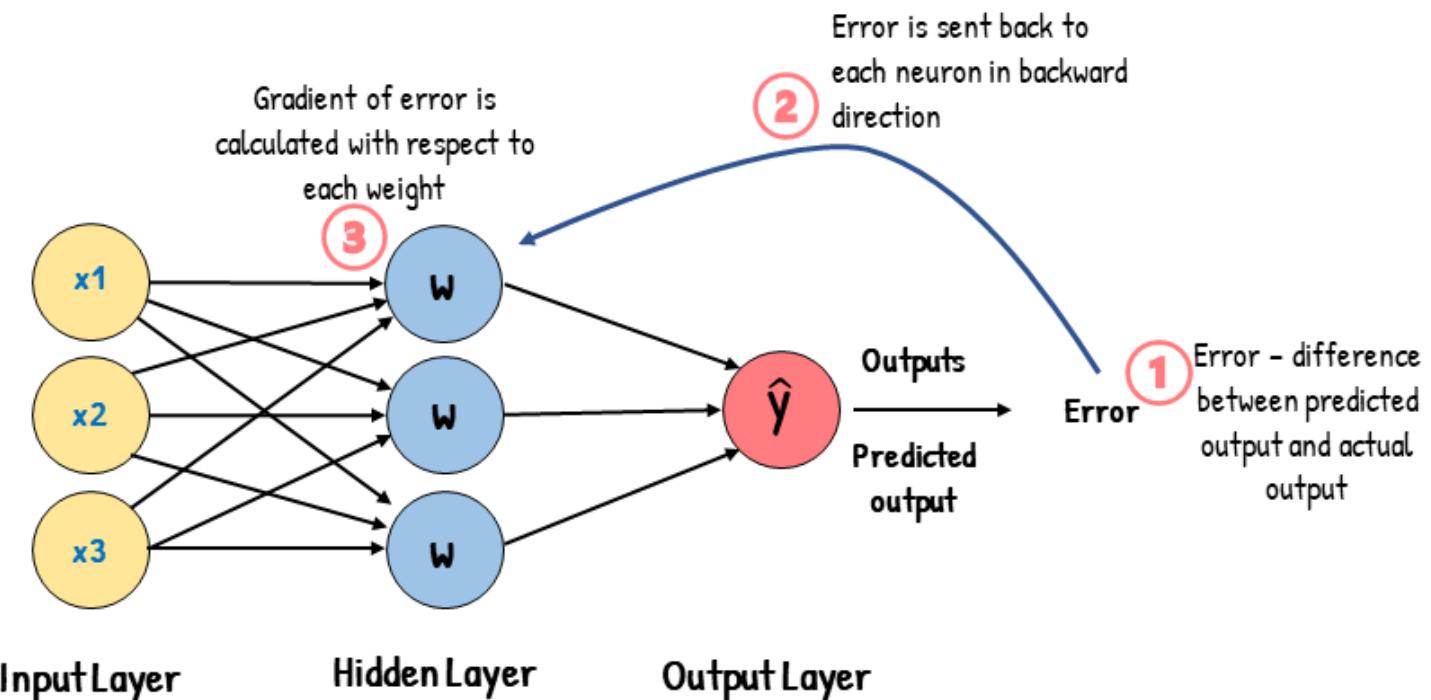
Objective: Minimize the Cost, $J(w)$

How to change the Hidden Layer L-2 activations so that Cost, $J(w)$ minimizes?

BACKPROPAGATION: THE IDEA

Step N: Layer L-(N-1)

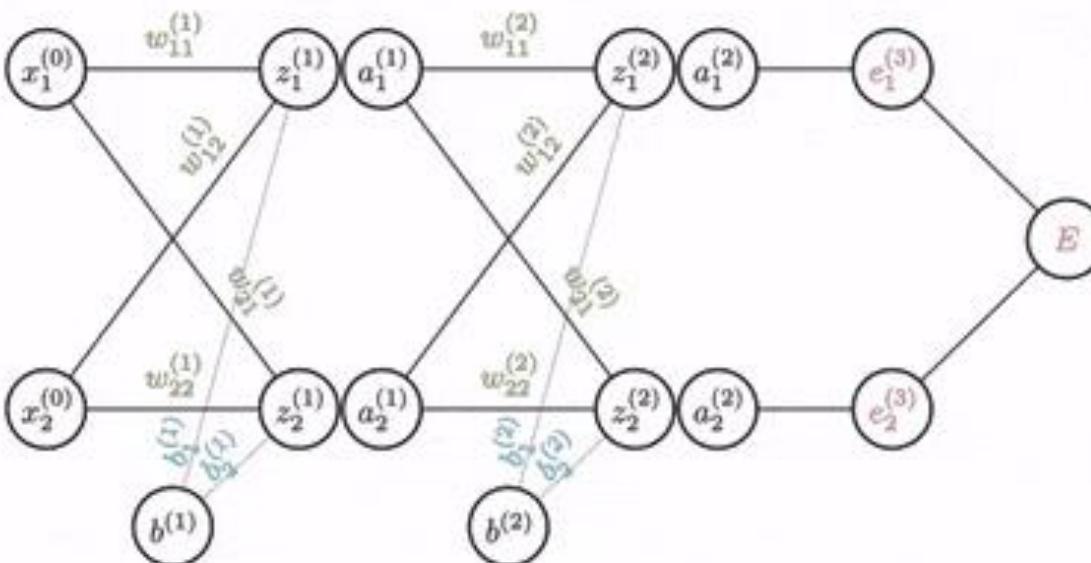
And the **weight optimization** process through **Backpropagation** continues this way. The **Error** is propagated backwards to every neuron in every layer and exploitation of the **Chain Rule** enables to **update every single weight** right up until the first layer (Input Layer → Hidden Layer 1). Backpropagation is a piece of mathematical genius, and a fascinating implementation of **Gradient Descent** approach in Neural Networks.



BACKPROPAGATION

Take a look at the animation below to understand how the chain rule is applied on Gradient vectors and Matrices to implement the **Backpropagation** process efficiently.

$$\frac{\partial E}{\partial w_{11}^{(2)}} = \frac{\partial e_1^{(3)}}{\partial a_1^{(2)}} \frac{\partial a_1^{(2)}}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial w_{11}^{(2)}}$$
$$\frac{\partial E}{\partial w_{12}^{(2)}} = \frac{\partial e_1^{(3)}}{\partial a_1^{(2)}} \frac{\partial a_1^{(2)}}{\partial z_1^{(2)}} \frac{\partial z_1^{(2)}}{\partial w_{12}^{(2)}}$$
$$\frac{\partial E}{\partial w_{21}^{(2)}} = \frac{\partial e_2^{(3)}}{\partial a_2^{(2)}} \frac{\partial a_2^{(2)}}{\partial z_2^{(2)}} \frac{\partial z_2^{(2)}}{\partial w_{21}^{(2)}}$$
$$\frac{\partial E}{\partial w_{22}^{(2)}} = \frac{\partial e_2^{(3)}}{\partial a_2^{(2)}} \frac{\partial a_2^{(2)}}{\partial z_2^{(2)}} \frac{\partial z_2^{(2)}}{\partial w_{22}^{(2)}}$$



Source: <https://medium.com/aimonks/understanding-backpropagation-the-engine-behind-neural-network-learning-a7c2e1acdbf>

BACKPROPAGATION vs GRADIENT DESCENT

	Backpropagation	Gradient Descent
Definition	An algorithm for calculating the gradients of the cost function	Optimization algorithm used to find the weights that minimize the cost function
Requirements	Differentiation via the chain rule	<ul style="list-style-type: none">• Gradient via Backpropagation• Learning rate
Process	Propagating the error backwards and calculating the gradient of the error function with respect to the weights	Descending down the cost function until the minimum point and find the corresponding weights

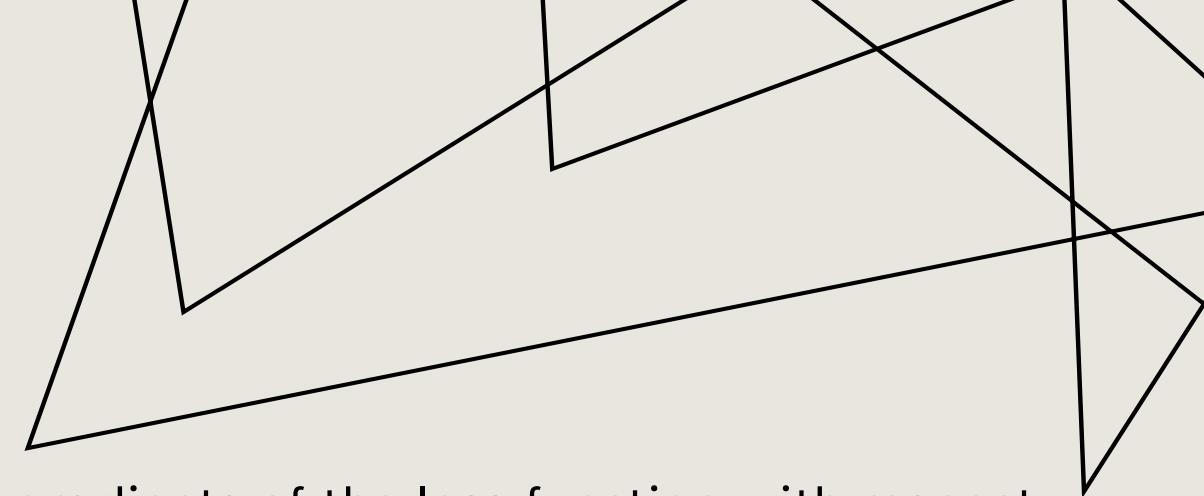


CHALLENGES WITH
DEEP LEARNING

CHALLENGES WITH DL

- **Data Quality and Quantity:** Deep learning models require massive datasets to train effectively. Without enough data, they tend to overfit and fail to generalize well to new data. Noisy, incomplete, or biased data can lead to poor performance, making data preprocessing crucial.
- **Model Interpretability:** Many deep learning models operate as “*black boxes*”, making it difficult to understand how they make decisions, and which features are significant. This lack of interpretability can be problematic in areas like healthcare or finance, where understanding the reasoning behind decisions is crucial.
- **Computational Resources:** Training deep learning models requires powerful hardware such as GPUs or TPUs, especially for large models or big datasets. Training can take days or even weeks, depending on the model size and hardware, making experimentation and tuning slower.
- **Overfitting:** Deep learning models have a large number of parameters, which makes them prone to overfitting, especially when trained on small or noisy datasets. Techniques like **dropout**, **data augmentation**, or **weight regularization** are needed to reduce overfitting.

CHALLENGES WITH DL: VANISHING GRADIENTS

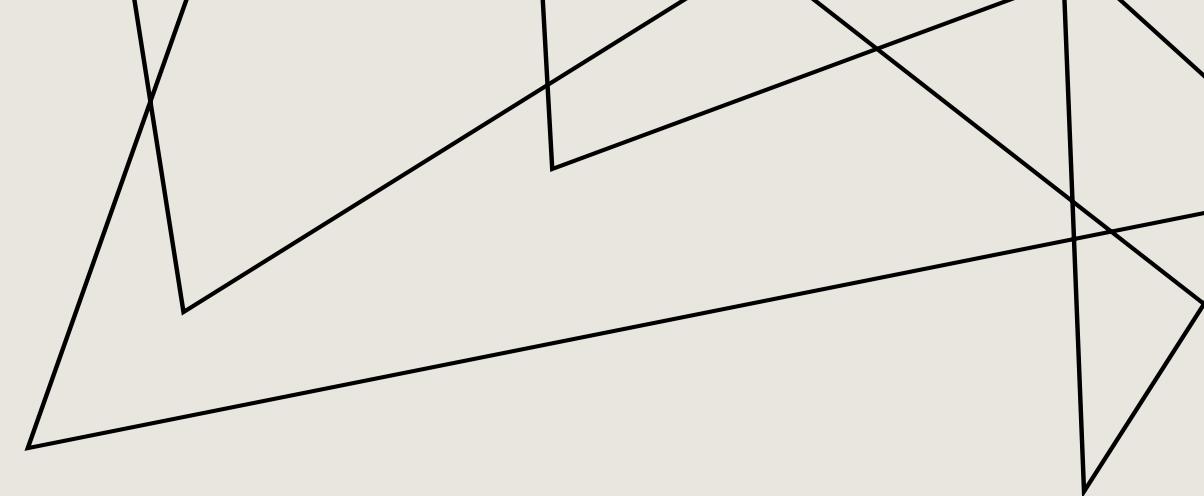


The **Vanishing Gradient** problem occurs when the gradients of the loss function with respect to the model's parameters become very small as they are propagated backward through the network. This leads to extremely slow updates in the early layers of the network, making it hard for the model to learn from the data. This causes the model to learn very slowly and may get stuck in local minima.

Causes

- **Activation Functions:** Non-linear activation functions like the sigmoid or tanh "squash" input values into a small range. When these functions are used, their derivatives are also small for large input values, causing gradients to shrink as they move back through layers.
- **Deep Networks:** As backpropagation proceeds through many layers, the multiplication of small derivatives in the chain rule can cause the gradients to diminish exponentially.

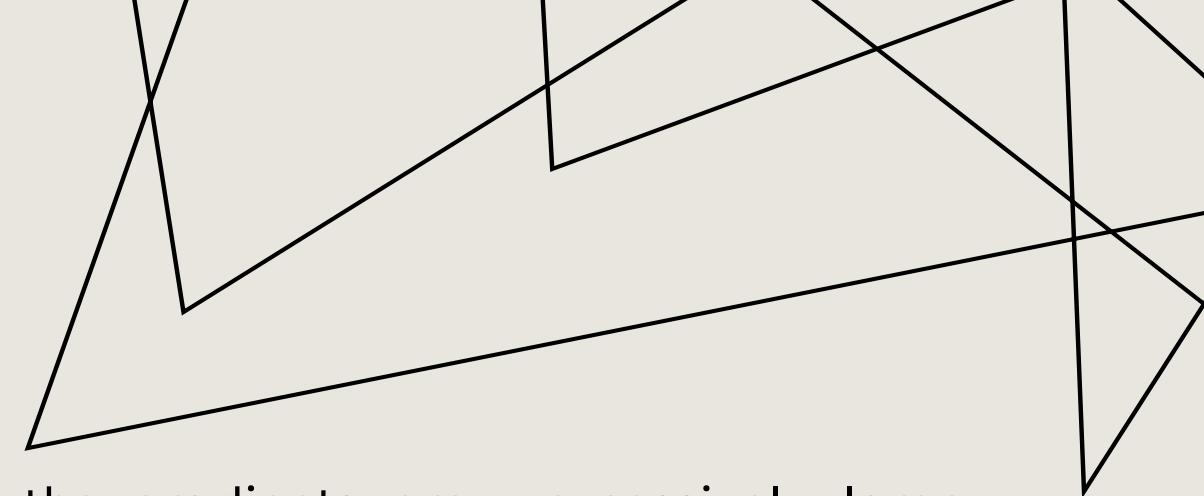
CHALLENGES WITH DL: VANISHING GRADIENTS



Solution

- **ReLU Activation:** ReLU (Rectified Linear Unit) and its variants (like Leaky ReLU) help mitigate this problem by not squashing the gradients to small values (ReLU keeps gradients at 1 for positive inputs).
- **Batch Normalization:** Normalizing the inputs of each layer helps stabilize training and reduce the impact of vanishing gradients.
- **Weight Initialization:** Proper initialization methods like Xavier (Glorot) initialization or He initialization can help prevent gradients from vanishing by keeping the variance of activations consistent throughout the network.
- **Skip Connections/Residual Networks:** Residual networks (ResNets) use skip connections to bypass layers, which helps gradients flow more freely and mitigates the vanishing gradient problem.

CHALLENGES WITH DL: EXPLODING GRADIENTS

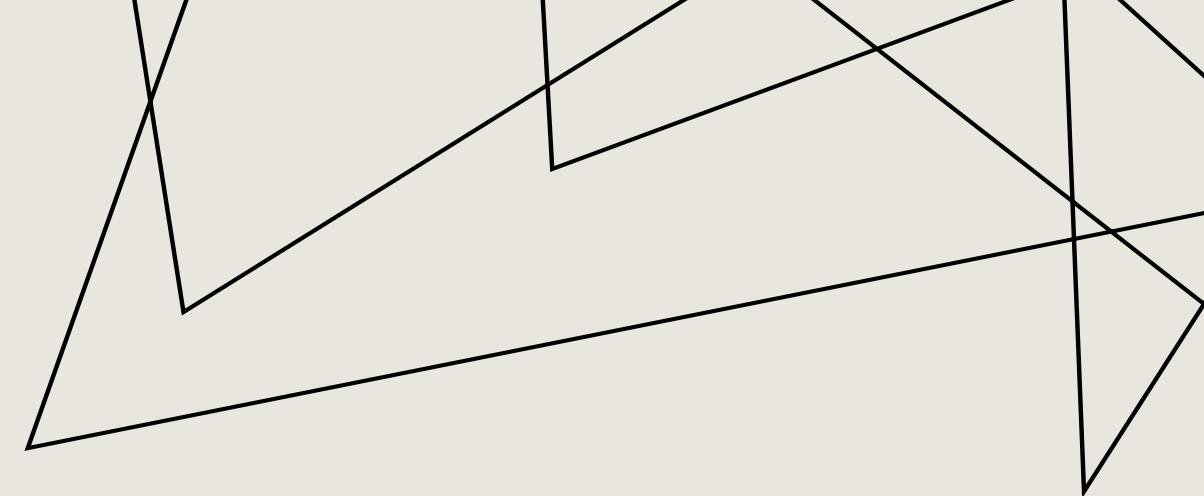


The **Exploding Gradient** problem occurs when the gradients grow excessively large during backpropagation, causing large weight updates. This can make the model unstable, and the training process can diverge instead of converging to a minimum. The large weights updates cause the loss function to oscillate and potentially diverge, making it impossible for the model to converge. The gradients can become so large that they lead to numerical instability, causing the model to fail or generate NaN values.

Causes:

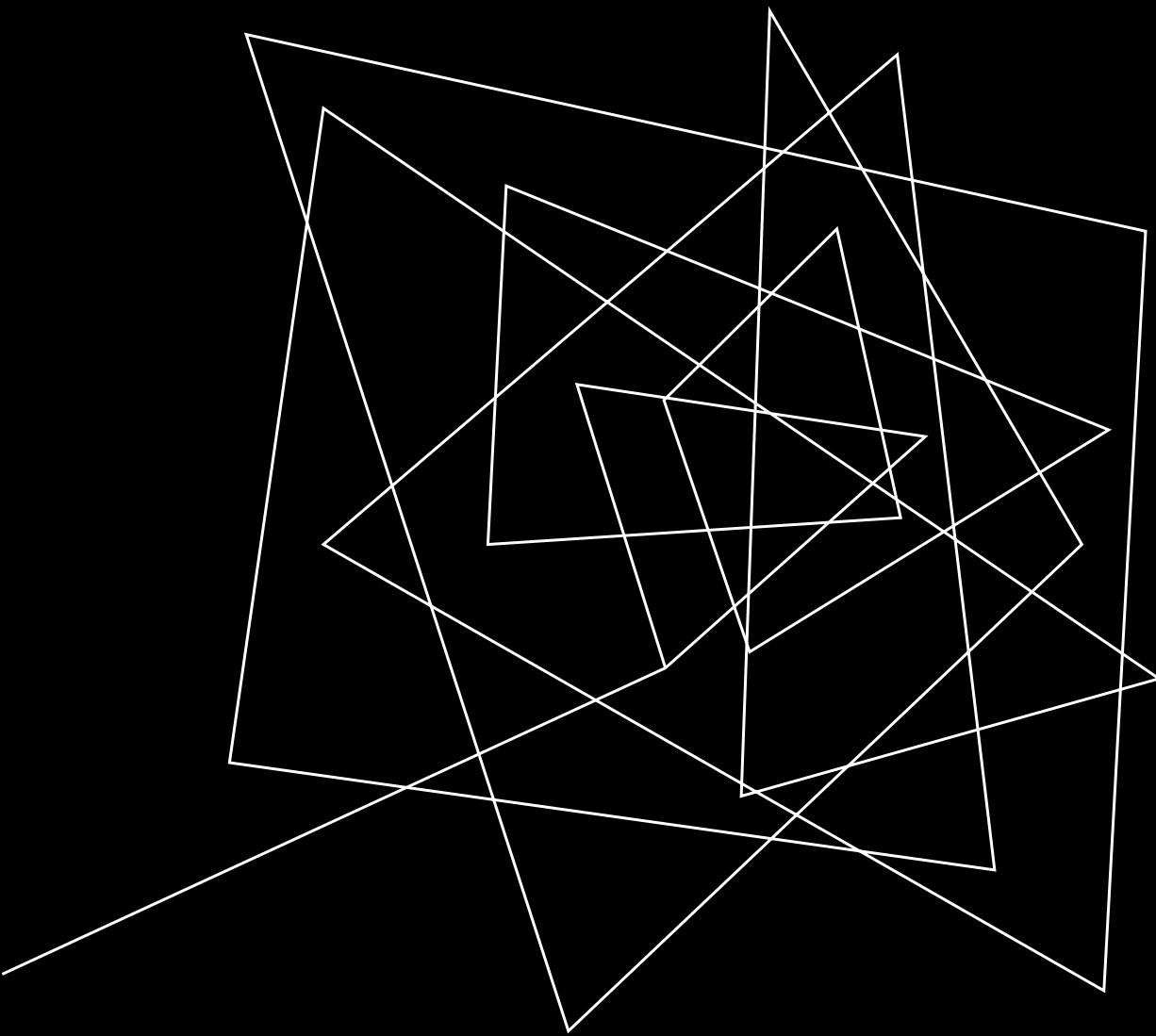
- **Weight Magnitude:** When weights in the network become too large, the computed gradients also grow exponentially as they are propagated backward through the layers.
- **Deep Networks:** As gradients are propagated backward through many layers, multiplying large numbers (gradients) can cause them to blow up.

CHALLENGES WITH DL: EXPLODING GRADIENTS



Solutions

- **Gradient Clipping:** By capping the gradients at a predefined threshold, **gradient clipping** ensures that gradients do not grow too large, keeping the training process stable.
- **Weight Regularization:** Techniques like L2 regularization (weight decay) penalize large weights, preventing them from growing too large and causing exploding gradients.
- **Normalized Initialization:** Using initialization techniques like Xavier or He helps prevent weights from becoming too large and gradients from exploding.
- **Smaller Learning Rates:** Reducing the learning rate can help prevent large updates that might exacerbate the problem of exploding gradients.



DROPOUT

GENERALIZATION IN NEURAL NETWORKS

Deep neural networks have many layers, and **each layer can learn increasingly complex representations**. While this enables the network to learn **intricate patterns**, it also means the model can easily memorize the training data, resulting in **poor generalization** to unseen data. Also, they contain a large number of parameters (weights and biases). This high capacity allows them to fit very complex patterns in the training data, including noise and irrelevant details, which leads to overfitting. Theoretically, A single hidden layer network can learn almost any relationship between features and target, given enough neurons and data. For these reasons, **Neural Networks are particularly Overfitting-Prone**.

Regularization methods like **L₂ Regularization**, **Data Augmentation**, **Early Stopping**, these techniques are often necessary in neural networks to ensure proper generalization. However, there are some techniques specific to Neural Networks that are very effective. One such technique is called **Dropout**. This method is very commonly used to counter Overfitting in Deep Neural Networks.

DROPOUT

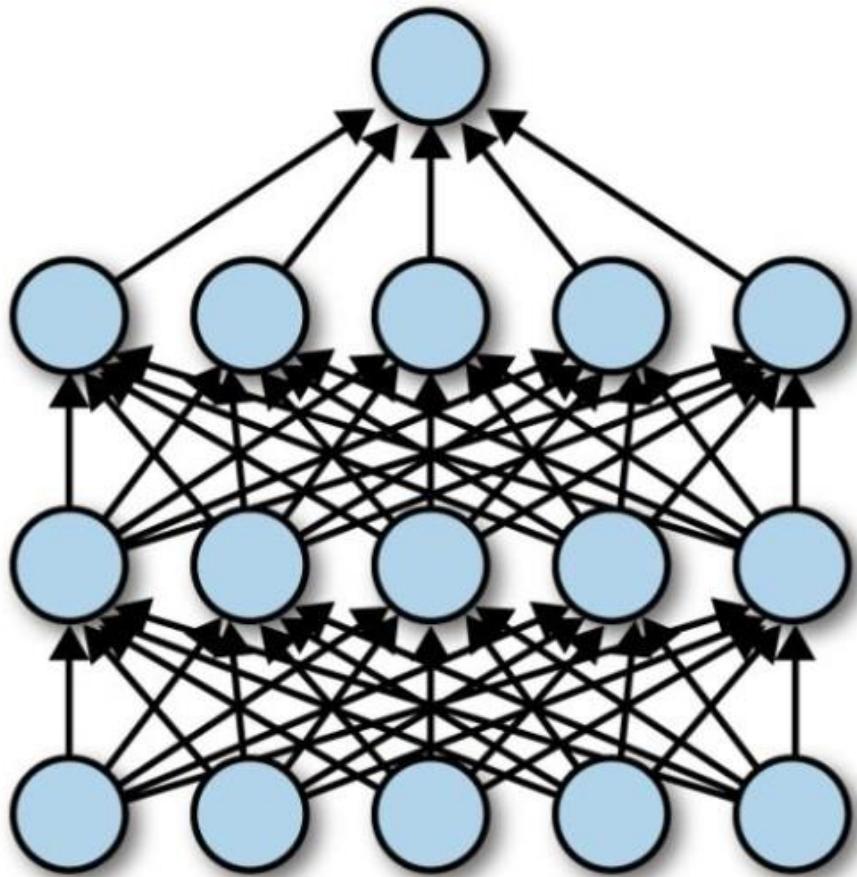
Dropout is a regularization technique used in neural networks to prevent overfitting. During training, dropout randomly "drops out" or deactivates a subset of neurons in a given layer with a certain probability. This forces the network to learn more robust features and prevents the model from becoming overly reliant on any single neuron, effectively creating many sub-models. For a neural network layer with 100 neurons, if a **dropout rate** of **0.5** is used, then during each iteration of training, only 50 randomly selected neurons will be active, and the remaining 50 will be “dropped”.

How Dropout Works

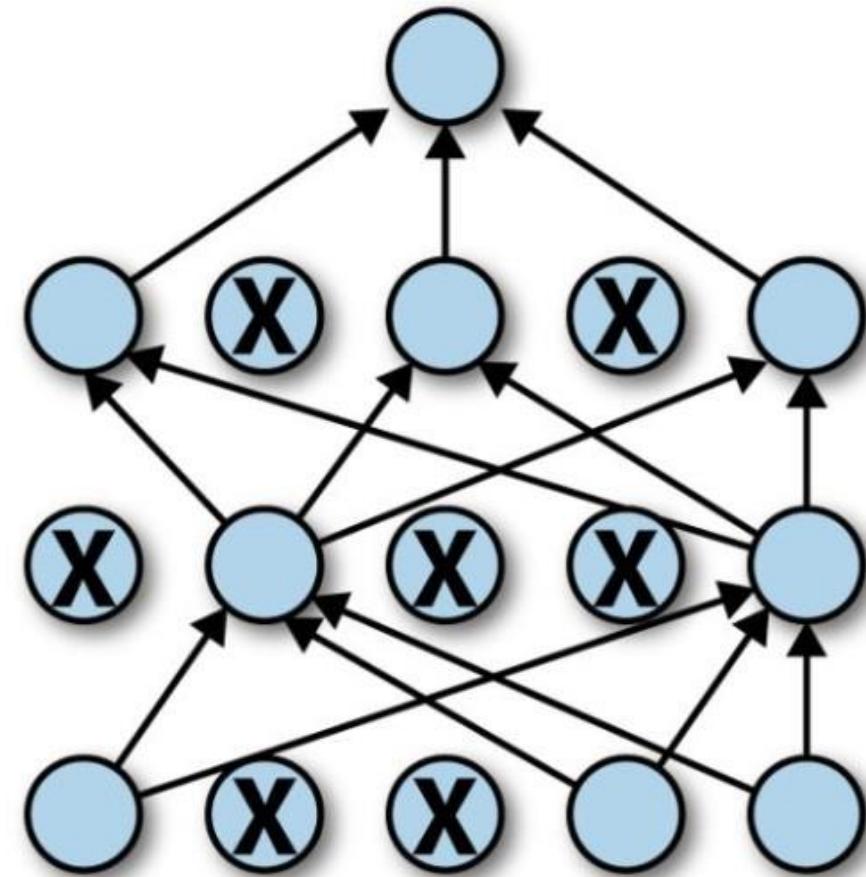
In each forward pass, a fraction of neurons are temporarily set to zero (dropped out). The probability of dropping out a neuron is controlled by a hyperparameter called the **dropout rate** (commonly denoted as **p**). *For example, if $p = 0.5$, 50% of the neurons are randomly deactivated during each iteration.*

During Testing: No neurons are dropped out, but the activations are scaled by the dropout rate (i.e., multiplied by **p**) to ensure consistency in the network's behavior between training and testing.

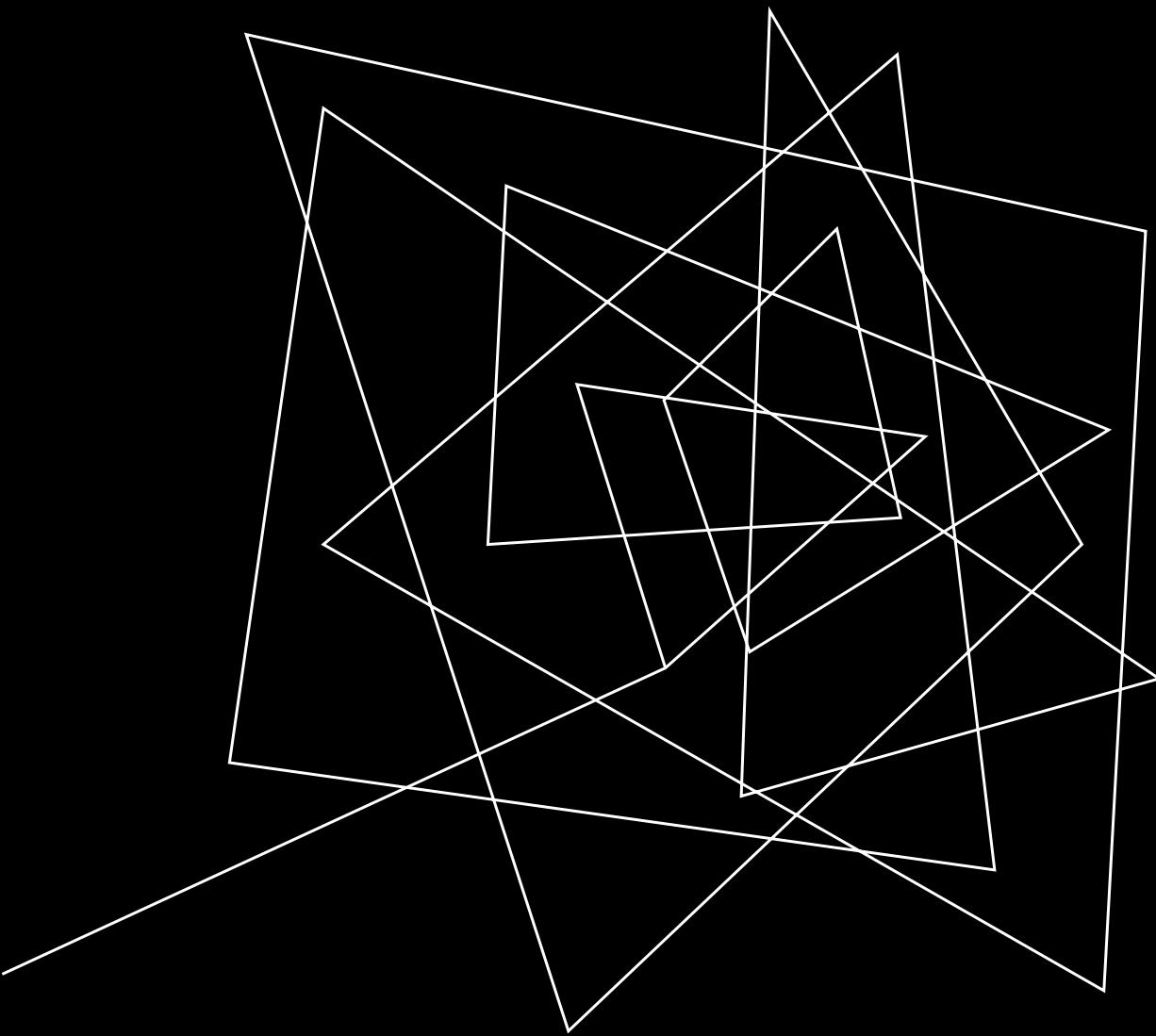
DROPOUT



Without Dropout



With Dropout



BATCH
NORMALIZATION

BATCH NORMALIZATION

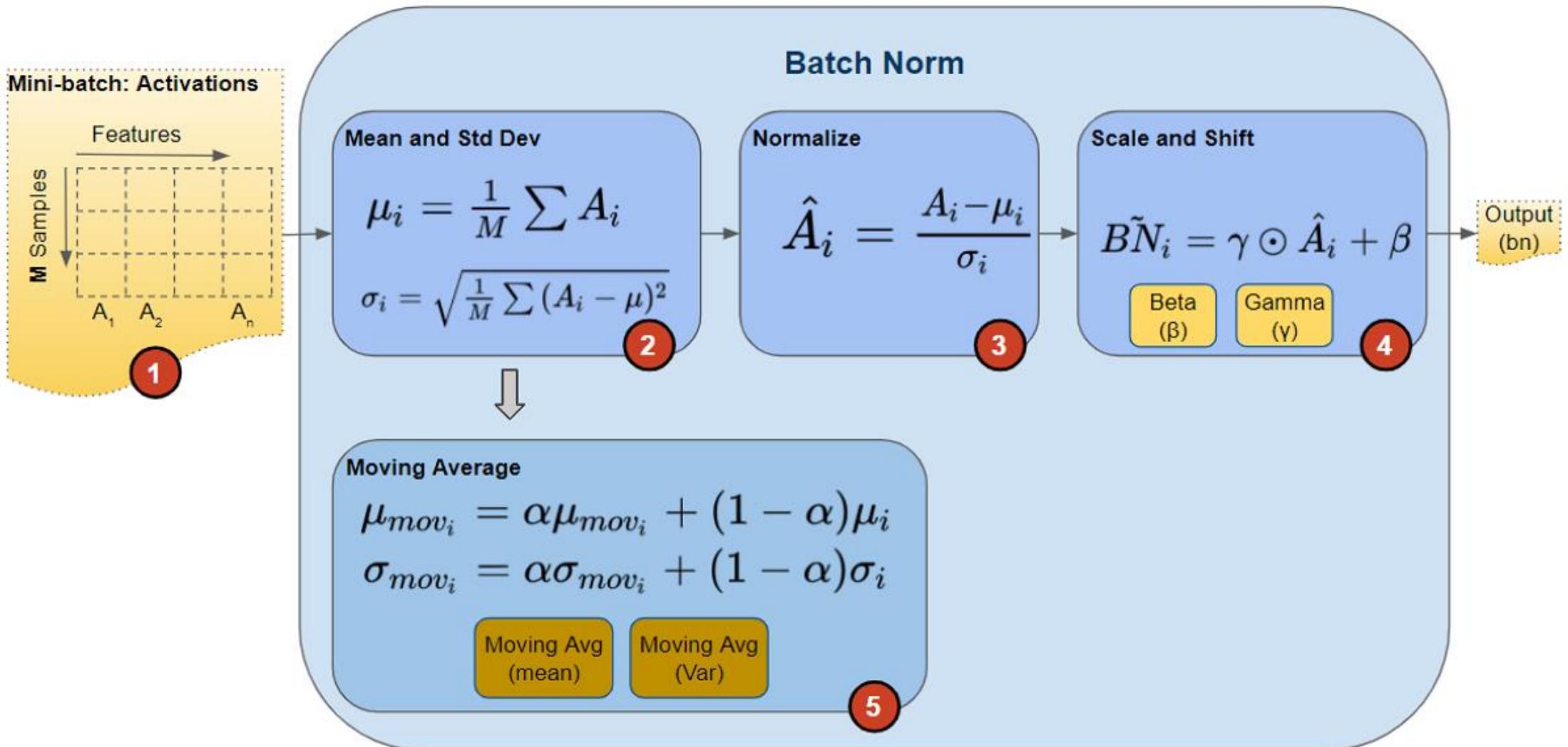
Batch Normalization is a technique used in deep learning to stabilize and accelerate the training process by normalizing the inputs of each layer. It helps in mitigating problems like vanishing and exploding gradients, making neural networks more robust, and faster to converge. *This is usually applied **after each layer** in a Neural Network.*

How Batch Normalization Works

- For each mini-batch, the **mean** and **variance** of the activations for each neuron are computed.
- The **activations**(Hidden Layer Outputs) are then **normalized** by subtracting the mean and dividing by the standard deviation.
- After normalization, a learnable **scaling factor γ** (gamma) and a learnable **shift factor β** (beta) are applied, allowing the network to undo the normalization if necessary. This flexibility ensures that the model can represent more complex patterns if required.

The activations can vary widely as they propagate through the network, leading to unstable gradients. This stabilization makes it easier for the optimizer to converge and allows the use of higher learning rates.

BATCH NORMALIZATION



Source: <https://towardsdatascience.com/batch-norm-explained-visually-how-it-works-and-why-neural-networks-need-it-b18919692739>

REFERENCES

1. Stanford cs231n Course Notes (<https://cs231n.github.io/>)
2. CMU's Introduction to Machine Learning (10-601) Lectures
(https://www.cs.cmu.edu/%7Etom/10701_sp11/lectures.shtml)
3. MIT OpenCourseWare: 6.867 Machine Learning
(<https://people.csail.mit.edu/dsontag/courses/ml16/>)
4. University of Toronto - CSC411/2515: Machine Learning and Data Mining
(https://www.cs.toronto.edu/~rgrosse/courses/csc311_f20/)
5. Applied ML course at Cornell and Cornell Tech
(<https://github.com/kuleshov/cornell-cs5785-2020-applied-ml/>)



THANK YOU