Introduction to Deep Learning

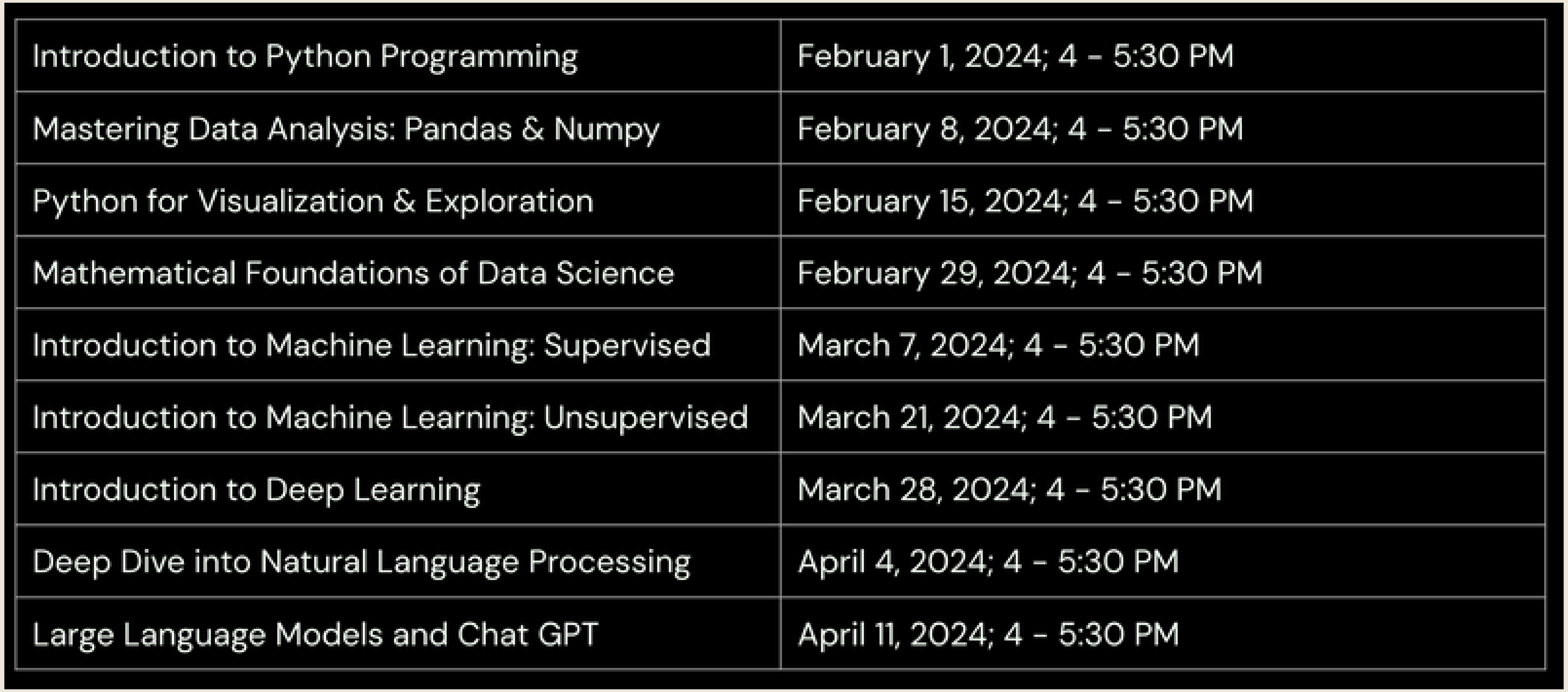
WORKSHOP DETAILS

Instructor: Tanya Khanna

Email: tk759@scarletmail.rutgers.edu

Course Materials: Github Link - https://github.com/Tanya-Khanna/DataScienceWorkshop_2024_NBL

WORKSHOPS SCHEDULE



Introduction to Python Programming	February 1, 2024; 4 - 5:30 PM
Mastering Data Analysis: Pandas & Numpy	February 8, 2024; 4 - 5:30 PM
Python for Visualization & Exploration	February 15, 2024; 4 - 5:30 PM
Mathematical Foundations of Data Science	February 29, 2024; 4 - 5:30 PM
Introduction to Machine Learning: Supervised	March 7, 2024; 4 - 5:30 PM
Introduction to Machine Learning: Unsupervised	March 21, 2024; 4 - 5:30 PM
Introduction to Deep Learning	March 28, 2024; 4 - 5:30 PM
Deep Dive into Natural Language Processing	April 4, 2024; 4 - 5:30 PM
Large Language Models and Chat GPT	April 11, 2024; 4 - 5:30 PM

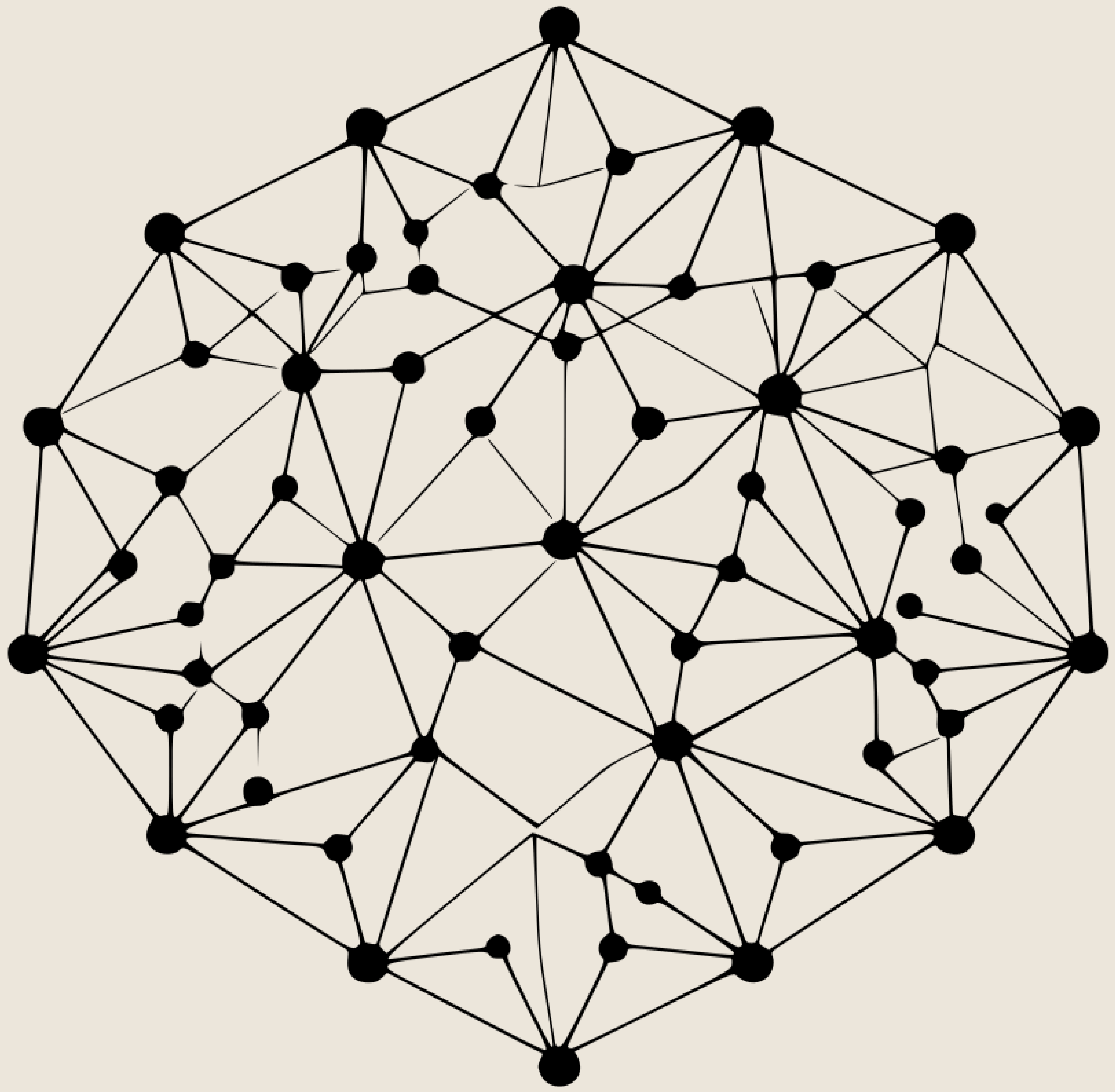
https://libcal.rutgers.edu/calendar/nblworkshops?cid=4537&t=d&d=0000-00-00&cal=4537&inc=0

TABLE OF CONTENTS

What is Deep Learning?

Deep learning and neural networks are integral to the advancements we've witnessed in artificial intelligence, fueling technologies behind some of the most innovative tools like ChatGPT, Stable Diffusion, and more.

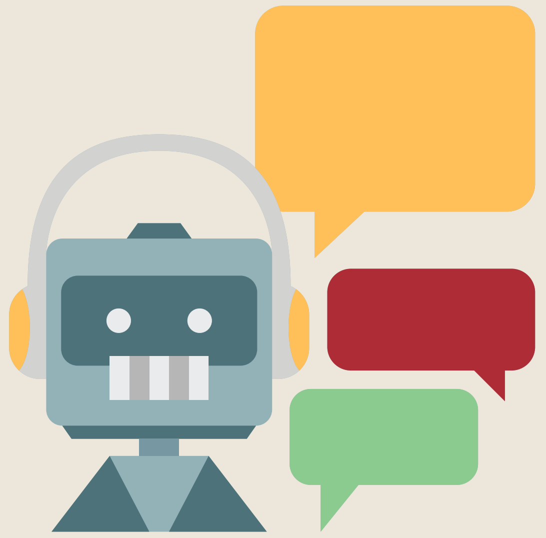
While machine learning has significantly advanced the ability of computers to interpret data, its effectiveness is often limited by the necessity of manual feature selection and its struggles with unstructured data. Deep learning overcomes these limitations by using neural networks that learn features directly from data, enabling the handling of complex, high-dimensional datasets with minimal human intervention. This self-learning capability allows deep learning models to achieve higher accuracy and adaptability in tasks like natural language understanding, speech recognition, and image classification, which are challenging for traditional machine learning models.



Uses of Deep Learning

From unlocking your phone with your face to asking a virtual assistant to play your favorite song, deep learning makes it happen.

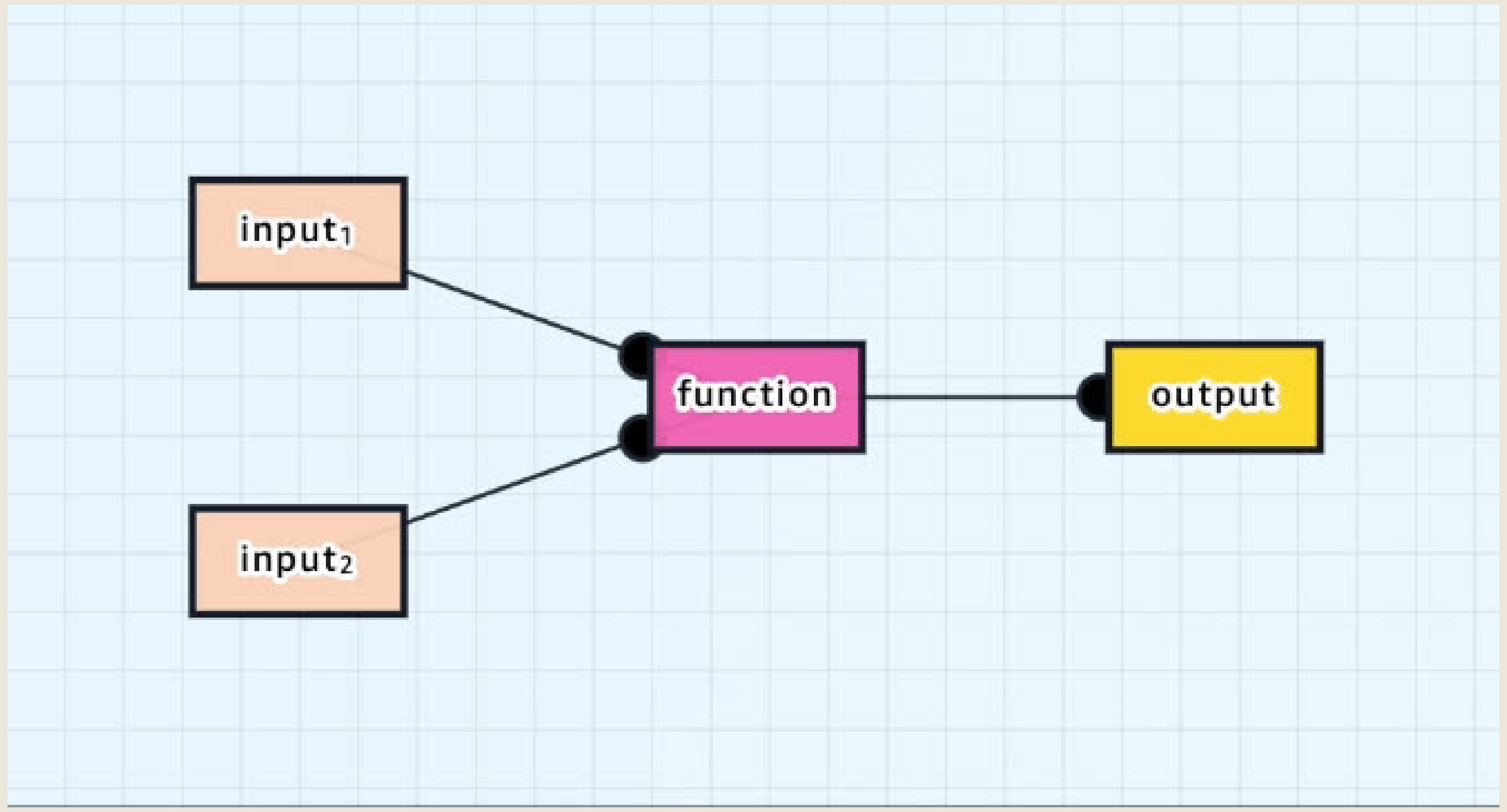
- **Chatbots and Translation Apps:** Ever wondered how chatbots understand and respond to you? Or how translation apps work? Deep learning helps computers understand and use human language. It's used in translating languages, sentiment analysis, and creating chatbots that understand human language nuances.
- **Self-driving Cars:** These cars use deep learning to see and understand the road, making safe driving decisions on their own. Deep learning models process the sensory data from vehicles to make real-time navigation decisions.
- **Healthcare:** Doctors use deep learning to better spot diseases in X-rays and scans, helping them catch things they might miss. These models analyze medical images to detect diseases like cancer with high accuracy, sometimes surpassing human experts.
- **Image and Speech Recognition:** Deep learning powers applications like facial recognition security systems and voice-controlled assistants (e.g., Siri, Alexa).



What are Neural Networks?

Deep learning uses neural networks with many layers (hence "deep") to analyze data, learn from it, and make decisions. The depth of these networks, which refers to the number of layers, allows for the processing of complex, high-dimensional data in ways simpler models cannot. This depth enables the model to learn features at various levels of abstraction, making deep learning particularly effective for tasks involving large amounts of unstructured data.

Neural networks are computing systems vaguely inspired by the biological neural networks of animal brains. They consist of layers of nodes, or neurons, each capable of performing simple computations. Data is processed through these layers, allowing the network to learn from the input data's features and patterns.



BUILDING BLOCKS: COMPUTATIONAL GRAPHS

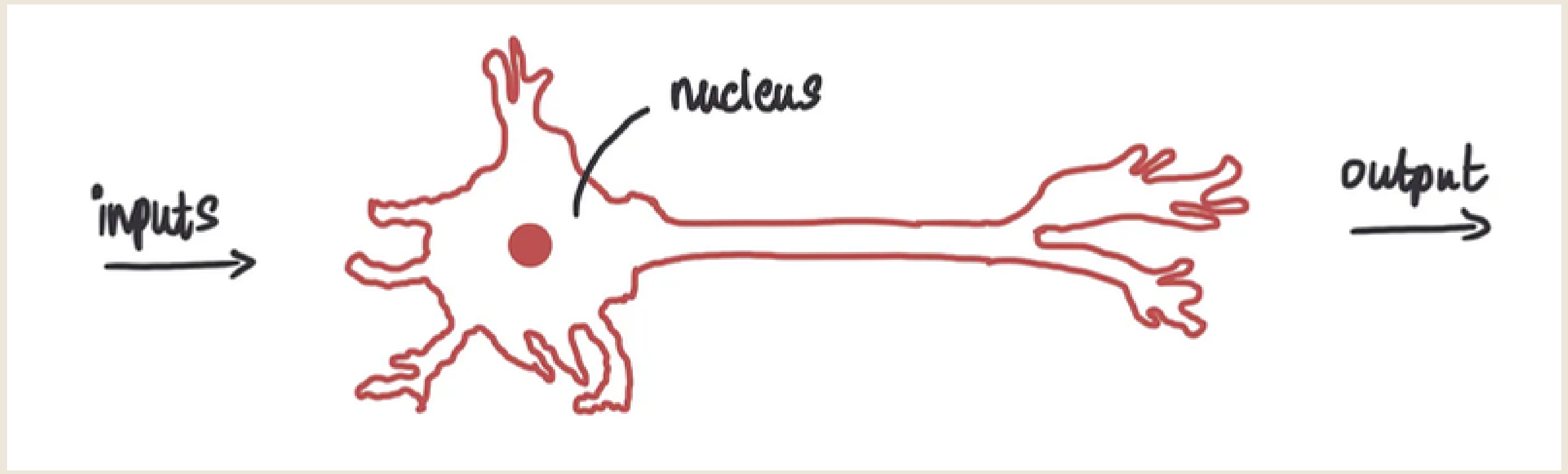
In a neural network context, this graph represents a single neuron with two inputs, where the function node encapsulates the neuron's activation process, and the output node is the neuron's output signal to subsequent layers or as a final prediction.

Neural networks are networks - that much is clear. But what is a "network"? A network is a structure consisting of interconnected computational nodes, or 'neurons', arranged in layers. These nodes perform mathematical operations on input data, learning some underlying patterns in the data, before producing some output based on those patterns.

Some Background

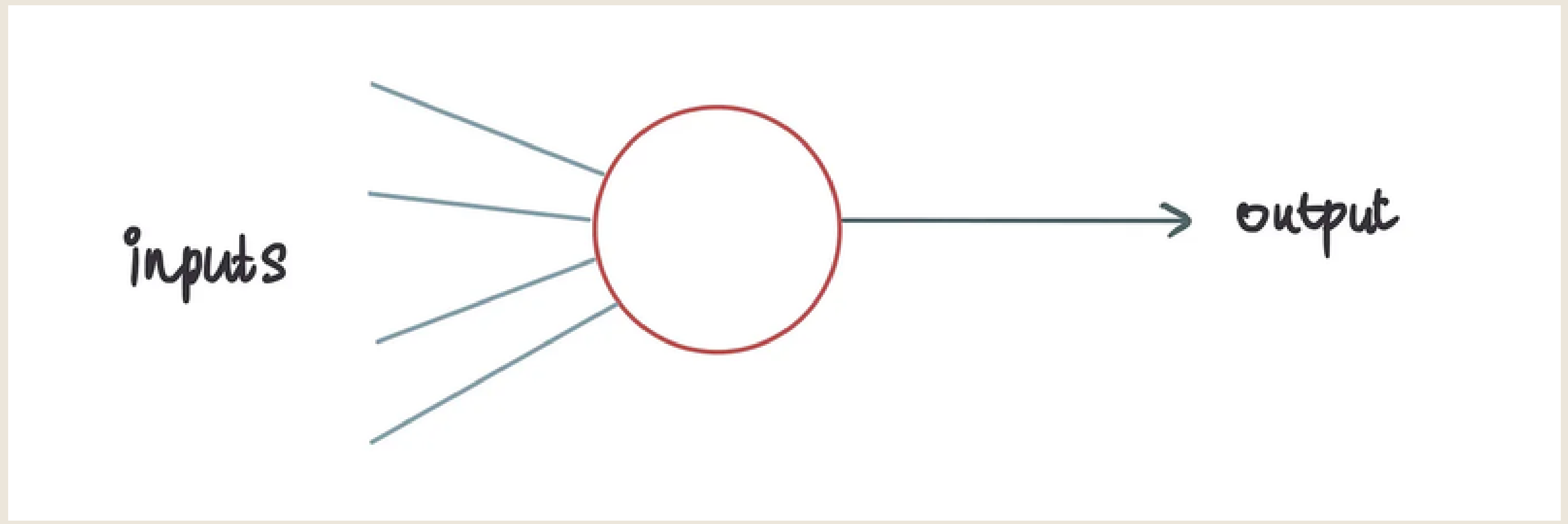
A neural network enables computers to process data in a manner inspired by the human brain. It utilizes interconnected neurons arranged in layers, resembling the structure of the human brain.

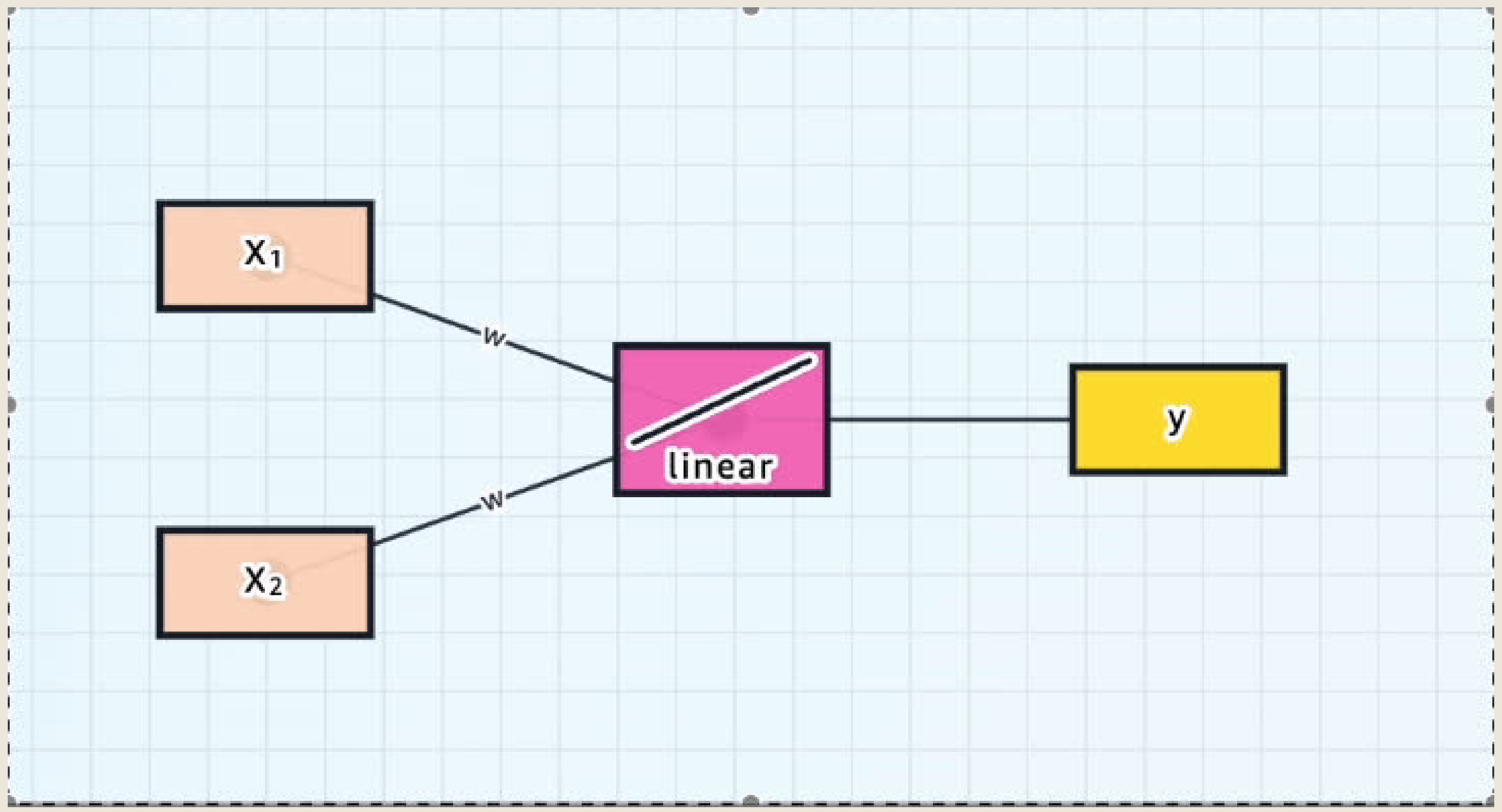
This is a biological neuron.



It receives inputs, processes the received inputs or data (this processing is nothing short of magical), and generates an output.

Just like the human brain, which processes data by receiving inputs and generating outputs, the neural network operates similarly.





Computational graphs are versatile and can model various algorithms, including simple ones like **linear regression**. The equation for linear regression can be visually laid out in a computational graph:

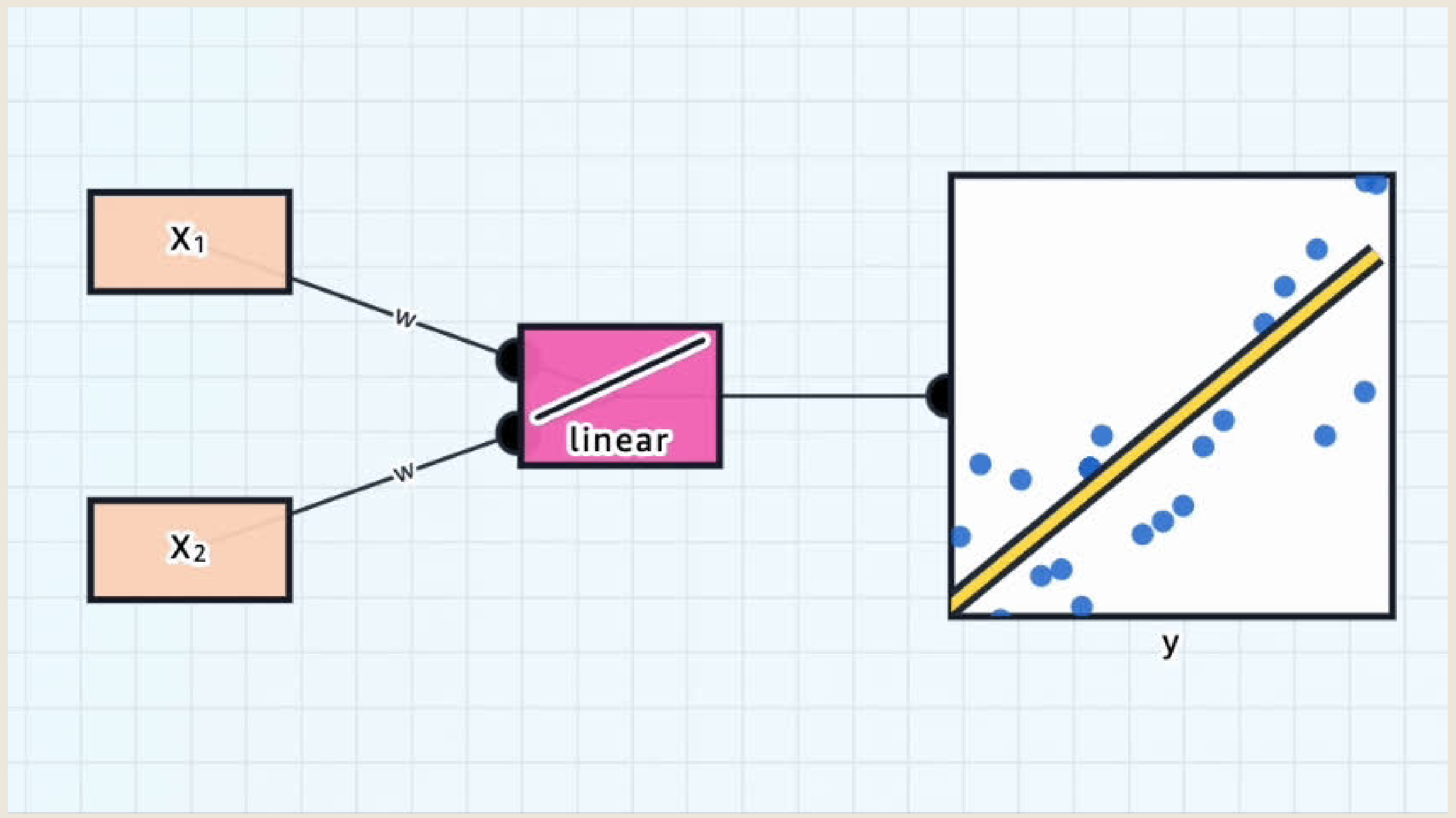
$$y = w0 + w1 * x1$$

To do so, we need only change two things:

1. Assign weights to the connections that lead from the input nodes to the function node. Each of these connections represents a feature's weight in the regression equation.
2. Modify the function within the function node to calculate a linear combination of the inputs. The function node sums the product of each input and its corresponding weight:

$$\text{linear} = \Sigma\,(wi * xi)$$

By making these changes—we can transform our simple computational graph into one that represents linear regression. This graph now shows how linear regression takes input values, applies weights to them, combines them in a linear function, and produces an output, which is the prediction 'y'.



Visualizing the outputs of a linear regression model helps clarify its functionality. Typically, when we graph the predictions of this model against the input data, we get a straight line (often colored yellow for visibility) that best fits the data points. This line represents the model's prediction for the output value 'y' given different input values 'x1'.

As we introduce complexity into the model—by adding more features, using a different algorithm, or tuning parameters—the nature of the output visualization will also change. For example, with a polynomial regression, instead of a straight line, the prediction line could curve to fit the data.

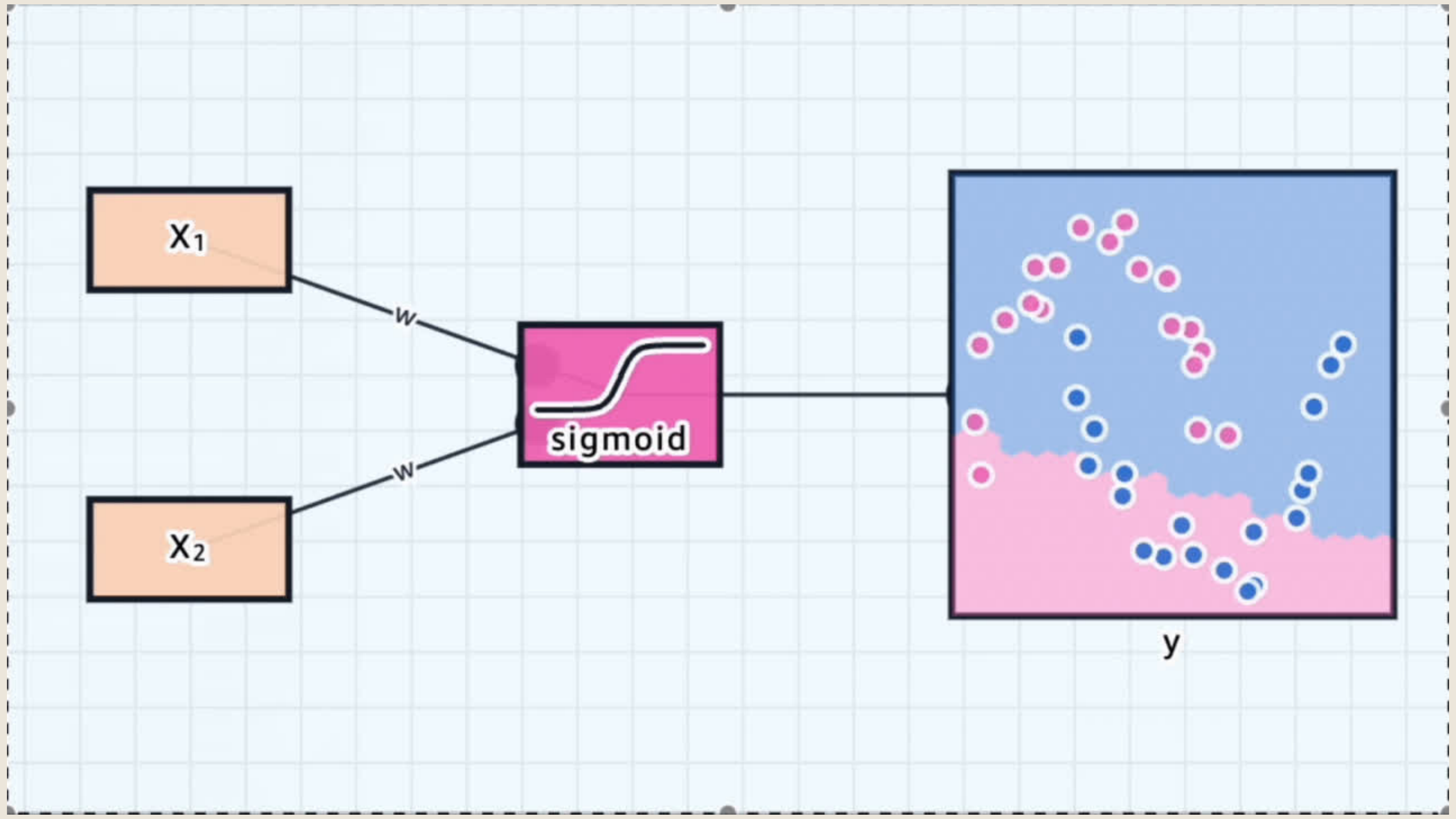
Should our task require a transition from predicting a continuous variable to classifying data into distinct categories, we can adapt our model from linear to **logistic regression**. The change reflects a move from predicting continuous outcomes to predicting categorical ones, like classifying whether an email is spam or not.

Alter the Computation at the Function Node: Replace the linear equation at the function node with a **sigmoid function.** The sigmoid function is an S-shaped curve that converts the weighted sum of inputs into a probability between 0 and 1:

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

where z is the weighted sum:

$$z = w_0 + w_1 x_1 + w_2 x_2 + \ldots + w_n x_n$$

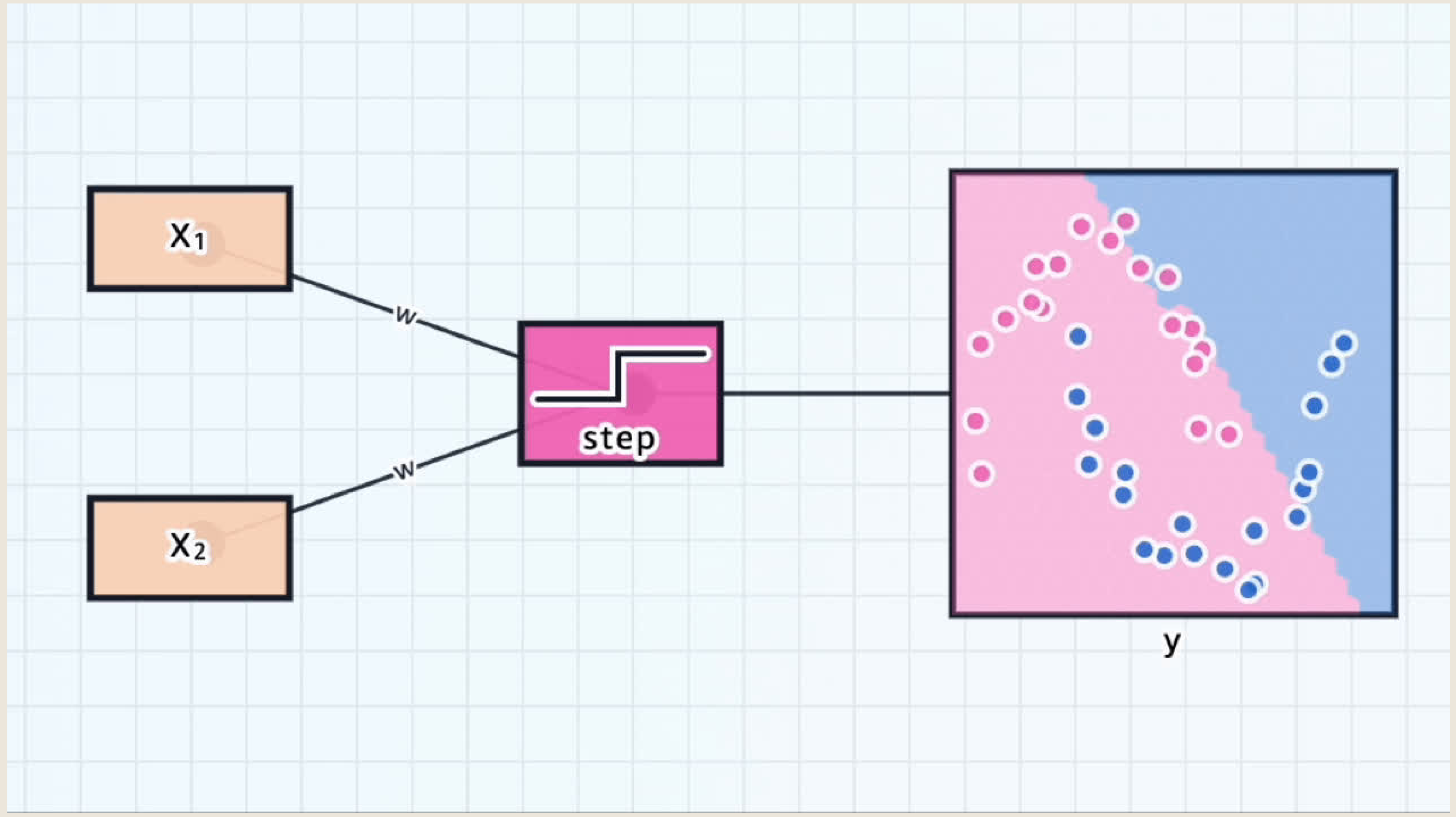


With the sigmoid function in place, the model's output is no longer a continuous value but a probability that signifies class membership. This is illustrated in a new way on our graph, where we now observe a decision boundary clearly delineating where the model transitions from predicting one class to another, typically at the 0.5 probability mark. The model's visualization will show areas indicating the likelihood of class membership

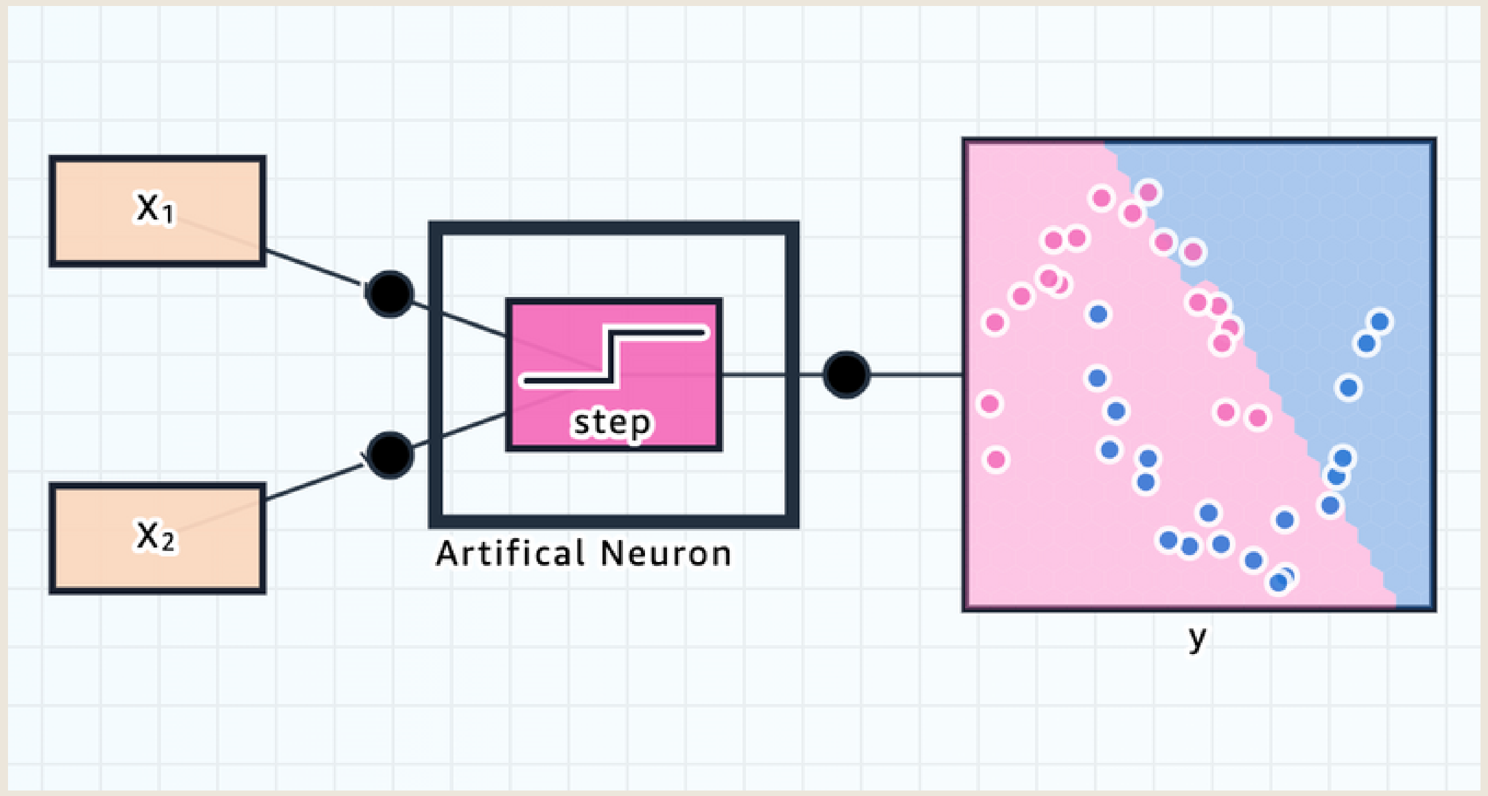
If the objective is to use a **perceptron** instead of logistic regression, the function applied at the computational graph's function node would change from a sigmoid to a **step function**. The step function produces a binary output, typically +1 or -1, based on the sign of the weighted sum of the inputs:

$$\text{step}(z) = \begin{cases} +1 & \text{if } \sum w_i x_i \geq 0 \\ -1 & \text{if } \sum w_i x_i < 0 \end{cases}$$

A perceptron is a type of artificial neuron or the simplest form of a neural network. It takes a set of inputs, multiplies each by a weight, sums them up, and then passes this sum through a step function to produce a single binary output. The perceptron is designed for binary classification tasks, predicting one of two possible classes.



ACTIVATION FUNCTIONS & ARTIFICIAL NEURONS



In a neural network, this function node we're changing is very special - we call it an artificial neuron.

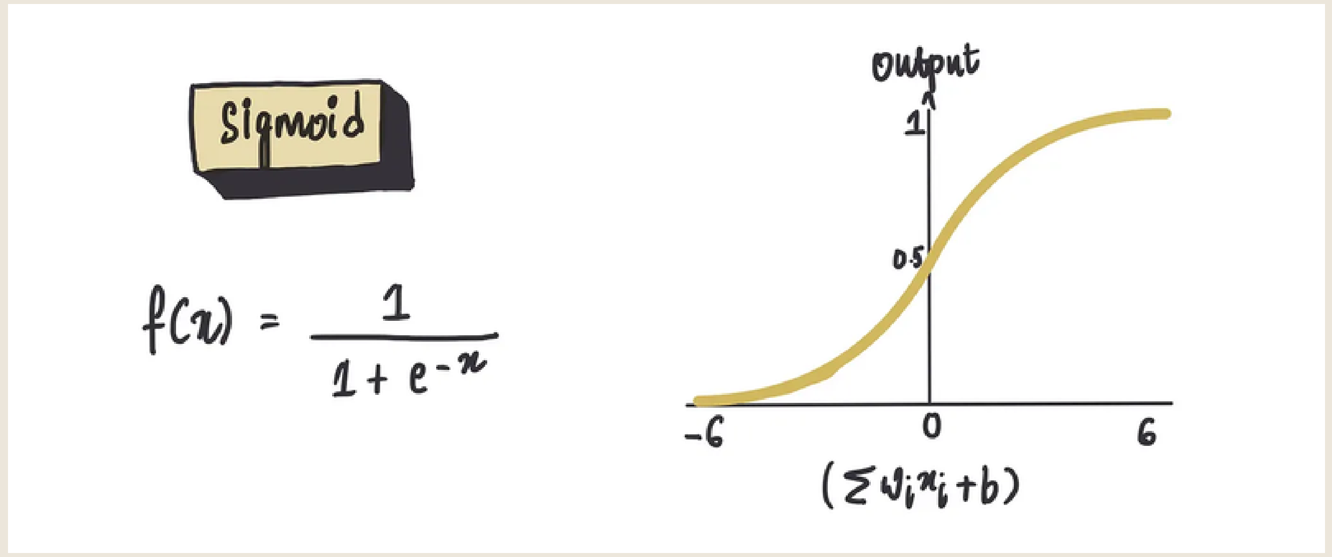
An artificial neuron is a fundamental computational element that receives inputs, performs a weighted operation on these inputs, and passes the result through a function.

In neural networks, these functions must be non-linear, and are referred to as activation functions.

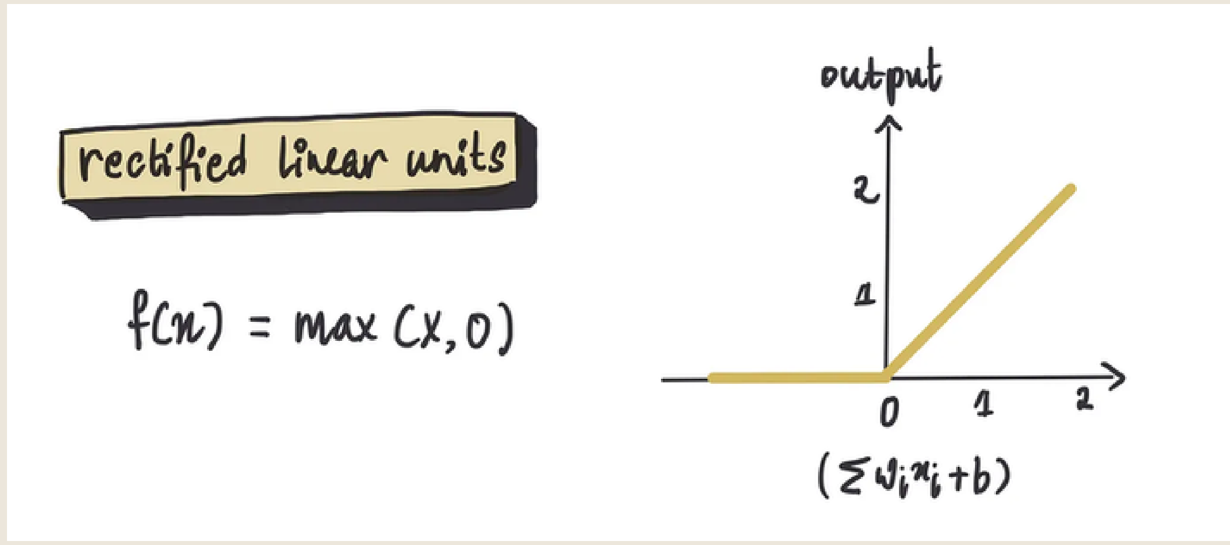
Activation functions are at the heart of artificial neurons in a neural network. These crucial components introduce non-linearity into the model, transforming the weighted inputs to generate an output. Simply put, an activation function decides how much signal to pass onto the next layer based on the input it receives. This idea of chaining many weighted signals together is what allows neural networks to learn very complex relationships.

The non-linear nature of these functions is essential for neural networks to learn from complex data. If we only used linear activation functions, no matter how many layers we stacked, the network would behave just like a single-layer perceptron because the composition of linear functions is still a linear function. This limits the complexity of tasks the network can learn. Non-linear activation functions, on the other hand, enable the network to learn complex patterns and solve intricate problems by adding layers of abstraction.

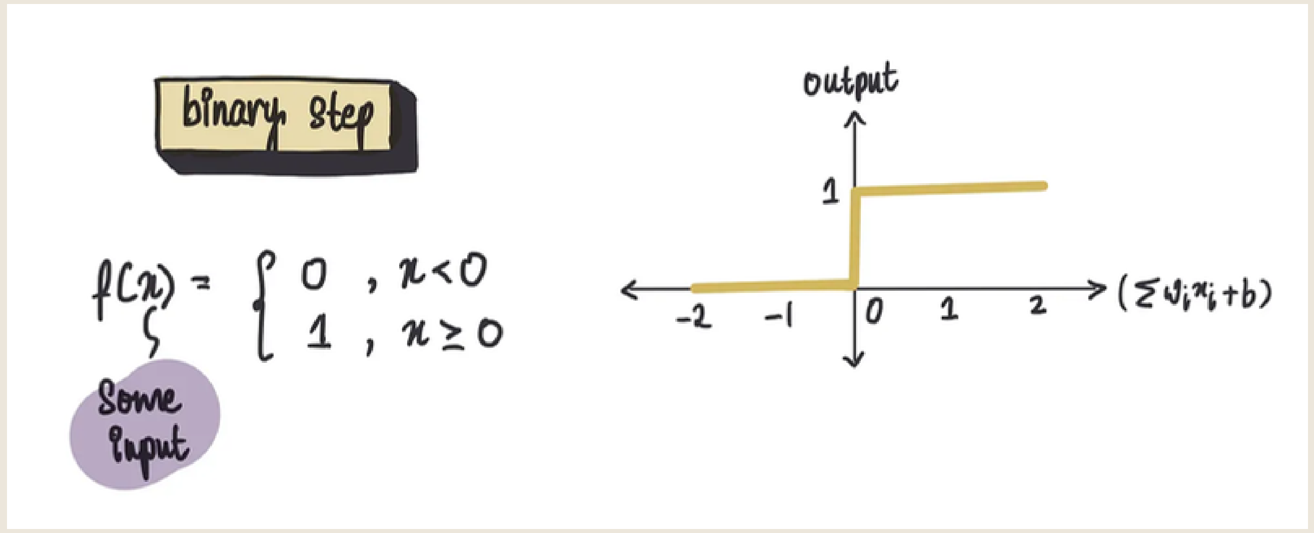
TYPES OF ACTIVATION FUNCTIONS



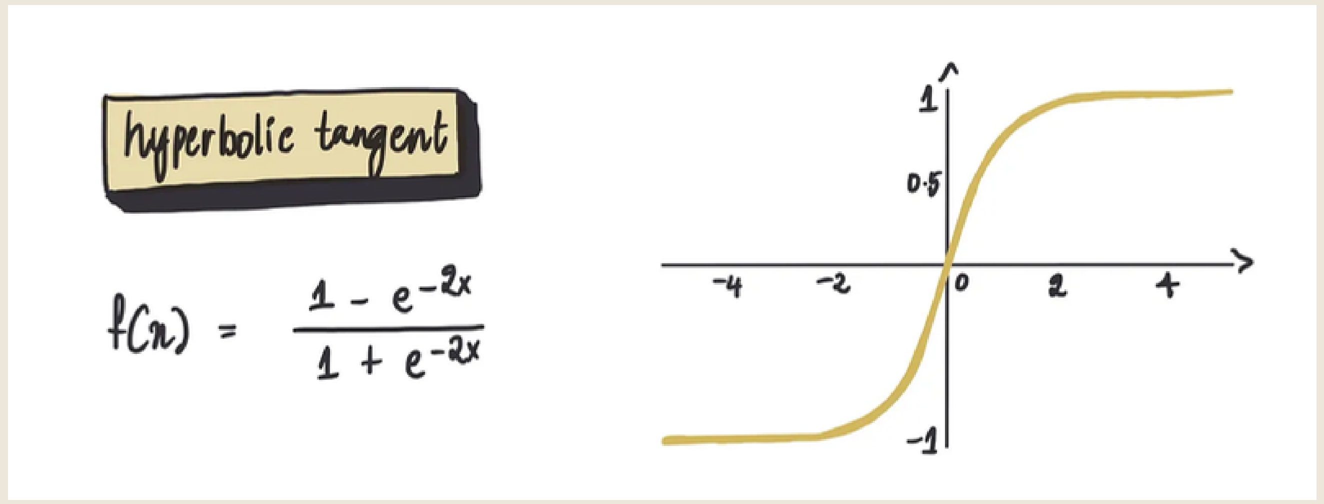
The sigmoid (or logistic) function, which ranges from 0 to 1, is particularly useful in the output layer of binary classification models, representing the probability of a binary event. Thus, it's perfect for probability-based questions. For example, what's the likelihood of a house selling given certain conditions?



The Rectified Linear Unit (or 'ReLU') function is a popular choice in hidden layers due to its efficiency. It activates a node if its input is positive, otherwise, it outputs zero. This simplicity reduces computational cost and mitigates the vanishing gradients problem, but it can lead to dead neurons where some neurons never activate.


binary step
$f(x) = \begin{cases} 0, & x<0 \\ 1, & x \geq 0 \end{cases}$
Some input
output
1
-2
-1
0
1
2
$(\sum w_i x_i + b)$

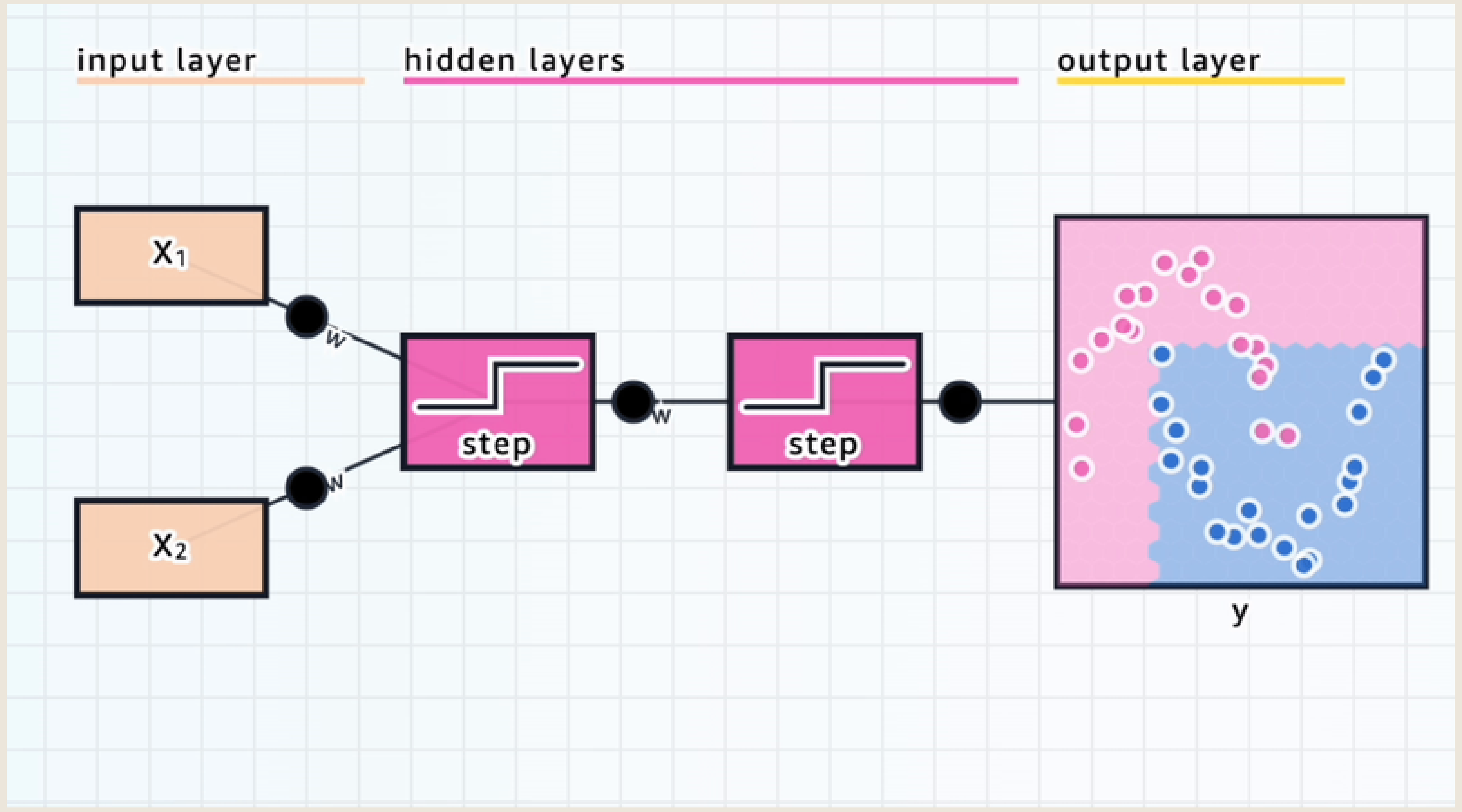
The binary step function, it's straightforward: if your input (let's call it x) is equal to or greater than 0, the function spits out a 1; otherwise, it gives you a 0. This is super handy when you need a clear-cut decision, like a yes or no. For example, based on the inputs will this house sell?



The hyperbolic tangent (or 'tanh') function, which ranges from -1 to 1, So, larger positive inputs hover near 1, while larger negative ones approach -1. It provides a zero-centered output designed to make learning for the next layer easier.

Despite their simplicity, chaining these functions together in a neural network can have magical results. That said, it's crucial to remember that there's no one-size-fits-all solution when it comes to choosing activation functions. The best choice often depends on the specific characteristics of the problem at hand, the nature of the input and output data, and the architecture of the network. For instance, if we're predicting something continuous, like the price of a house (a regression problem), the rectifier function is a great pick. It only gives positive outputs, aligning well with the fact that house prices aren't negative. But if we're estimating probabilities, like the chances of a house selling, the sigmoid function is our go-to, with its neat 0 to 1 range mirroring probability values.

FROM ARTIFICIAL NEURONS TO NEURAL NETWORK



Chaining multiple artificial neurons together, feeding one to another. **This is all a neural network is!** Thus, a neural network is essentially an assembly of artificial neurons linked sequentially, with the output of one serving as the input to the next.

Indeed, the earliest forms of neural networks were known as multilayer perceptrons. They were structured in successive layers of perceptrons (these are the artificial neurons that use step functions) that pass signals forward from one to the next. Typically, a neural network's structure includes three kinds of layers:

Input Layer: This is the initial layer, containing a node for each variable that the network uses as input.

Hidden Layers: These are one or more layers filled with artificial neurons that process the inputs from the previous layer.

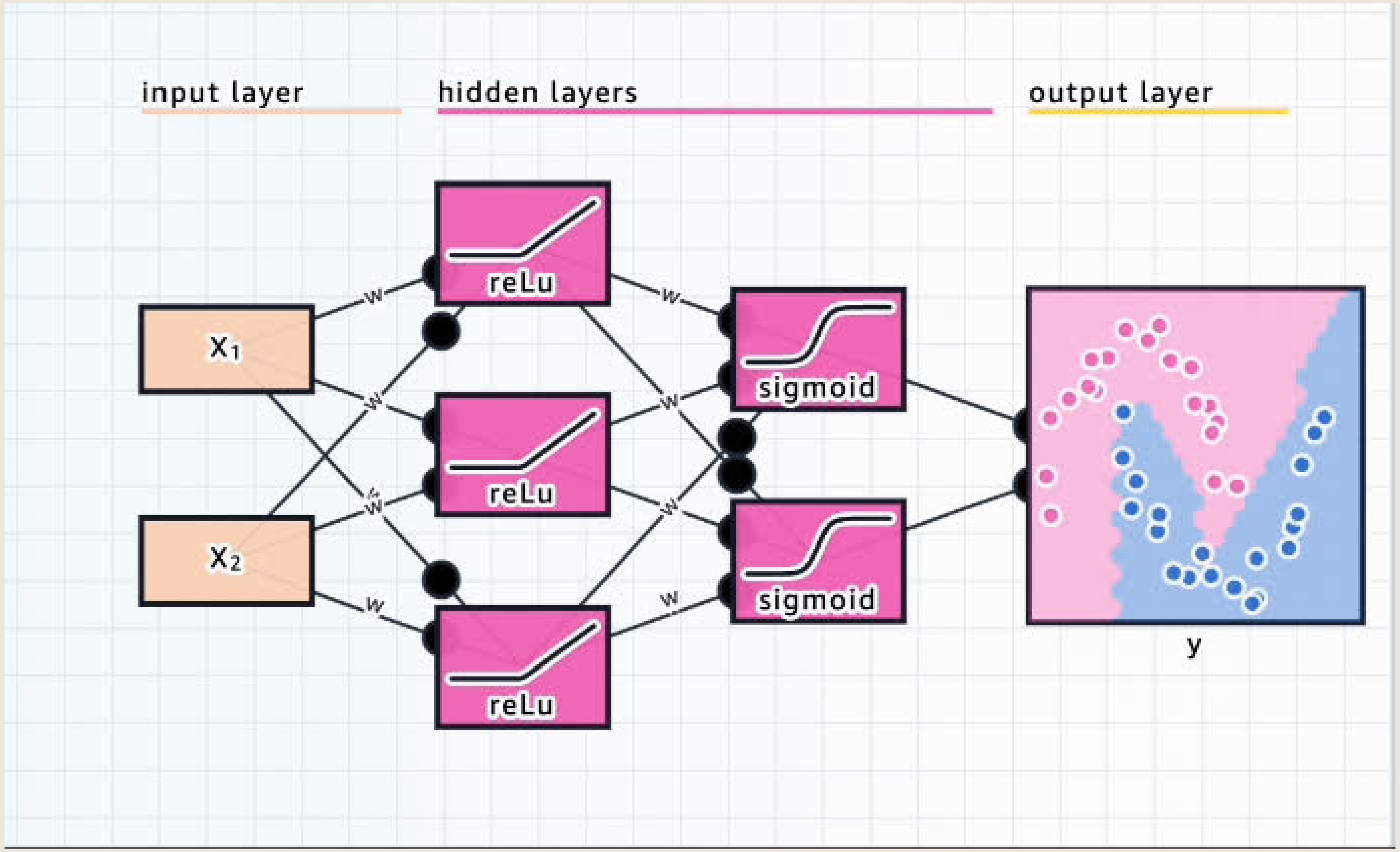
Output Layer: The final layer provides the network's prediction or classification.

A neural network is required to have one input and one output layer. However, the number of hidden layers can vary widely, with each layer adding depth and complexity to the model.

Designing a neural network architecture is more of an art than a science. The input and output layer will be selected for the specific problem, but the hidden layer is fairly arbitrary.

Neural networks can be **wide**: having many neurons in a given hidden layer, or **deep**: having many hidden layers in the network. Balancing neuron count optimizes performance; more neurons enable complex learning, at the cost of the risk of overfitting and more computational cost.

They are just computational graphs, channeling inputs through successive layers of computation to generate outputs. This process of inference, whereby inputs are fed through the network to produce output predictions, is called the **forward pass.**



HOW DO NEURAL NETWORKS LEARN?

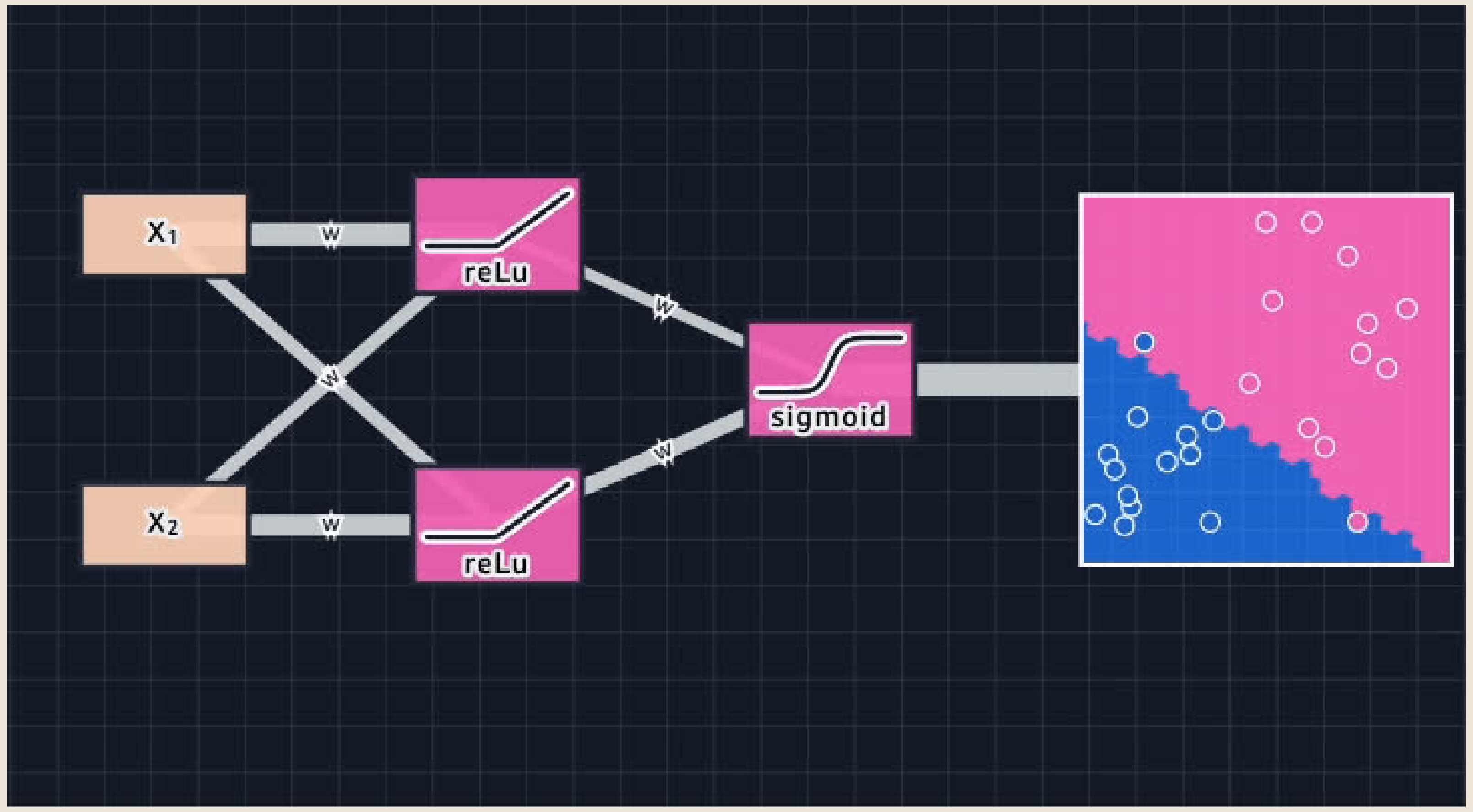
STEP 1: FORWARD PASS

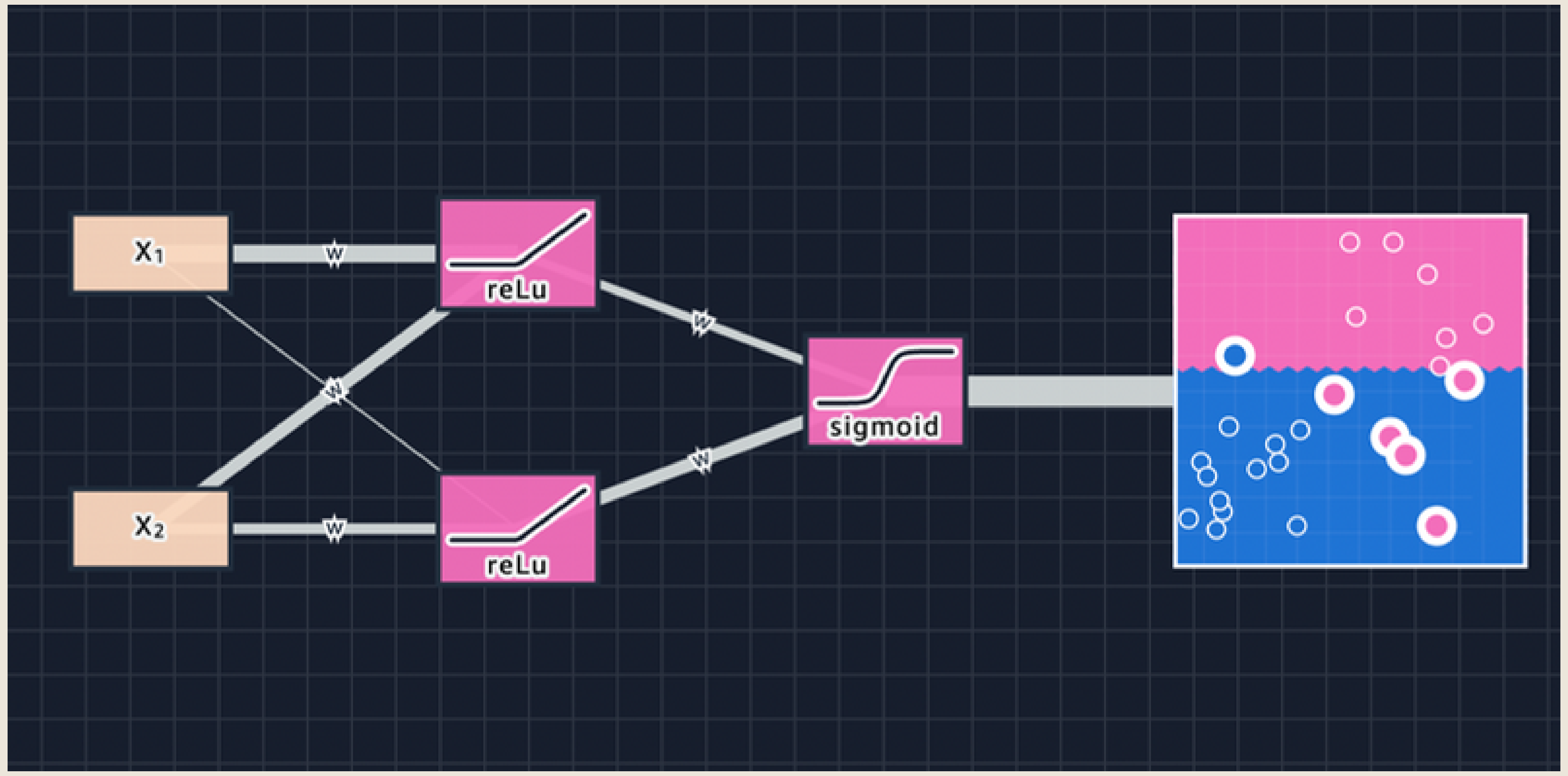
To understand how these networks learn - the magic behind this learning process is a technique known as backpropagation.

Backpropagation is an algorithm used during the training of neural networks. The goal of backpropagation is to update the weights so that the Neural Network makes better predictions. Specifically, backpropagation will calculate the gradient of the loss function with respect to the weights of the network, updating the weights layer-by-layer to minimize the network's prediction error.

During the forward pass, input data is fed through a neural network's layers to produce a prediction. This process involves calculating the weighted sums and applying activation functions for each neuron in each layer.

We can see that the classification boundary for our current network (the two-colored background region) isn't great.

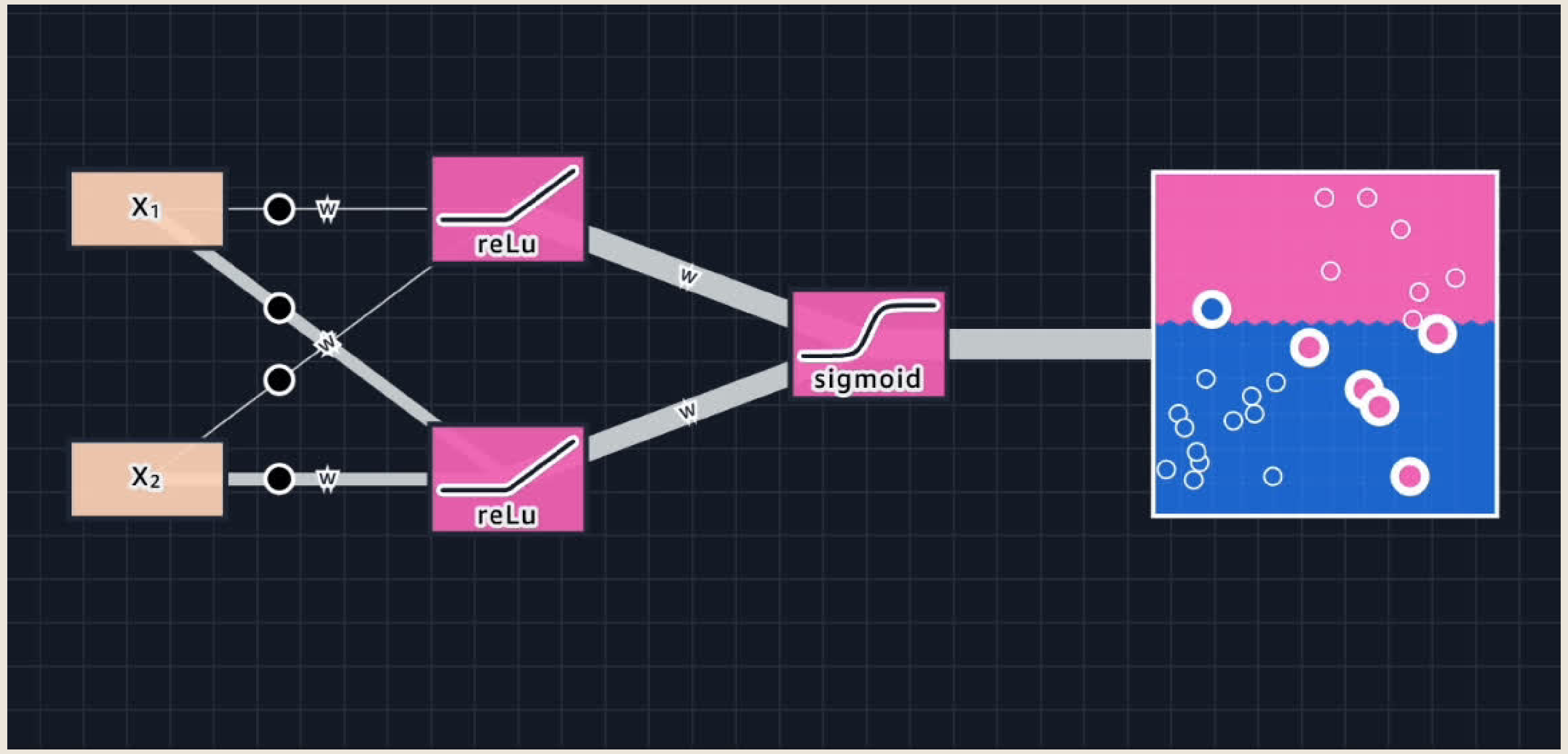




STEP 2: BACKWARD PASS

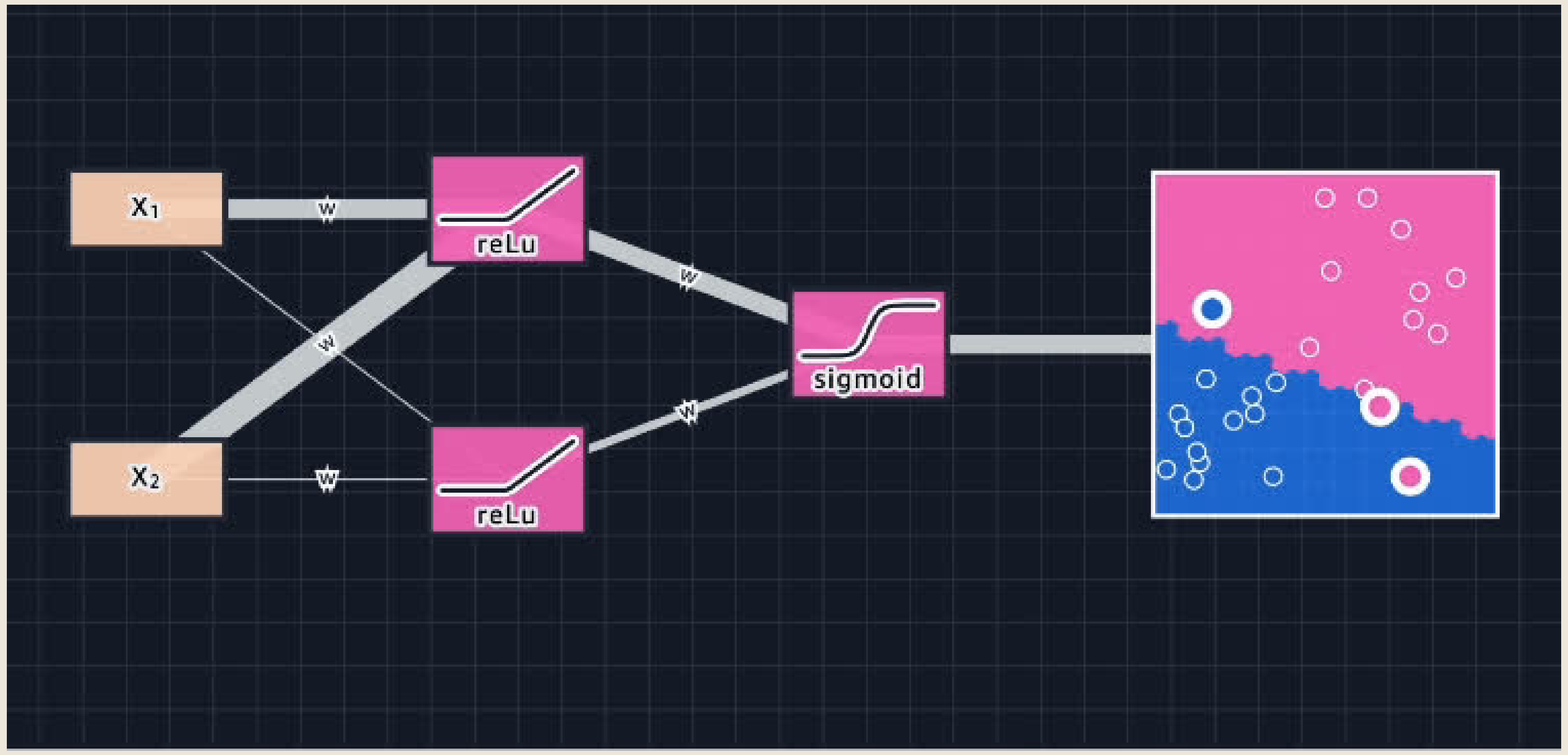
In the backward pass, the error is propagated back through the network, starting from the output layer, to adjust the weights and biases of each neuron. This weight adjustment process, guided by the gradients of the error with respect to the weights, aims to minimize the overall error of the network. (This process uses the Chain Rule from calculus to update the weights layer-by-layer)
In our example, the width of the lines connecting each neuron represent the weights of the network. The weights will be adjusted to reduce the prediction error of our Neural network.

Notice those extra big circles in the chart, the ones that are colored differently than their background? Those are the values that were falsely predicted. The goal of backpropagation is to update the model's weights so that it learns to predict with less error. The error, also known as loss, measures the difference between the neural network's predicted output and the actual target values. By minimizing this error, the neural network learns to improve its predictions during training.

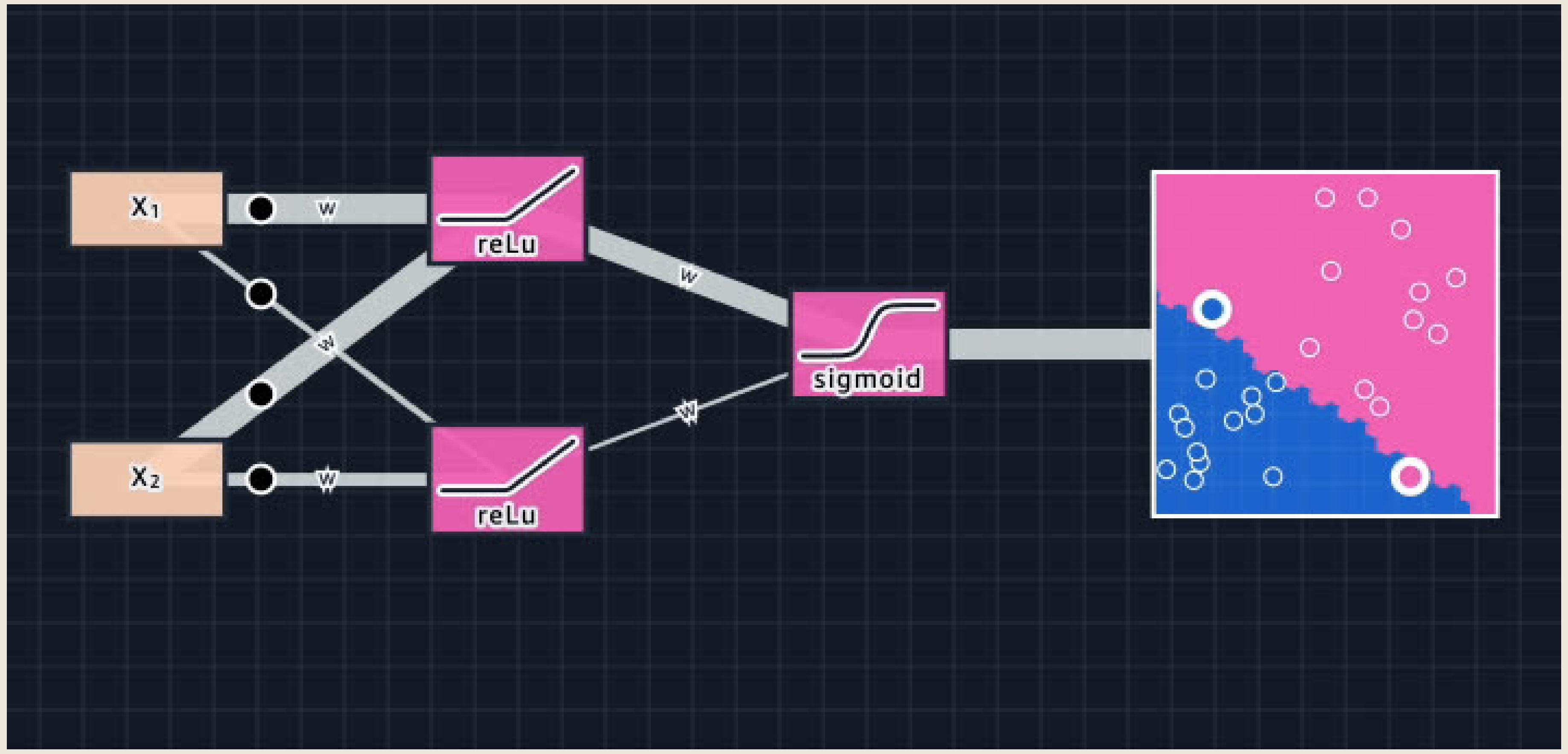


STEP 3: BACKPROPOGATION

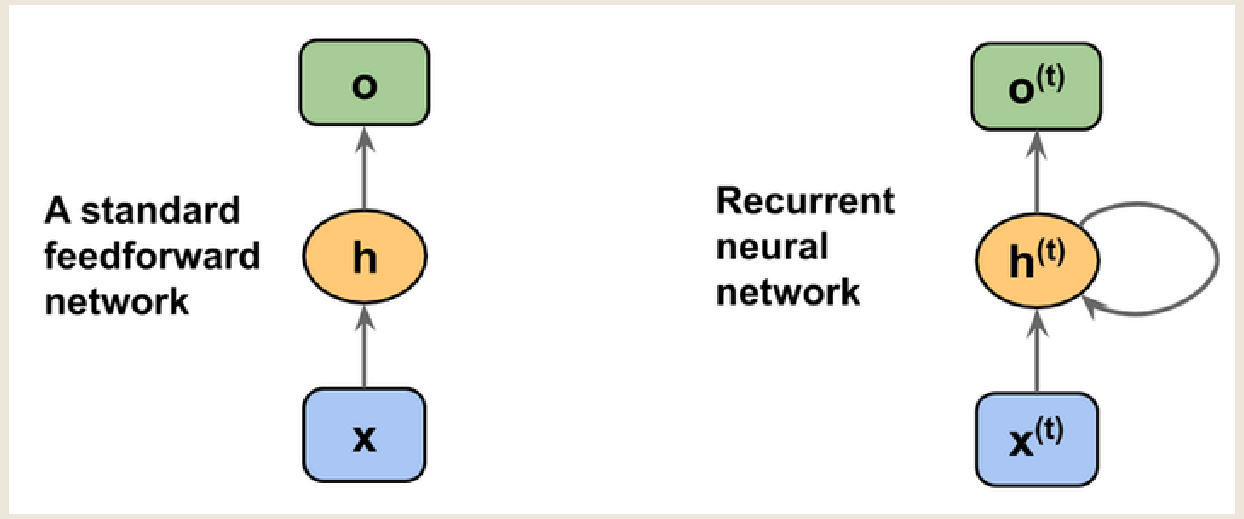
Predictions flow forward through the network in the forward pass, and errors flow backwards through the network, adjusting weights along the way. This process is know as Backpropagation. Backpropagation is a supervised learning algorithm used in neural networks that optimizes their weights and biases through iterative forward and backward passes. By computing the gradients of the error with respect to the weights, backpropagation enables the network to learn and improve its predictions. Thus, it is the process by which Neural Networks try to find the optimal weights for the given prediction task (optimal here meaning the weights that result in the lowest error value).



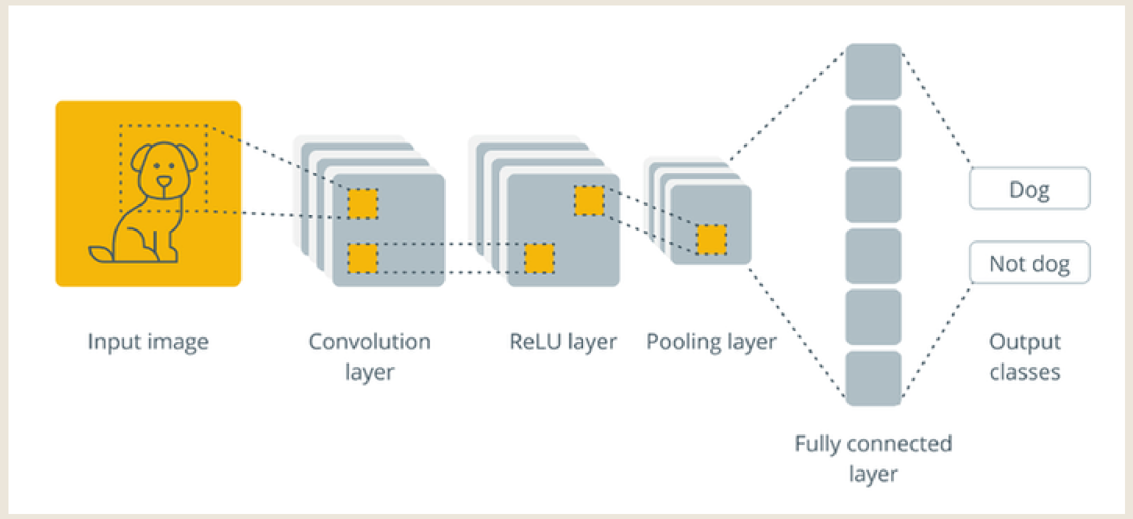
Backpropagation doesn't occur just once! For a typical neural network, backpropagation is repeated hundreds, if not thousands, of times. The key thing to note is that, at each run, the network's weights are updated in a manner that improves our model's performance!. To see this directly, you can observe how the classification region (the background colors) correspond correctly to the circles.



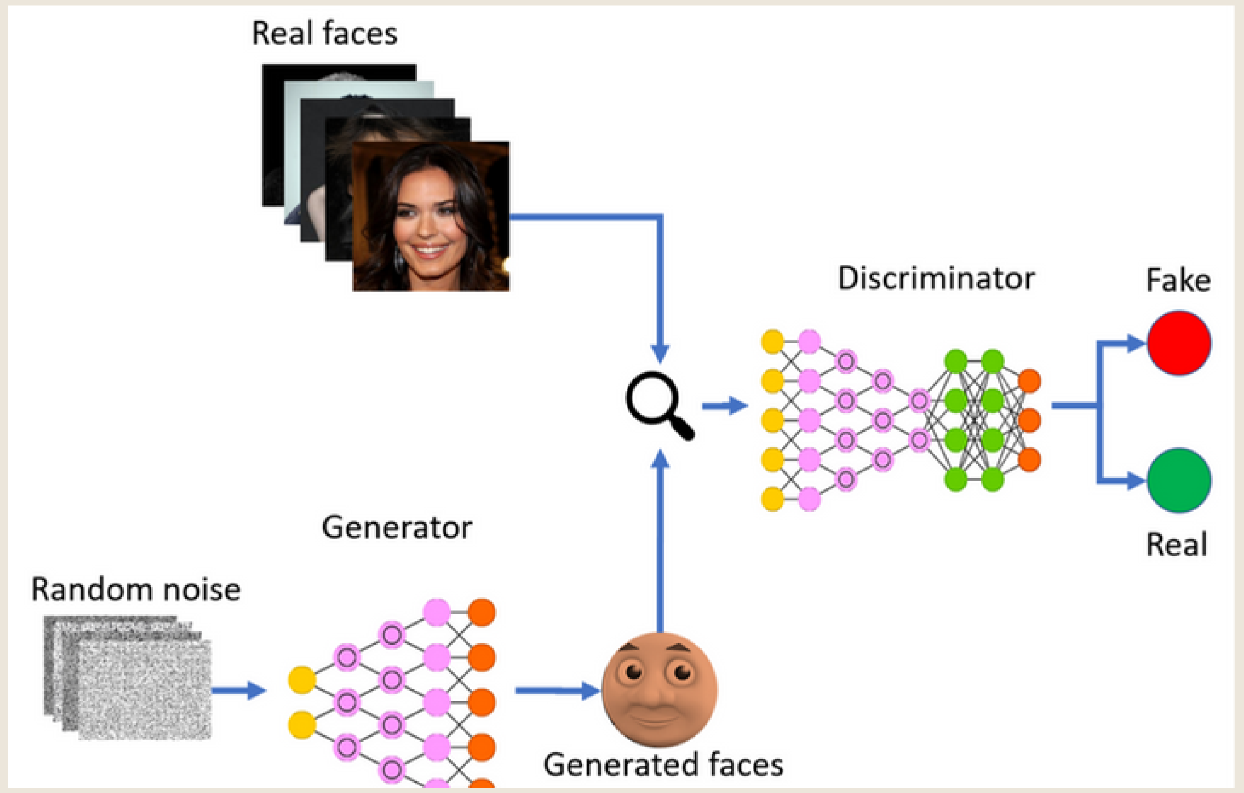
TYPES OF NEURAL NETWORKS



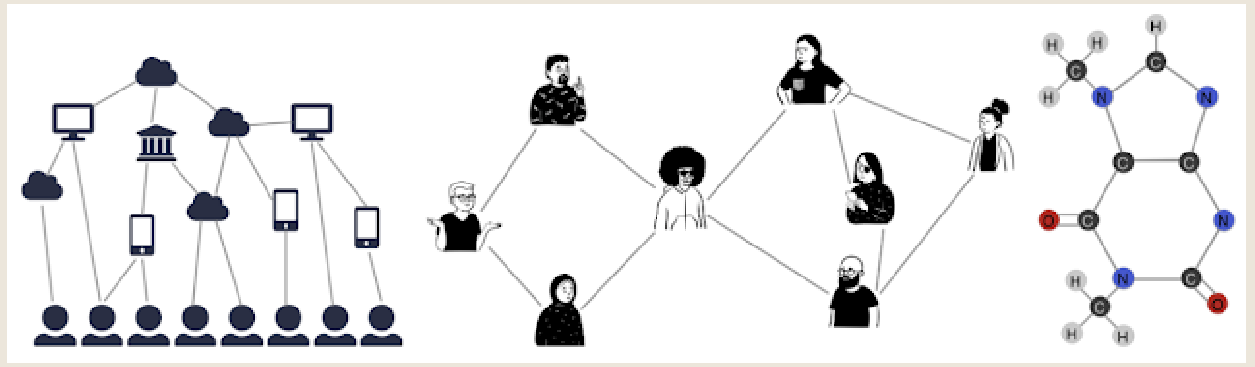
Recurrent Neural Networks (RNNs): Suited for sequential data, RNNs shine in natural language processing and time series analysis by using inherent memory to link past and future data points.



Convolutional Neural Networks (CNNs): Designed for spatial data, CNNs excel in image recognition and computer vision by identifying patterns through convolutional layers that scan inputs.



Generative Adversarial Networks (GANs): Comprising two networks, a generator and a discriminator, GANs learn to create new data mimicking real datasets, useful in image generation and style transfer.



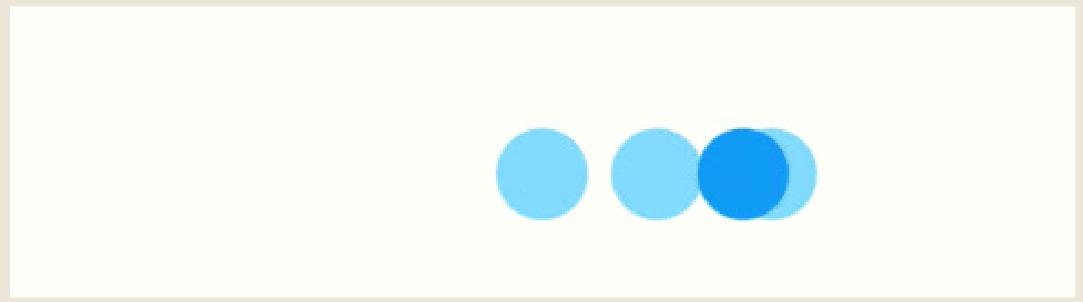
Graph Neural Networks: Focused on graph-structured data, these networks analyze relationships between nodes, ideal for social network analysis and recommendation systems.

RECURRENT NEURAL NETWORKS

Recurrent Neural Networks (RNNs) differ from feed-forward neural networks as they have a built-in memory, allowing them to process sequences of data. This makes RNNs well-suited for tasks like natural language processing and time series prediction. They can learn patterns in sequences by connecting the output from one time step to the input of the next, remembering previous information (the recurrence in the namesake).
Say you take a still snapshot of a ball moving in time.

Let's also say you want to predict the direction that the ball was moving. So with only the information that you see on the screen, how would you do this? Well, you can go ahead and take a guess, but any answer you'd come up with would be that, a random guess. Without knowledge of where the ball has been, you wouldn't have enough data to predict where it's going.

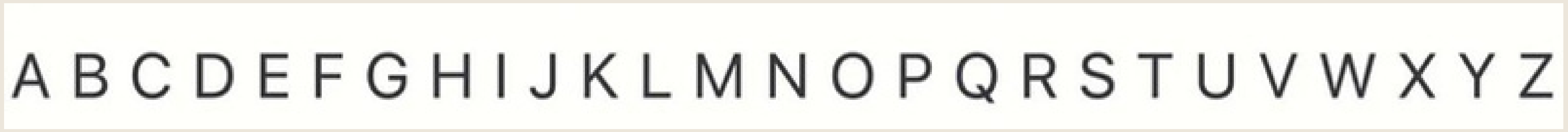
If you record many snapshots of the ball's position in succession, you will have enough information to make a better prediction. So this is a sequence, a particular order in which one thing follows another. With this information, you can now see that the ball is moving to the right.



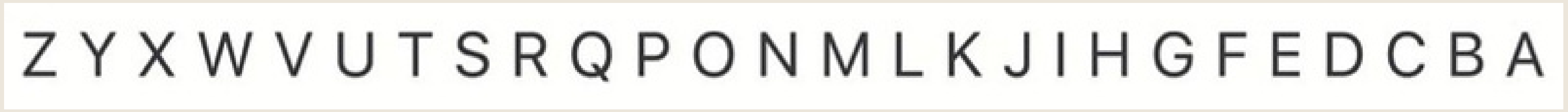
Text is a form of sequences. You can break Text up into a sequence of characters or a sequence of words.

SEQUENTIAL MEMORY IN RNNs

RNNs excel in handling sequential data for making predictions, thanks to what can be described as "sequential memory." To understand what sequential memory entails, here’s an exercise:
Imagine reciting the alphabet silently to yourself.


A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

That likely felt quite simple. If you've learned this sequence by heart, it flows naturally.
Now, attempt to recite the alphabet in reverse order.

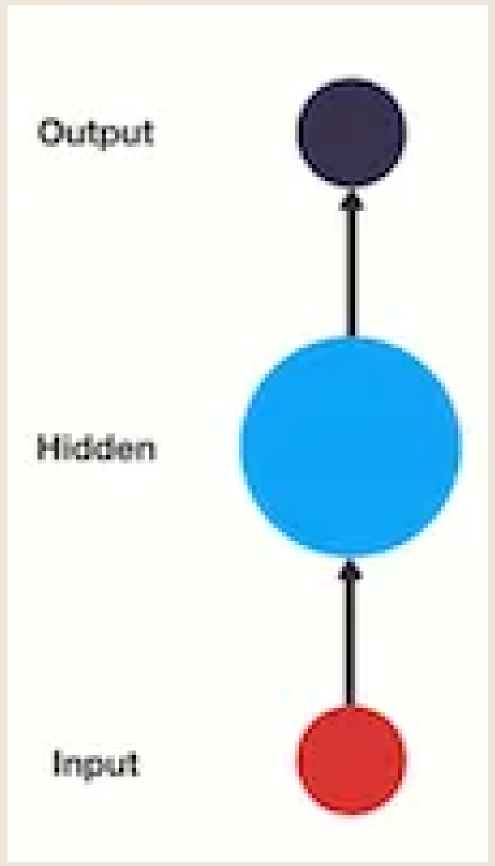

Z Y X W V U T S R Q P O N M L K J I H G F E D C B A

This task is presumably more challenging. Without previous practice in this reverse sequence, most people struggle.
For a bit of amusement, try beginning with the letter F.

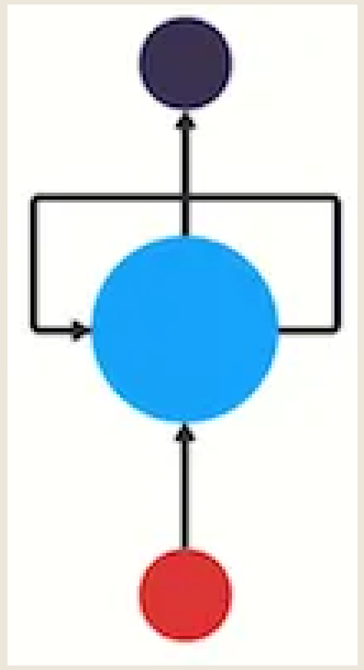
F

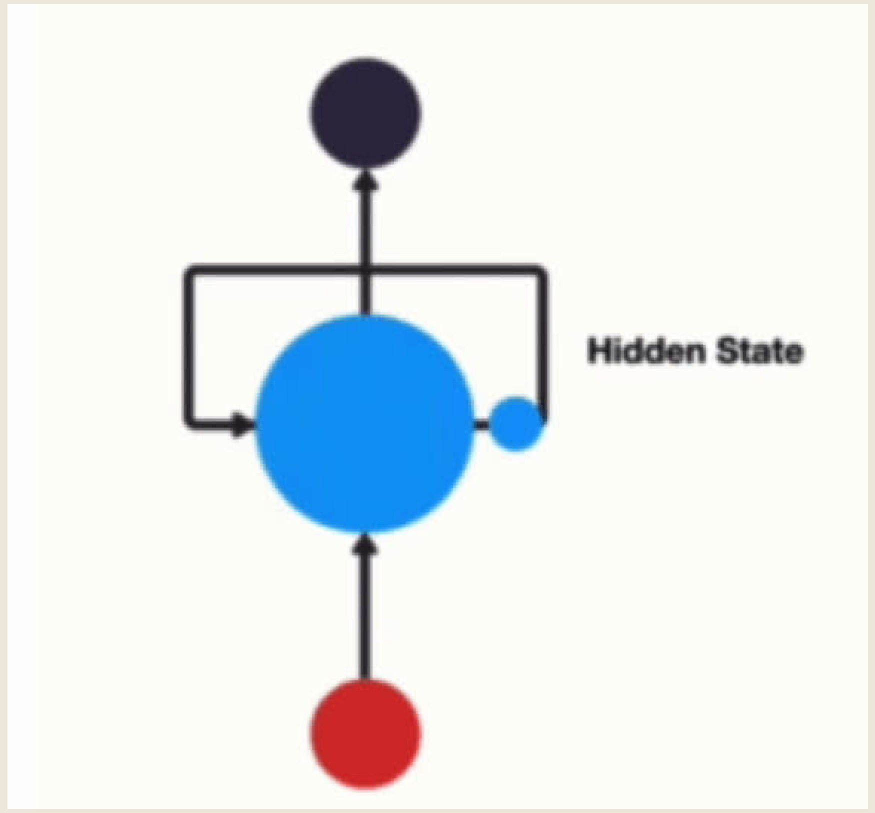
Initially, you might find it tricky to start from an unusual point in the sequence, but once you catch on to the pattern, the rest tends to follow more smoothly. This difficulty has a rational explanation. You've learned the alphabet as a specific sequence, and sequential memory is what allows your brain to easily identify and recall patterns within sequences.

Let's look at a traditional neural network also known as a feed-forward neural network. It has its input layer, hidden layer, and the output layer.



How do we get a feed-forward neural network to be able to use previous information to effect later ones? What if we add a loop in the neural network that can pass prior information forward? And that's essentially what a recurrent neural network does. An RNN has a looping mechanism that acts as a highway to allow information to flow from one step to the next.



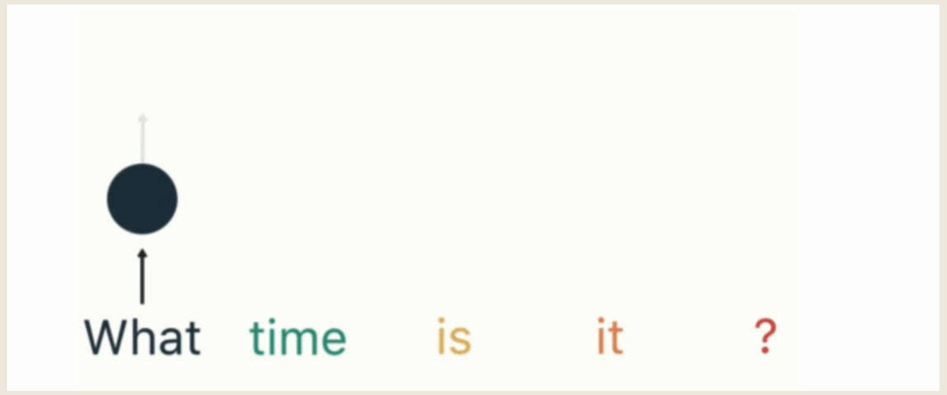


This information is the hidden state, which is a representation of previous inputs.

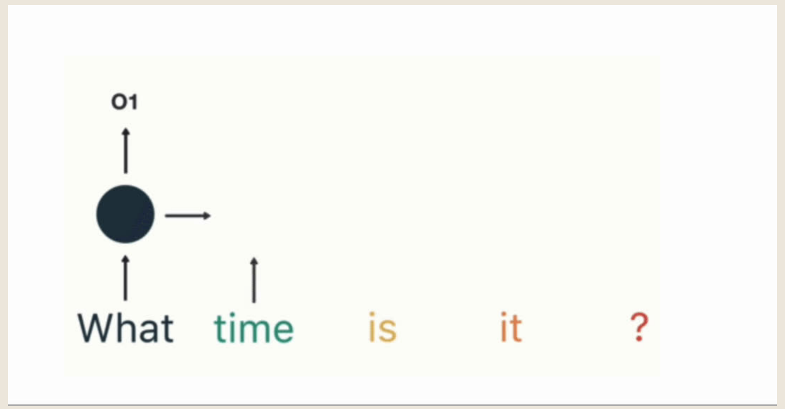
Let's take a sentence... "what time is it?". To start, we break up the sentence into individual words. RNN's work sequentially so we feed it one word at a time.

What
time
is
it
?

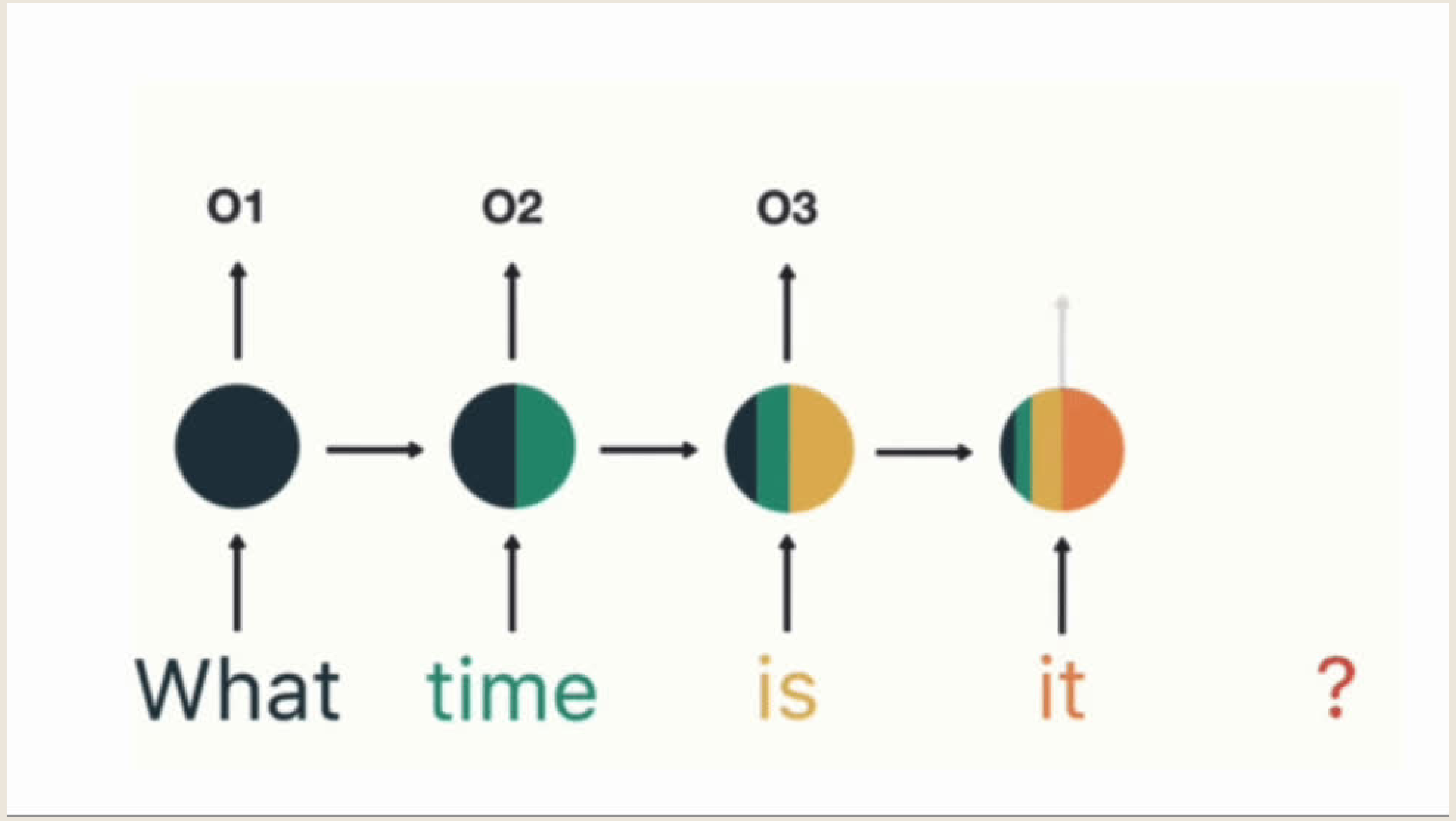
The first step is to feed “What” into the RNN. The RNN encodes “What” and produces an output.



For the next step, we feed the word “time” and the hidden state from the previous step. The RNN now has information on both the word “What” and “time.”



We repeat this process, until the final step. At the final step, the RNN has encoded information from all the words in previous steps.



Since the final output was created from the rest of the sequence, we should be able to take the final output and pass it to the feed-forward layer to classify an intent.

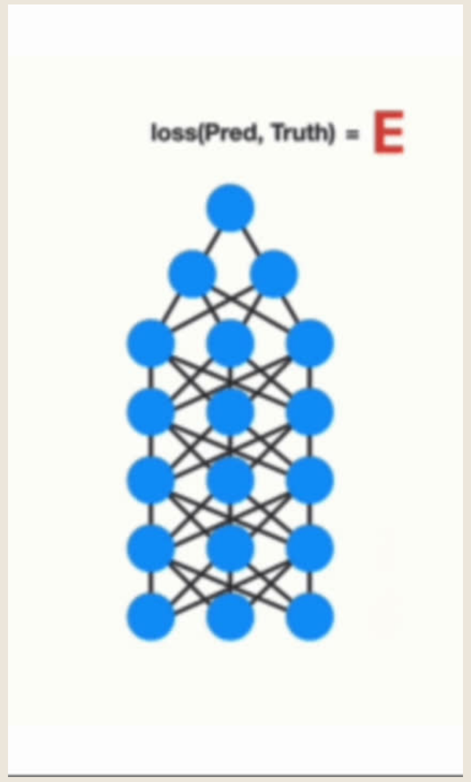
PROBLEMS WITH RNNs

short-term memory.

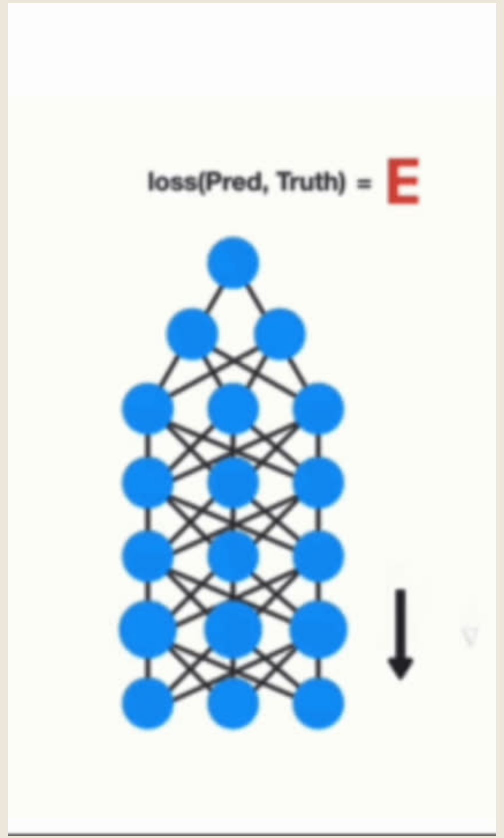
Short-term memory is caused by the infamous **vanishing gradient problem**, which is also prevalent in other neural network architectures. As the RNN processes more steps, it has troubles retaining information from previous steps. As you can see, the information from the word "what" and "time" is almost non-existent at the final time step. Short-Term memory and the vanishing gradient is due to the nature of back-propagation; an algorithm used to train and optimize neural networks. To understand why this is, let's take a look at the effects of back propagation on a deep feed-forward neural network. Training a neural network has three major steps.

- First, it does a forward pass and makes a prediction.
- Second, it compares the prediction to the ground truth using a loss function. The loss function outputs an error value which is an estimate of how poorly the network is performing.
- Last, it uses that error value to do back propagation which calculates the gradients for each node in the network.

The gradient is the value used to adjust the networks internal weights, allowing the network to learn. The bigger the gradient, the bigger the adjustments and vice versa. Here is where the problem lies. When doing back propagation, each node in a layer calculates it's gradient with respect to the effects of the gradients, in the layer before it. So if the adjustments to the layers before it is small, then adjustments to the current layer will be even smaller. That causes gradients to exponentially shrink as it back propagates down. The earlier layers fail to do any learning as the internal weights are barely being adjusted due to extremely small gradients. And that's the vanishing gradient problem.



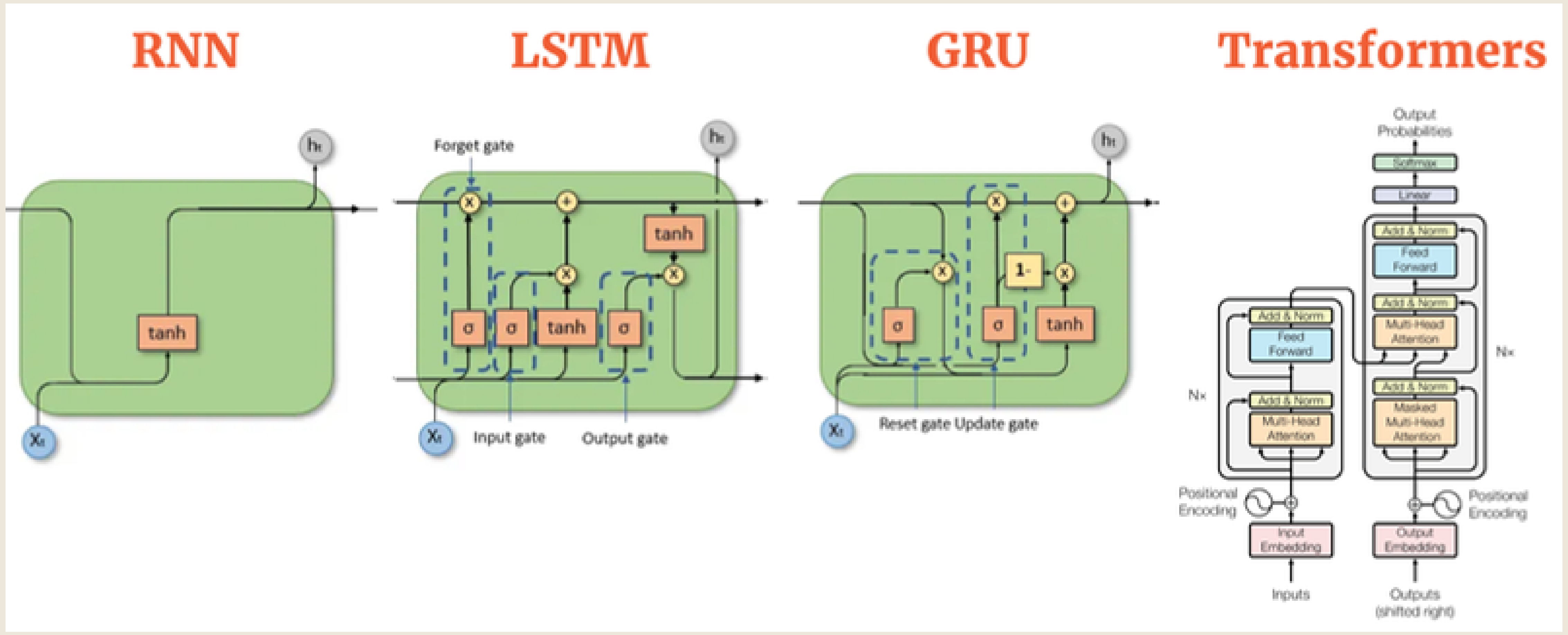
Each time step in a recurrent neural network can be considered a layer. Training a recurrent neural network involves an application of back-propagation known as back-propagation through time. The gradient values exponentially shrink as they propagate through each time step.



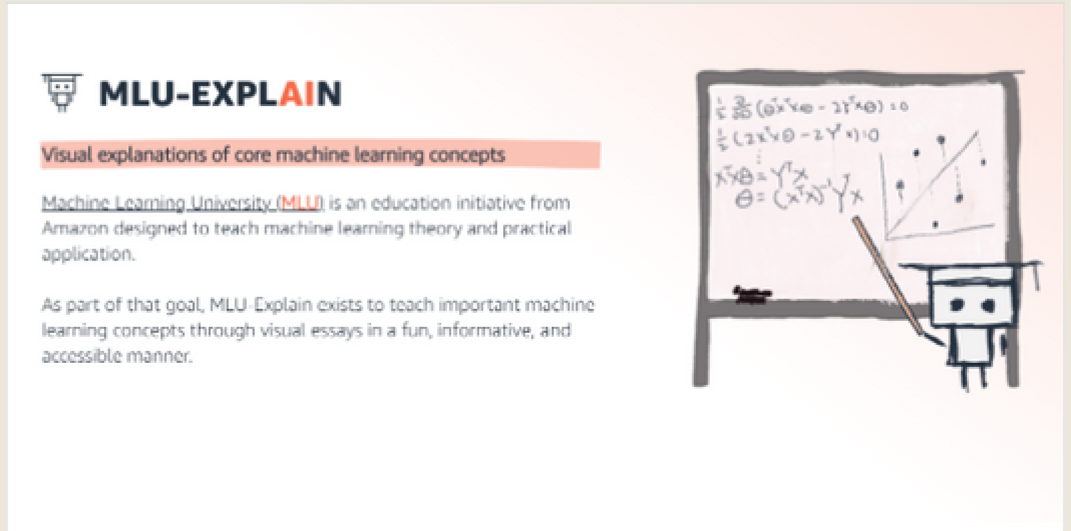
Again, the gradient is used to make adjustments in the neural networks weights thus allowing it to learn. Small gradients mean small adjustments. That causes the early layers not to learn.

Because of vanishing gradients, the RNN doesn't learn the long-range dependencies across time steps. That means that there is a possibility that the word "what" and "time" are not considered when trying to predict the user's intention. The network then has to make the best guess with "is it?". That's pretty ambiguous and would be difficult even for a human. So not being able to learn on earlier time steps causes the network to have a short-term memory.

Ok so RNN's suffer from short-term memory, so how do we combat that? To mitigate short-term memory, two specialized recurrent neural networks were created. One called ***Long Short-Term Memory*** *or LSTM's for short. The other is* ***Gated Recurrent Units*** *or GRU's. LSTM's and GRU's essentially function just like RNN's, but they're capable of learning long-term dependencies using mechanisms called "****gates****." These gates are different tensor operations that can learn what information to add or remove to the hidden state. Because of this ability, short-term memory is less of an issue for them.*



References


MLU-EXPLAIN

Visual explanations of core machine learning concepts

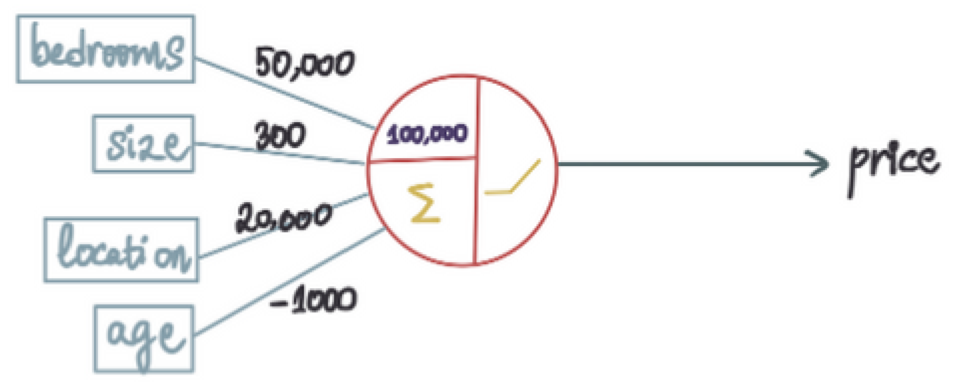
Machine Learning University (MLU) is an education initiative from Amazon designed to teach machine learning theory and practical application.

As part of that goal, MLU-Explain exists to teach important machine learning concepts through visual essays in a fun, informative, and accessible manner.

MLU-Explain

Visual explanations of core machine learning concepts.

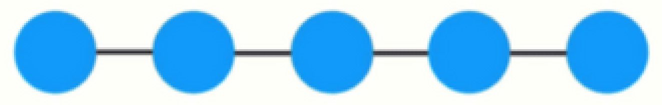
github.io



Deep Learning Illustrated, Part 1: How Does a Neural Network Work?

An illustrated and intuitive guide to Neural Networks

Towards Data Science / Feb 8



Illustrated Guide to Recurrent Neural Networks

Understanding the Intuition

Towards Data Science / Jun 28, 2020