# Artificial Neural Networks

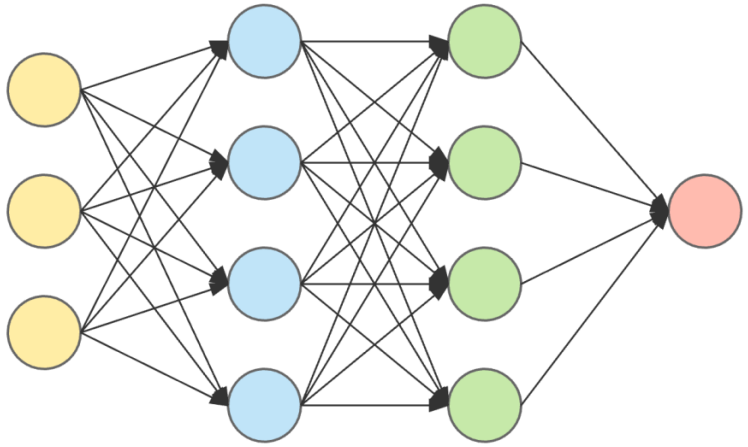
Most of these presentation slides are taken from the tutorials of **Quoc V. Le** titled "A Tutorial on Deep Learning Part 1: Nonlinear Classifiers and The Backpropagation Algorithm.“ and “A Tutorial on Deep Learning Part 2: Autoencoders, Convolutional Neural Networks and Recurrent Neural Networks”

# Introduction

- Artificial Neural Networks (ANNs) are a computational model inspired by the biological neural networks of the human brain.

- ANNs are used in machine learning and artificial intelligence to solve complex problems and make predictions based on input data.

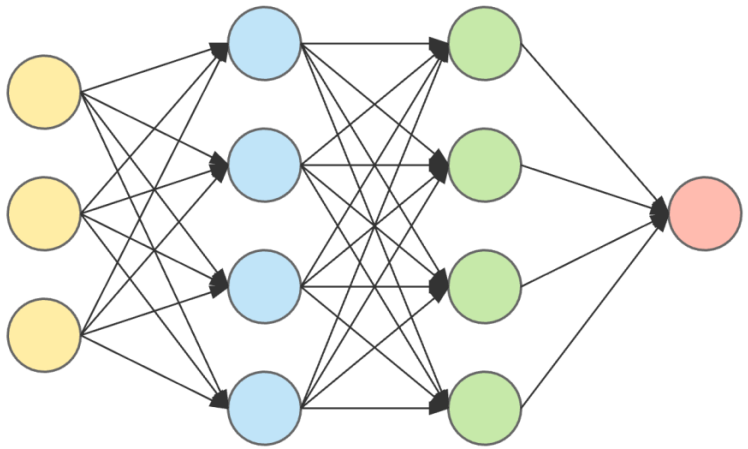
- They consist of interconnected nodes, called artificial neurons or units, organized in layers.



# Introduction

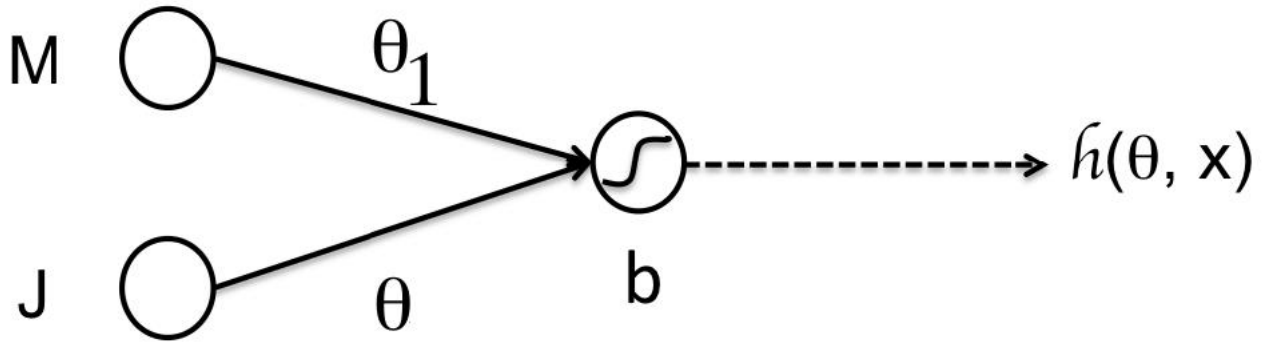
- The basic building block of an ANN is the artificial neuron, which receives one or more inputs, applies a mathematical function to them, and produces an output.

- The output is typically passed through an activation function to introduce non-linearity into the network, allowing it to model complex relationships in the data.



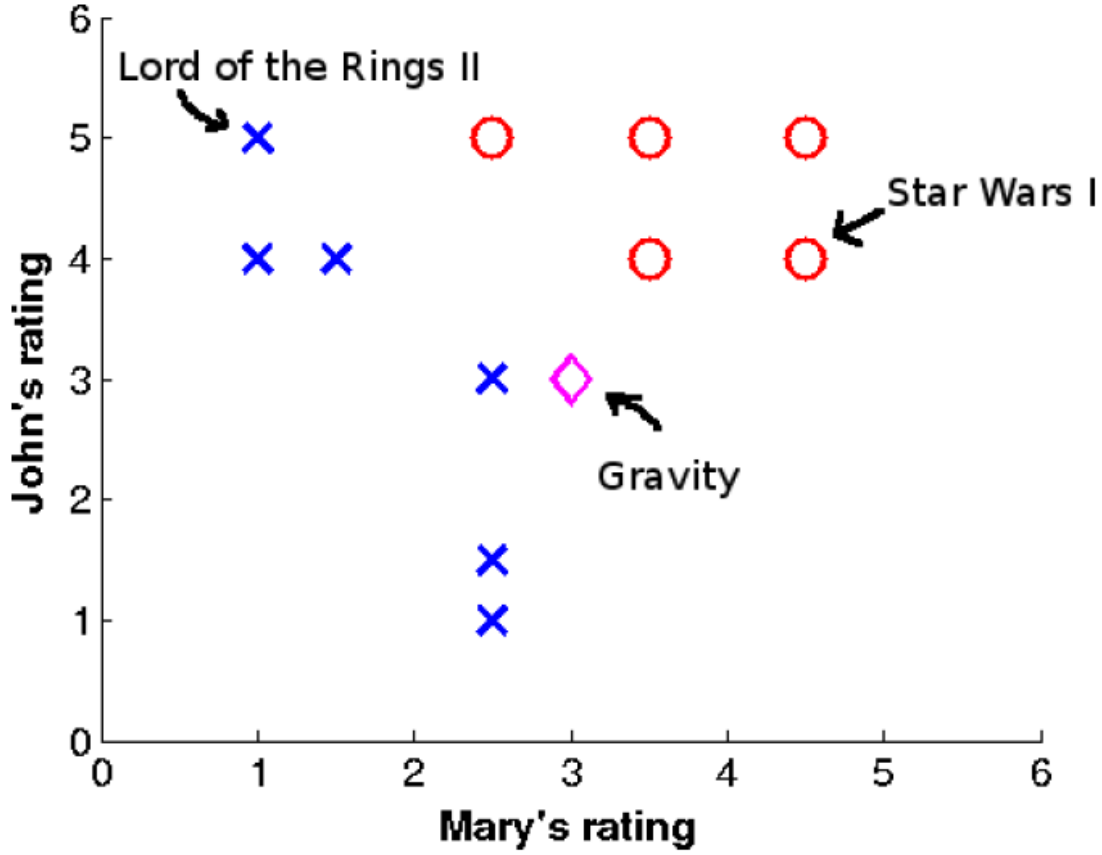
## An example of movie recommendations

| Movie name | Mary's rating | John's rating | I like? |
|---|---|---|---|
| Lord of the Rings II | 1 | 5 | No |
| ... | ... | ... | ... |
| Star Wars I | 4.5 | 4 | Yes |
| Gravity | 3 | 3 | ? |



# An example of movie recommendations

## Let's visualize the data



# An example of movie recommendations - A bounded decision function

Let's write a computer program to answer this question. For every movie, we construct an example x which has two dimensions: the first dimension $\mathbf{x_1}$ is Mary's rating and the second dimension $\mathbf{x_2}$ is John’s rating.

Every past movie is also associated with a label y to indicate whether I like the movie or not. For now, let's say **y** is a scalar that should have one of the two values, **0** to mean “**I do not like**" or **1** to mean “**I do like**" the movie.

Our goal is to come up with a **decision function h(x)** to approximate **y**.

# An example of movie recommendations - A bounded decision function

- Our **decision function h(x)** can be as simple as a weighted linear combination of Mary's and John's ratings:

$$h(x;\theta,b) = \theta_1 x_1 + \theta_2 x_2 + b, \text{ which can also be written as } h(x;\theta,b) = \theta^T x + b$$

In the equation above, the value of function $h(x)$ depends on $\theta_1, \theta_2$ and $b$.

# An example of movie recommendations - A bounded decision function

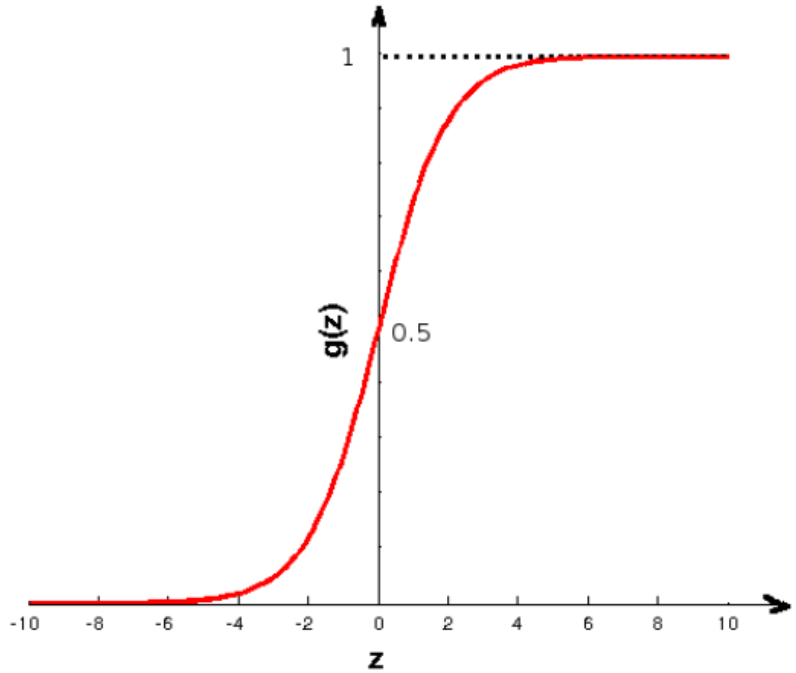
- The decision function h unfortunately has a problem: its values can be arbitrarily **large** or **small**. We wish its values to fall between **0** and **1** because those are the two extremes of **y** that we want to approximate.

A simple way to force h to have values between 0 and 1 is to map it through another function called the sigmoid function, which is bounded between 0 and 1:

$$h(x;\theta,b) = g(\theta^T x + b), \text{ where } g(z) = \frac{1}{1+\exp(-z)}$$

# An example of movie recommendations - A bounded decision function

$$h(x;\theta,b) = g(\theta^T x + b), \text{ where } g(z) = \frac{1}{1+\exp(-z)}$$




The value of function **h** is now bounded between **0** and **1**

## Using past data to learn the decision function

We will use the past data to learn $\theta, b$ to approximate $y$. In particular, we want to obtain $\theta$, $b$ such that:

$h(x^{(1)}; \theta, b) \approx y^{(1)}$, where $x^{(1)}$ is Mary's and John's ratings for 1st movie, "Lord of the Rings II"

$h(x^{(2)}; \theta, b) \approx y^{(2)}$, where $x^{(2)}$ is Mary's and John's ratings for 2nd movie

...

$h(x^{(m)}; \theta, b) \approx y^{(m)}$, where $x^{(m)}$ is Mary's and John's ratings for m-th movie

To find the values of $\theta$ and $b$ we can try to minimize the following *objective function*, which is the sum of differences between the decision function $h$ and the label $y$:

$$J(\theta, b) = \left(h(x^{(1)}; \theta, b) - y^{(1)}\right)^2 + \left(h(x^{(2)}; \theta, b) - y^{(2)}\right)^2 + ... + \left(h(x^{(m)}; \theta, b) - y^{(m)}\right)^2$$

$$= \sum_{i=1}^{m} \left(h(x^{(i)}; \theta, b) - y^{(i)}\right)^2$$

## Using stochastic gradient descent to minimize a function

To minimize the above function, we can iterate through the examples and slowly update the parameters $\theta$ and $b$ in the direction of minimizing each of the small objective $\left(h(x^{(i)};\theta,b)-y^{(i)}\right)^2$. Concretely, we can update the parameters in the following manner:

$$\theta_1 = \theta_1 - \alpha\Delta\theta_1$$
$$\theta_2 = \theta_2 - \alpha\Delta\theta_2$$
$$b = b - \alpha\Delta b$$

where $\alpha$ is a small non-negative scalar. A large $\alpha$ will give aggressive updates whereas a small $\alpha$ will give conservative updates. Such algorithm is known as *stochastic gradient descent (or SGD)* and $\alpha$ is known as the *learning rate*.

## Using stochastic gradient descent to minimize a function

To minimize the above function, we can iterate through the examples and slowly update the parameters $\theta$ and $b$ in the direction of minimizing each of the small objective $\left(h(x^{(i)};\theta,b)-y^{(i)}\right)^2$. Concretely, we can update the parameters in the following manner:

$$\theta_1 = \theta_1 - \alpha\Delta\theta_1$$
$$\theta_2 = \theta_2 - \alpha\Delta\theta_2$$
$$b = b - \alpha\Delta b$$

where $\alpha$ is a small non-negative scalar. A large $\alpha$ will give aggressive updates whereas a small $\alpha$ will give conservative updates. Such algorithm is known as *stochastic gradient descent (or SGD)* and $\alpha$ is known as the *learning rate*.

## Using stochastic gradient descent to minimize a function

$$\Delta\theta_1 = 2\left[g(\theta^T x^{(i)} + b) - y^{(i)}\right]\left[1 - g(\theta^T x^{(i)} + b)\right]g(\theta^T x^{(i)} + b)x_1^{(i)}$$

$$\Delta\theta_2 = 2\left[g(\theta^T x^{(i)} + b) - y^{(i)}\right]\left[1 - g(\theta^T x^{(i)} + b)\right]g(\theta^T x^{(i)} + b)x_2^{(i)}$$

$$\Delta b = 2\left[g(\theta^T x^{(i)} + b) - y^{(i)}\right]\left[1 - g(\theta^T x^{(i)} + b)\right]g(\theta^T x^{(i)} + b)$$

where

$$g(\theta^T x^{(i)} + b) = \frac{1}{1 + \exp(-\theta^T x^{(i)} - b)}$$

# Learning

Now, we have the stochastic gradient descent algorithm **to learn the decision function** $h(x; \theta, b)$:

1. Initialize the parameters $\theta, b$ at random,
2. Pick a random example $\{x^{(i)}, y^{(i)}\}$,
3. Compute the partial derivatives $\theta_1, \theta_2$ and $b$
4. Update parameters as follows:

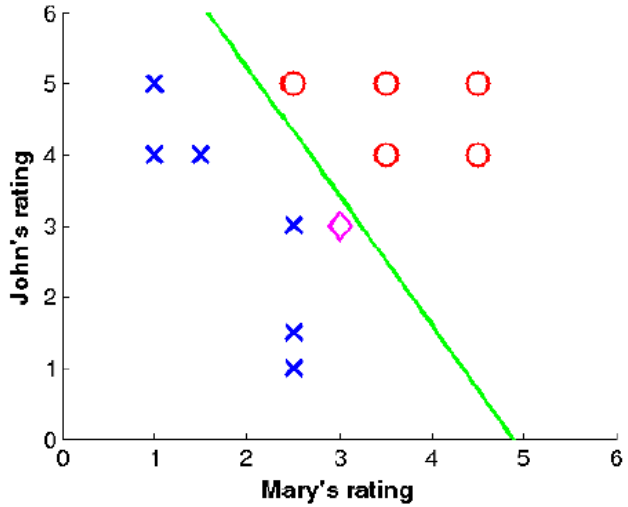
$$\theta_1 = \theta_1 - \alpha\Delta\theta_1$$
$$\theta_2 = \theta_2 - \alpha\Delta\theta_2$$
$$b = b - \alpha\Delta b$$

5. Back to step 2.

# Learning

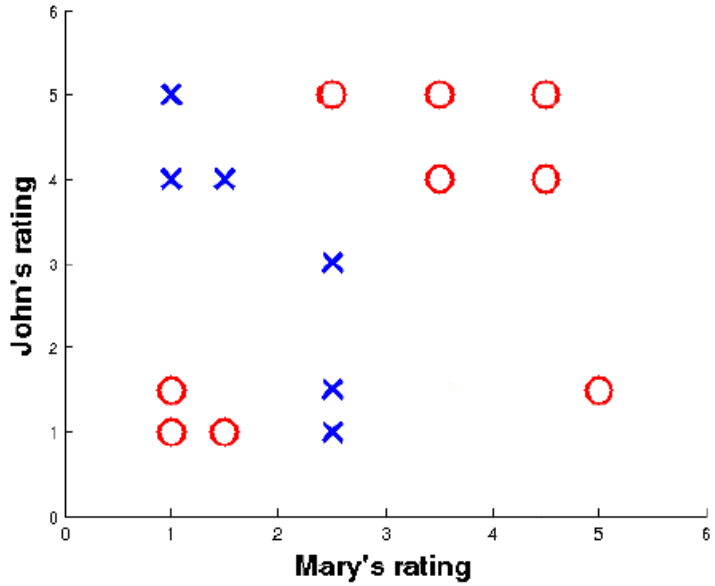
- We can stop stochastic gradient descent when **the parameters do not change, or the number of iteration exceeds a certain upper bound.**

We can stop stochastic gradient descent when the parameters do not change or the number of iteration exceeds a certain upper bound. At convergence, we will obtain a function $h(x;\theta,b)$ which can be used to predict whether I like a new movie $x$ or not: $h > 0.5$ means I will like the movie, otherwise I do not like the movie. The values of $x$'s that cause $h(x;\theta,b)$ to be 0.5 is the "decision boundary." We can plot this "decision boundary" to have:



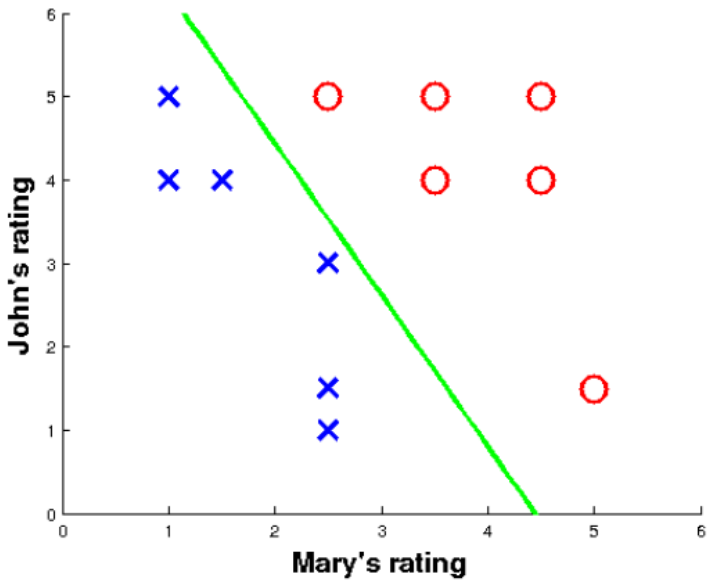
# The limitations of linear decision function

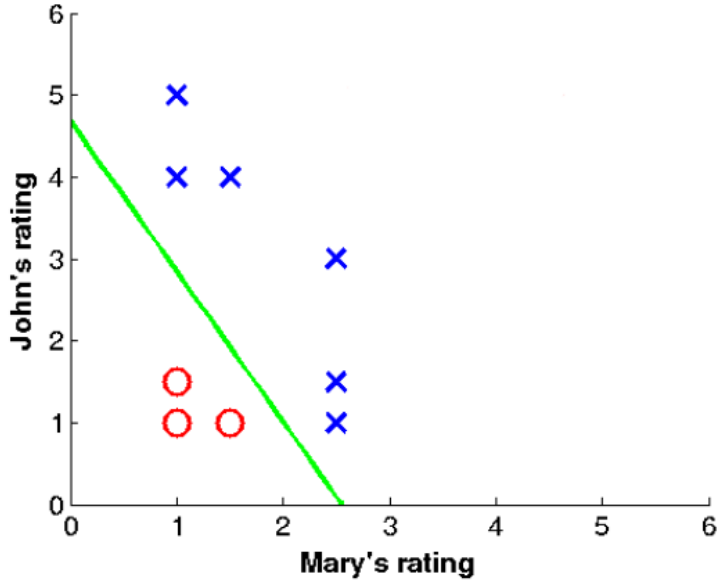
- Assume your friend **Susan has different movie tastes**. If we plot her data, the graph will look rather dfferent:




Susan likes some of the movies that Mary and John rated poorly. The question is how we can come up with a decision function for Susan. From looking at the data, **the decision function must be more complex** than the decision we saw before.

# Decompose the complex into smaller problems




**Is it possible to combine these two decision functions into one final decision function for the original data?**

# A decision function of decision functions

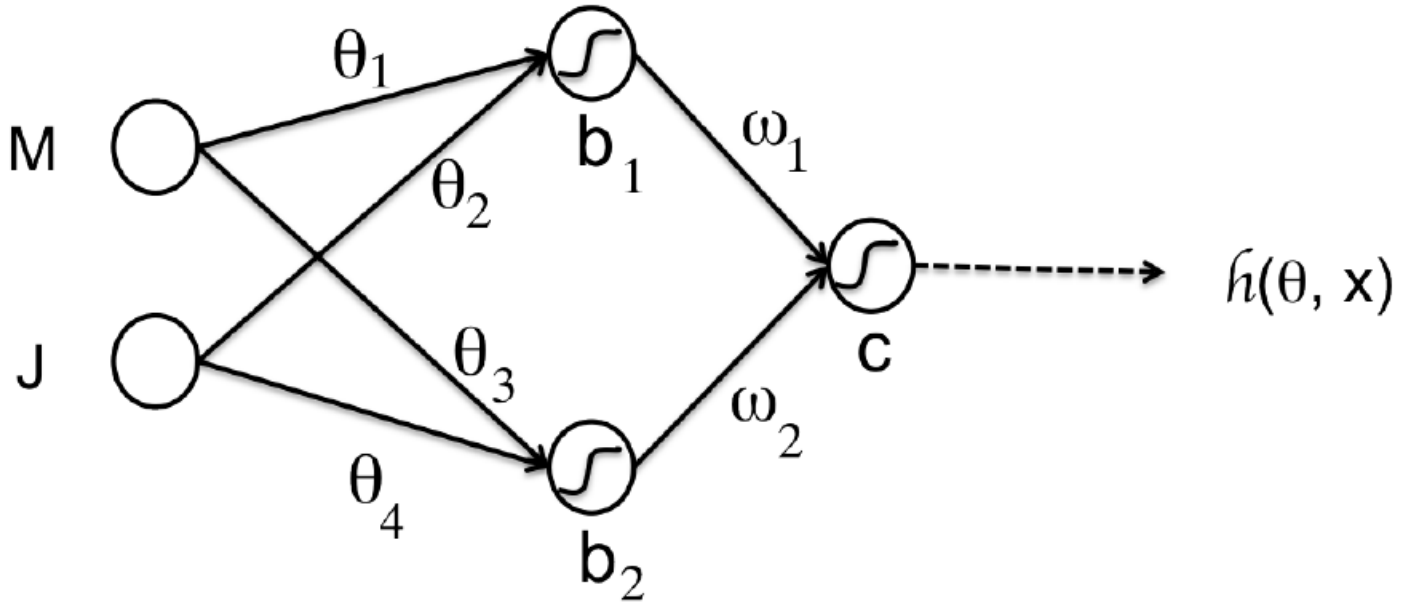
Let's suppose, as stated above, the two decision functions are $h_1(x; (\theta_1, \theta_2), b_1)$ and $h_2(x; (\theta_3, \theta_4), b_2)$. For every example $x^{(i)}$, we can then compute $h_1(x^{(i)}; (\theta_1, \theta_2), b_1)$ and $h_2(x^{(i)}; (\theta_3, \theta_4), b_2)$

If we lay out the data in a table, it would look like the first table that we saw:

| **Movie name** | **Output by decision function $h_1$** | **Output by decision function $h_2$** | **Susan likes?** |
|---|---|---|---|
| Lord of the Rings II | $h_1(x^{(1)})$ | $h_2(x^{(2)})$ | No |
| ... | ... | ... | ... |
| Star Wars I | $h_1(x^{(n)})$ | $h_2(x^{(n)})$ | Yes |
| Gravity | $h_1(x^{(n+1)})$ | $h_2(x^{(n+1)})$ | ? |

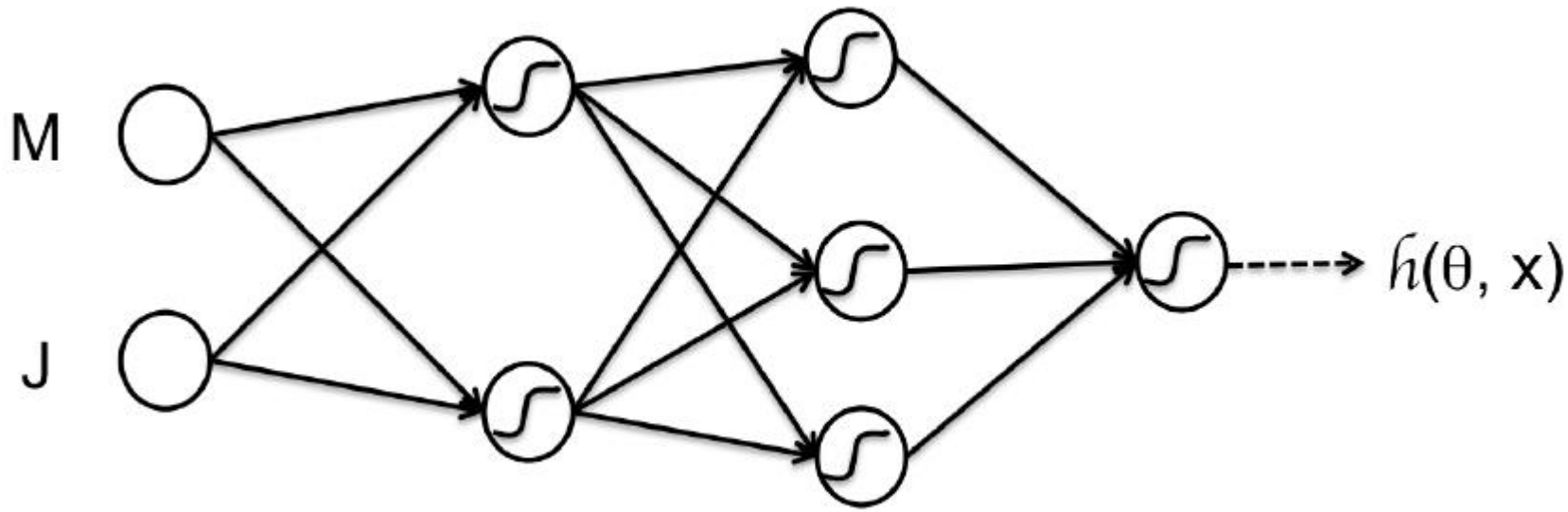
# A decision function of decision functions

- A graphical way to visualize the problem:



# Deeper network

- If you have a more complex function that you want to approximate, you may want to have a deeper network, maybe one that looks like this:



## The backpropagation algorithm

- Backpropagation is a fundamental algorithm used to train artificial neural networks. It is a method for efficiently computing the gradients of the network's parameters with respect to the loss function, which allows for updating the parameters during the training process.

# The backpropagation algorithm

Here's a step-by-step explanation of the backpropagation algorithm:

1. Initialize the network: Initialize the weights and biases of the neural network randomly or using some predetermined scheme.
2. Forward propagation: Pass input data through the network to compute the output predictions. Each neuron in the network computes a weighted sum of its inputs, applies an activation function, and passes the result to the next layer.
3. Compute the loss: Compare the network's predictions to the true labels using a loss function. The loss function measures the discrepancy between the predicted and actual values.
4. Backward propagation: Start from the output layer and propagate the error gradients backward through the network. Compute the gradients of the loss with respect to the network's parameters using the chain rule.
5. Update parameters: Adjust the weights and biases of the network based on the computed gradients. This step typically involves using an optimization algorithm, such as stochastic gradient descent (SGD), to update the parameters in the direction that minimizes the loss.
6. Repeat steps 2-5: Repeat the forward and backward propagation steps for multiple iterations or epochs until the network's performance improves and the loss is minimized.

## Challenges

- The training of a neural network using backpropagation with randomly initialized weights can be challenging.

- When weights are randomly initialized, the network initially makes random predictions, and the goal is to update the weights iteratively to minimize the loss and improve the network's performance.

## Challenges

- The training of a neural network using backpropagation with randomly initialized weights can be challenging.

- When weights are randomly initialized, the network initially makes random predictions, and the goal is to update the weights iteratively to minimize the loss and improve the network's performance.

## Pretraining of neural networks

- Pretraining is a technique used in deep learning to initialize the weights of a neural network before the actual training process begins.

- The idea behind pretraining is to leverage a large amount of labeled or unlabeled data to capture general patterns and features that can benefit the subsequent task.

- It can be particularly effective when the main task has limited labeled data, reducing the risk of overfitting and improving the network's ability to generalize.

## Autoencoders for pretraining neural networks

- Autoencoders are commonly used for pretraining neural networks.

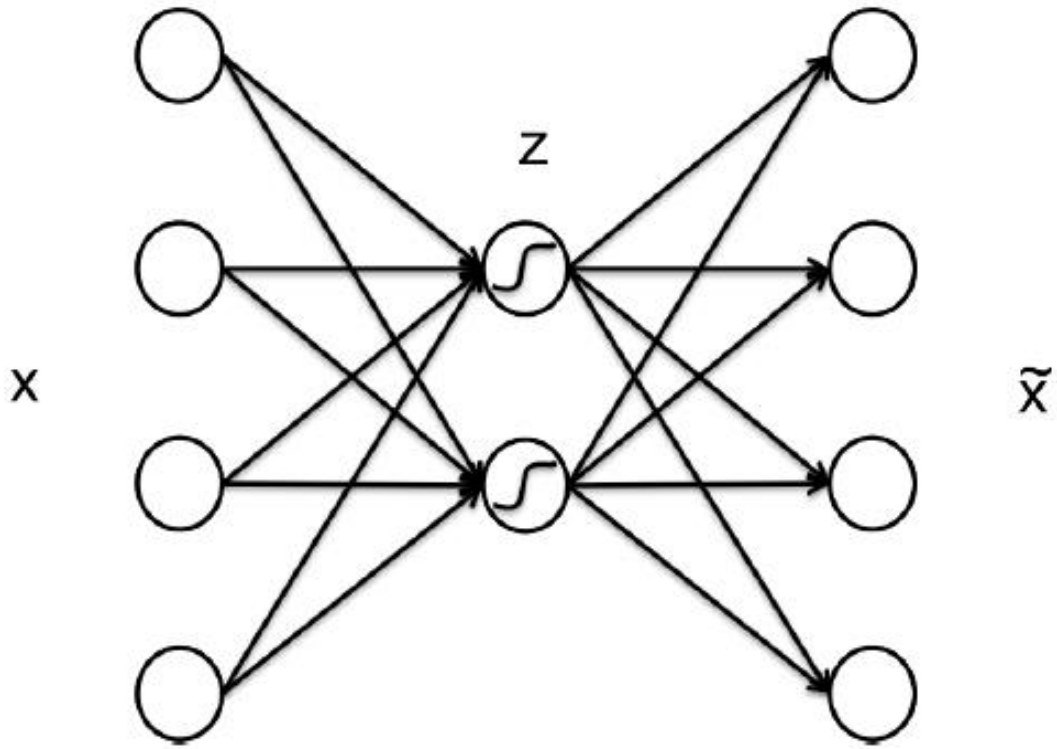
- An autoencoder is an unsupervised learning algorithm that aims to learn efficient representations of input data by reconstructing it from a compressed latent space.

- Pretraining with autoencoders can be beneficial in **initializing the weights** of a neural network before fine-tuning for a specific task.

# Autoencoders for pretraining neural networks

$$z^{(i)} = W_1 x^{(i)} + b_1$$
$$\tilde{x}^{(i)} = W_2 z^{(i)} + b_2$$

$$\begin{aligned} J(W_1, b_1, W_2, b_2) &= \sum_{i=1}^{m} \left( \tilde{x}^{(i)} - x^{(i)} \right)^2 \\ &= \sum_{i=1}^{m} \left( W_2 z^{(i)} + b_2 - x^{(i)} \right)^2 \\ &= \sum_{i=1}^{m} \left( W_2 (W_1 x^{(i)} + b_1) + b_2 - x^{(i)} \right)^2 \end{aligned}$$

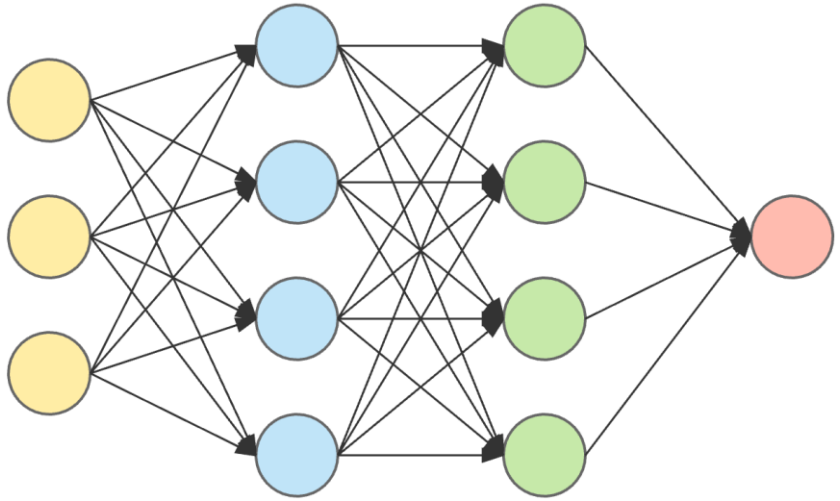


# Training Deep Neural Network

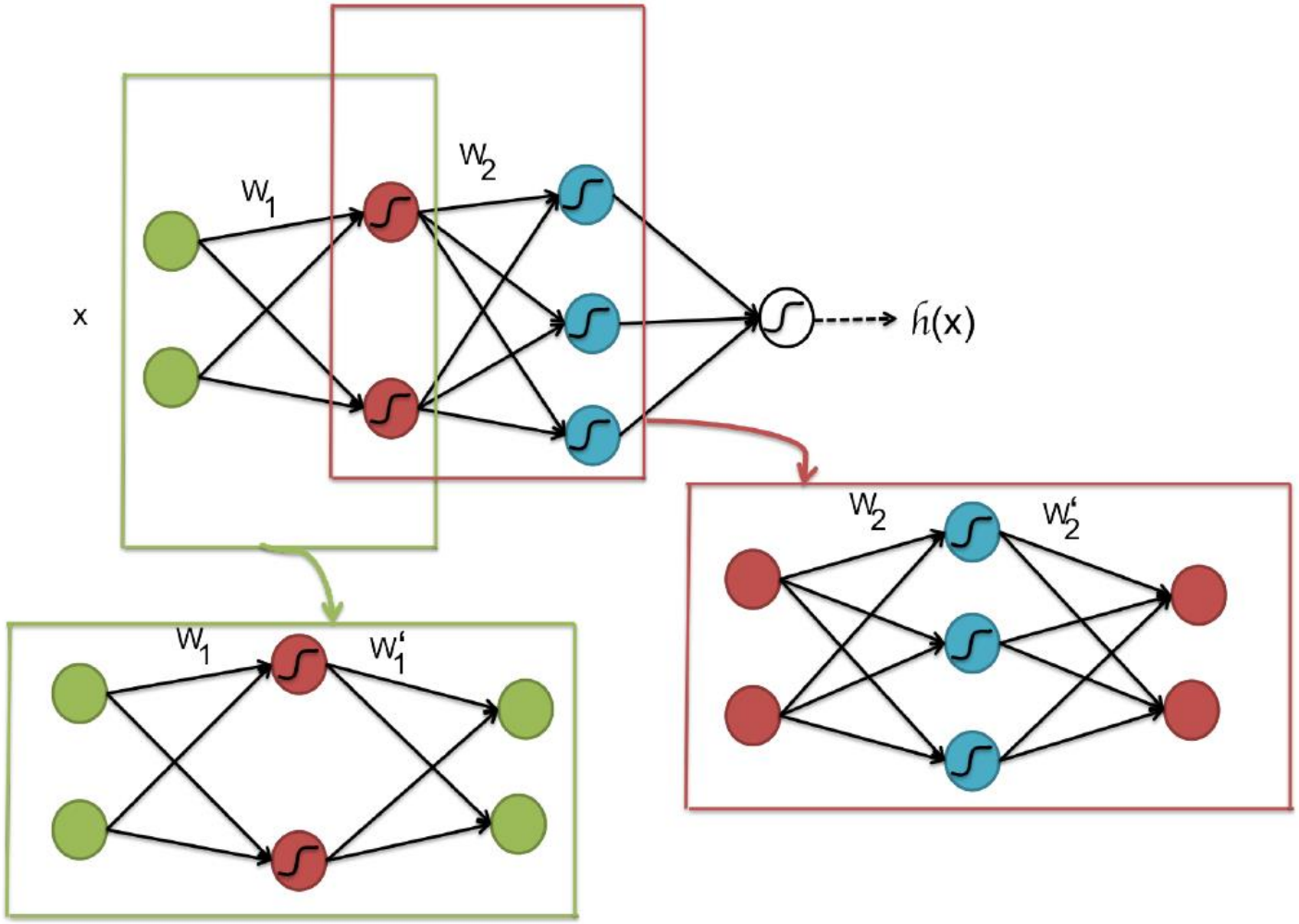
- The goal of pretraining is to address the above problems. With pretraining, the process of training **a deep network** is divided in a sequence of steps:

- **Pretraining step:** train a sequence of shallow autoencoders, greedily one layer at a time, using unsupervised data,
- **Fine-tuning step 1:** train the last layer using supervised data,

- **Fine-tuning step 2:** use backpropagation to ne-tune the entire network using supervised data.



# The pretraining step



## Pretraining for learning representation

- Pretraining for learning representations is a technique used in deep learning to train models to extract meaningful and informative representations from the input data.

- The goal is to learn representations that capture relevant features and patterns in the data

# Learning Representation

**Case study:** Machine Learning-Powered Malware Detection and Classification