UNIT-6

Dimensionality reduction:
In statistics, machine learning, and information theory, **dimensionality reduction** or **dimension reduction** is the process of reducing the number of random variables under consideration by obtaining a set of principle variables. Approaches can be divided into feature selection and feature extraction.

Principal Component Analysis

The main idea of principal component analysis (PCA) is to reduce the dimensionality of a data set consisting of many variables correlated with each other, either heavily or lightly, while retaining the variation present in the dataset, up to the maximum extent. The same is done by transforming the variables to a new set of variables, which are known as the principal components (or simply, the PCs) and are orthogonal, ordered such that the retention of variation present in the original variables decreases as we move down in the order. So, in this way, the 1st principal component retains maximum variation that was present in the original components. The principal components are the eigenvectors of a covariance matrix, and hence they are orthogonal.

Importantly, the dataset on which PCA technique is to be used must be scaled. The results are also sensitive to the relative scaling. As a layman, it is a method of summarizing data. Imagine some wine bottles on a dining table. Each wine is described by its attributes like colour, strength, age, etc. But redundancy will arise because many of them will measure related properties. So what PCA will do in this case is summarize each wine in the stock with less characteristics.

Intuitively, Principal Component Analysis can supply the user with a lower-dimensional picture, a projection or "shadow" of this object when viewed from its most informative viewpoint.

- Dimensionality (get sample code): It is the number of random variables in a dataset or simply the number of features, or rather more simply, the number of columns present in your dataset.
- Correlation (get sample code): It shows how strongly two variable are related to each other. The value of the same ranges for -1 to +1. Positive indicates that when one variable increases, the other increases as well, while negative indicates the other decreases on increasing the former. And the modulus value of indicates the strength of relation.
- Orthogonal: (get sample code) Uncorrelated to each other, i.e., correlation between any pair of variables is 0.
- Eigenvectors: (get sample code) Eigenvectors and Eigenvalues are in itself a big domain, let's restrict ourselves to the knowledge of the same which we would require here. So, consider a non-zero vector *v*. It is an eigenvector of a square matrix *A*, if *Av* is a scalar multiple of **v**. Or simply:

$Av = \lambda v$

Here, v is the eigenvector and λ is the eigenvalue associated with it.

- Covariance Matrix: This matrix consists of the covariances between the pairs of variables. The *(i,j)*th element is the covariance between *i*-th and *j*-th variable.

Properties of Principal Component

Technically, a principal component can be defined as a linear combination of optimally-weighted observed variables. The output of PCA are these principal components, the number of which is less than or equal to the number of original variables. Less, in case when we wish to discard or reduce the dimensions in our dataset. The PCs possess some useful properties which are listed below:

1. The PCs are essentially the linear combinations of the original variables, the weights vector in this combination is actually the eigenvector found which in turn satisfies the principle of least squares.
2. The PCs are orthogonal, as already discussed.
3. The variation present in the PCs decrease as we move from the 1st PC to the last one, hence the importance.

The least important PCs are also sometimes useful in regression, outlier detection, etc.

Implementing PCA on a 2-D Dataset

Step 1: Normalize the data (get sample code)

First step is to normalize the data that we have so that PCA works properly. This is done by subtracting the respective means from the numbers in the respective column. So if we have two dimensions X and Y, all X become □- and all Y become □-. This produces a dataset whose mean is zero.

Step 2: Calculate the covariance matrix (get sample code)

Since the dataset we took is 2-dimensional, this will result in a 2x2 Covariance matrix.

$$Matrix(Covariance) = \begin{bmatrix} Var[X_1] & Cov[X_1, X_2] \\ Cov[X_2, X_1] & Var[X_2] \end{bmatrix}$$

Please note that $Var[X_1] = Cov[X_1,X_1]$ and $Var[X_2] = Cov[X_2,X_2]$.

Step 3: Calculate the eigenvalues and eigenvectors (get sample code)

Next step is to calculate the eigenvalues and eigenvectors for the covariance matrix. The same is possible because it is a square matrix. λ is an eigenvalue for a matrix A if it is a solution of the characteristic equation:

$det(\lambda I - A) = 0$

Where, I is the identity matrix of the same dimension as A which is a required condition for the matrix subtraction as well in this case and '*det*' is the determinant of the matrix. For each eigenvalue λ, a corresponding eigen-vector v, can be found by solving:

$(\lambda I - A)v = 0$

Step 4: Choosing components and forming a feature vector: (get sample code)

We order the eigenvalues from largest to smallest so that it gives us the components in order or significance. Here comes the dimensionality reduction part. If we have a dataset with n variables,

then we have the corresponding *n* eigenvalues and eigenvectors. It turns out that the eigenvector corresponding to the highest eigenvalue is the principal component of the dataset and it is our call as to how many eigenvalues we choose to proceed our analysis with. To reduce the dimensions, we choose the first *p* eigenvalues and ignore the rest. We do lose out some information in the process, but if the eigenvalues are small, we do not lose much.

Next we form a feature vector which is a matrix of vectors, in our case, the eigenvectors. In fact, only those eigenvectors which we want to proceed with. Since we just have 2 dimensions in the running example, we can either choose the one corresponding to the greater eigenvalue or simply take both.

Feature Vector = (eig_1, eig_2)

Step 5: Forming Principal Components: (get sample code)

This is the final step where we actually form the principal components using all the math we did till here. For the same, we take the transpose of the feature vector and left-multiply it with the transpose of scaled version of original dataset.

$NewData = FeatureVector^T \times ScaledData^T$

Here,

NewData is the Matrix consisting of the principal components,

FeatureVector is the matrix we formed using the eigenvectors we chose to keep, and

ScaledData is the scaled version of original dataset

('T' in the superscript denotes transpose of a matrix which is formed by interchanging the rows to columns and vice versa. In particular, a 2x3 matrix has a transpose of size 3x2)

If we go back to the theory of eigenvalues and eigenvectors, we see that, essentially, eigenvectors provide us with information about the patterns in the data. In particular, in the running example of 2-D set, if we plot the eigenvectors on the scatterplot of data, we find that the principal eigenvector (corresponding to the largest eigenvalue) actually fits well with the data. The other one, being perpendicular to it, does not carry much information and hence, we are at not much loss when deprecating it, hence reducing the dimension.

All the eigenvectors of a matrix are perpendicular to each other. So, in PCA, what we do is represent or transform the original dataset using these orthogonal (perpendicular) eigenvectors instead of representing on normal *x* and *y* axes. We have now classified our data points as a combination of contributions from both *x* and *y*. The difference lies when we actually disregard one or many eigenvectors, hence, reducing the dimension of the dataset. Otherwise, in case, we take all the eigenvectors in account, we are just transforming the co-ordinates and hence, not serving the purpose.

Applications of Principal Component Analysis

PCA is predominantly used as a dimensionality reduction technique in domains like facial recognition, computer vision and image compression. It is also used for finding patterns in data of high dimension in the field of finance, data mining, bioinformatics, psychology, etc.

PCA for images: (get sample code)

You must be wondering many a times show can a machine read images or do some calculations using just images and no numbers. We will try to answer a part of that now. For simplicity, we will be restricting our discussion to square images only. Any square image of size *NxN* pixels can be represented as a *NxN* matrix where each element is the intensity value of the image. (The image is formed placing the rows of pixels one after the other to form one single image.) So if you have a set of images, we can form a matrix out of these matrices, considering a row of pixels as a vector, we are ready to start principal component analysis on it. How is it useful ?

Say you are given an image to recognize which is not a part of the previous set. The machine checks the differences between the to-be-recognized image and each of the principal components. It turns out that the process performs well if PCA is applied and the differences are taken from the 'transformed' matrix. Also, applying PCA gives us the liberty to leave out some of the components without losing out much information and thus reducing the complexity of the problem.

For image compression, on taking out less significant eigenvectors, we can actually decrease the size of the image for storage. But to mention, on reproducing the original image from this will lose out some information for obvious reasons.

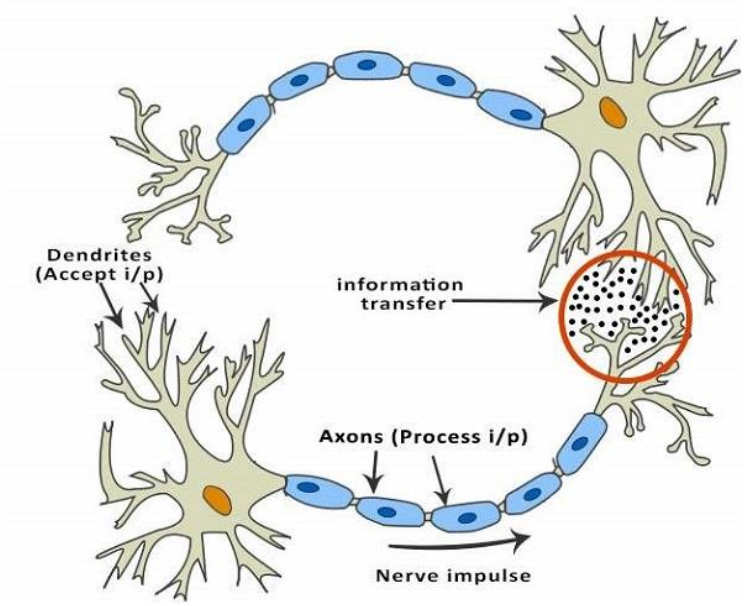
What are Artificial Neural Networks (ANNs)?

> "...a computing system made up of a number of simple, highly interconnected processing elements, which process information by their dynamic state response to external inputs."

Basic Structure of ANNs

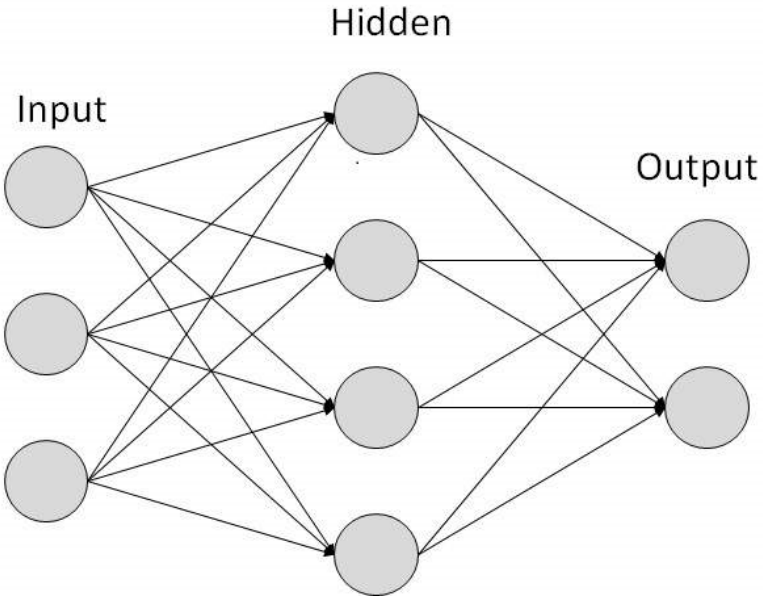
The idea of ANNs is based on the belief that working of human brain by making the right connections, can be imitated using silicon and wires as living **neurons** and **dendrites**.

The human brain is composed of 86 billion nerve cells called **neurons.** They are connected to other thousand cells by **Axons.** Stimuli from external environment or inputs from sensory organs are accepted by dendrites. These inputs create electric impulses, which quickly travel through the neural network. A neuron can then send the message to other neuron to handle the issue or does not send it forward.



ANNs are composed of multiple **nodes**, which imitate biological **neurons** of human brain. The neurons are connected by links and they interact with each other. The nodes can take input data and perform simple operations on the data. The result of these operations is passed to other neurons. The output at each node is called its **activation** or **node value.**

Each link is associated with **weight.** ANNs are capable of learning, which takes place by altering weight values. The following illustration shows a simple ANN −

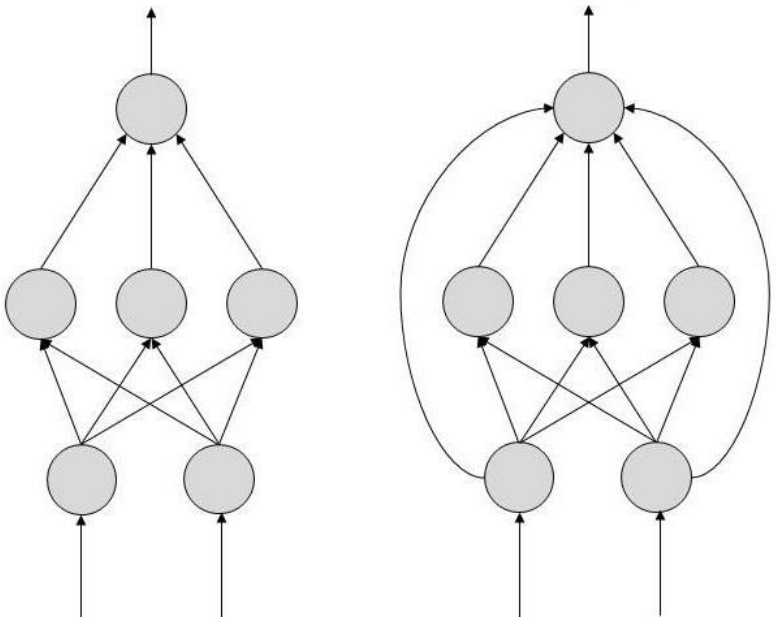


Types of Artificial Neural Networks

There are two Artificial Neural Network topologies − **FeedForward** and **Feedback.**

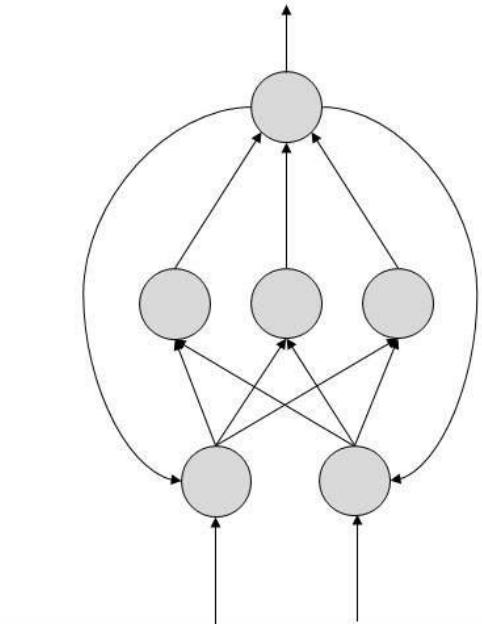
FeedForward ANN

In this ANN, the information flow is unidirectional. A unit sends information to other unit from which it does not receive any information. There are no feedback loops. They are used in pattern generation/recognition/classification. They have fixed inputs and outputs.



FeedBack ANN

Here, feedback loops are allowed. They are used in content addressable memories.



Working of ANNs

In the topology diagrams shown, each arrow represents a connection between two neurons and indicates the pathway for the flow of information. Each connection has a weight, an integer number that controls the signal between the two neurons.

If the network generates a “good or desired” output, there is no need to adjust the weights. However, if the network generates a “poor or undesired” output or an error, then the system alters the weights in order to improve subsequent results.

Machine Learning in ANNs

ANNs are capable of learning and they need to be trained. There are several learning strategies −

- **Supervised Learning** − It involves a teacher that is scholar than the ANN itself. For example, the teacher feeds some example data about which the teacher already knows the answers.

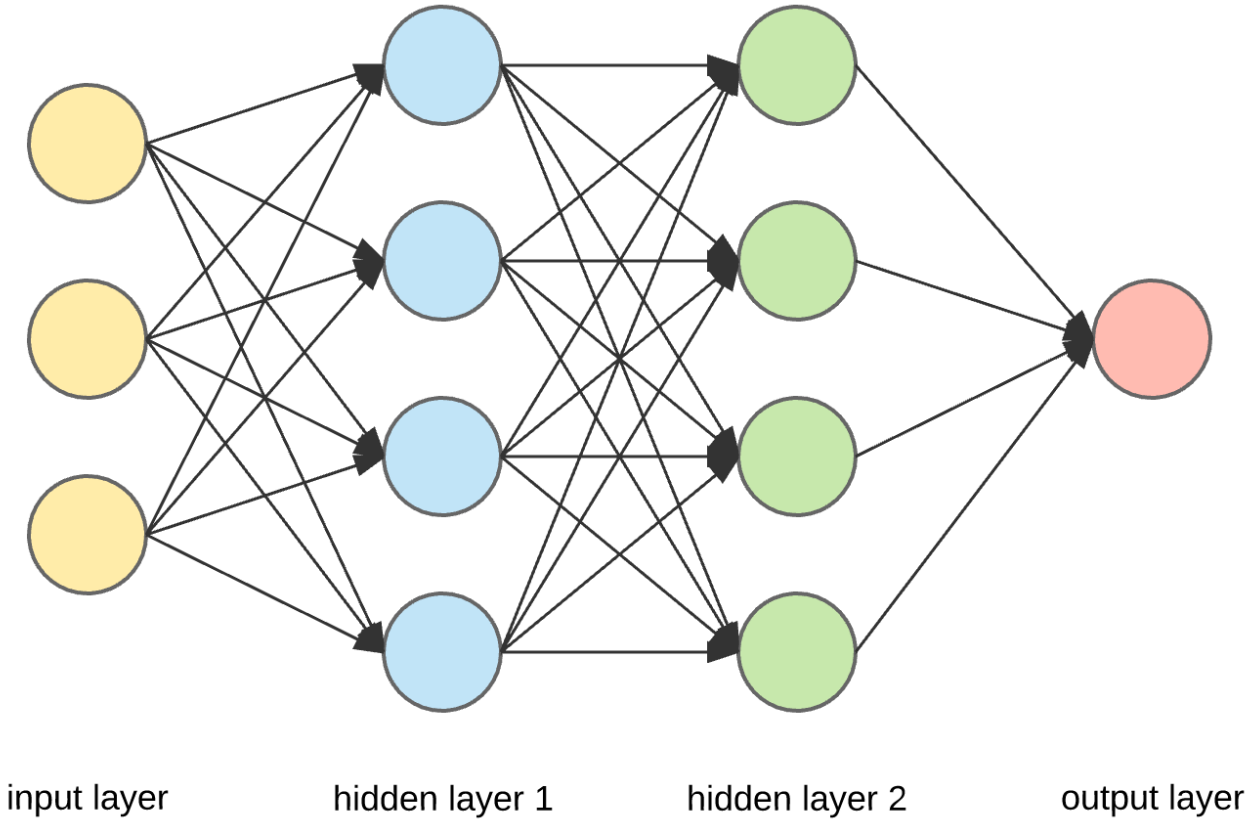
  For example, pattern recognizing. The ANN comes up with guesses while recognizing. Then the teacher provides the ANN with the answers. The network then compares it guesses with the teacher’s “correct” answers and makes adjustments according to errors.

- **Unsupervised Learning** − It is required when there is no example data set with known answers. For example, searching for a hidden pattern. In this case, clustering i.e. dividing a set of elements into groups according to some unknown pattern is carried out based on the existing data sets present.

- **Reinforcement Learning** − This strategy built on observation. The ANN makes a decision by observing its environment. If the observation is negative, the network adjusts its weights to be able to make a different required decision the next time.

Model Representation

Artificial Neural Network is computing system inspired by biological neural network that constitute animal brain. Such systems “learn” to perform tasks by considering examples, generally without being programmed with any task-specific rules.

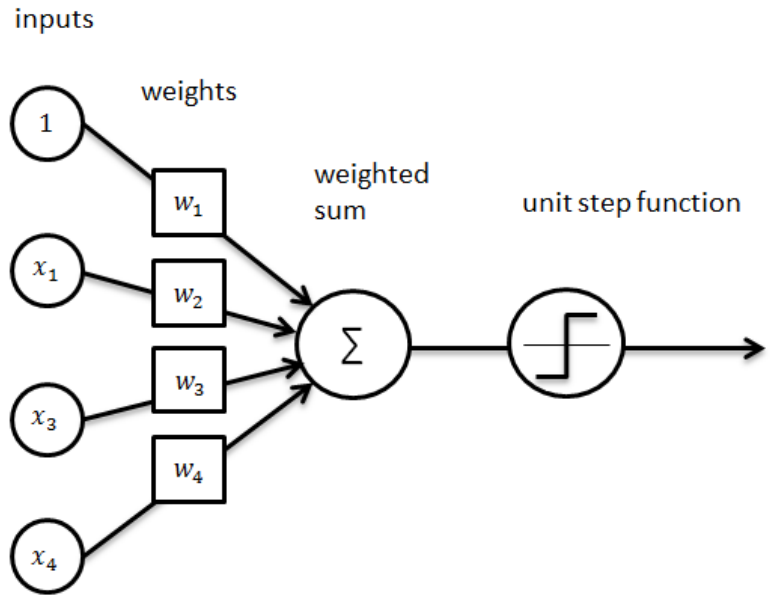


The Neural Network is constructed from 3 type of layers:

1. Input layer — initial data for the neural network.
2. Hidden layers — intermediate layer between input and output layer and place where all the computation is done.
3. Output layer — produce the result for given inputs.

There are 3 yellow circles on the image above. They represent the input layer and usually are noted as vector X. There are 4 blue and 4 green circles that represent the hidden layers. These circles represent the "activation" nodes and usually are noted as W or θ. The red circle is the output layer or the predicted value (or values in case of multiple output classes/types).

Each node is connected with each node from the next layer and each connection (black arrow) has particular weight. Weight can be seen as impact that that node has on the node from the next layer. So if we take a look on one node it would look like this



Let's look at the top blue node (*"Image 1"*). All the nodes from the previous layer (yellow) are connected with it. All these connections represent the weights (impact). When all the node values from the yellow layer are multiplied with their weight and all this is summarized it gives some value for the top blue node. The blue node has predefined "activation" function (*unit step function* on *"Image 2"*) which defines if this node will be "activated" or how "active" it will be, based on the summarized value. The additional node with value 1 is called "bias" node.

Model Representation Mathematics

In order to understand the mathematical equations I will use a simpler Neural Network model. This model will have 4 input nodes (3 + 1 "bias"). One hidden layer with 4 nodes (3 + 1 "bias") and one output node.

We are going to mark the "bias" nodes as x_0 and a_0 respectively. So, the input nodes can be placed in one vector *X* and the nodes from the hidden layer in vector *A*.

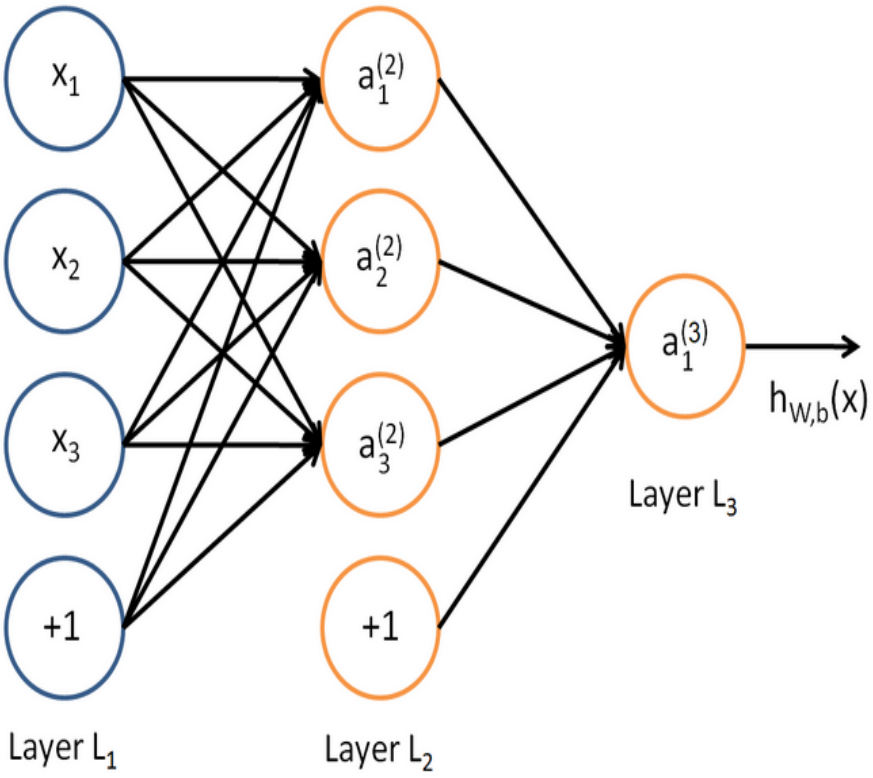


Image 3: Simple Neural Network

$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix}$$

--> x(input layer)
--> A(hidden layer)

The weights (arrows) are usually noted as θ or W. In this case I will note them as θ. The weights between the input and hidden layer will represent *3x4* matrix. And the weights between the hidden layer and the output layer will represent *1x4* matrix.

If network has **a** units in layer *j* and **b** units in layer *j*+1, then θ_j will be of dimension **b ×(a+1)**.

$$\theta^{(1)} = \begin{bmatrix} \theta_{10} & \theta_{11} & \theta_{12} & \theta_{13} \\ \theta_{20} & \theta_{21} & \theta_{22} & \theta_{23} \\ \theta_{30} & \theta_{31} & \theta_{32} & \theta_{33} \end{bmatrix}$$

Next, what we want is to compute the "activation" nodes for the hidden layer. In order to do that we need to multiply the input vector X and weights matrix θ^1 for the first layer $(X*\theta^1)$and then apply the activation function g. What we get is :

$$a_1^{(2)} = g(\Theta_{10}^{(1)} x_0 + \Theta_{11}^{(1)} x_1 + \Theta_{12}^{(1)} x_2 + \Theta_{13}^{(1)} x_3)$$
$$a_2^{(2)} = g(\Theta_{20}^{(1)} x_0 + \Theta_{21}^{(1)} x_1 + \Theta_{22}^{(1)} x_2 + \Theta_{23}^{(1)} x_3)$$
$$a_3^{(2)} = g(\Theta_{30}^{(1)} x_0 + \Theta_{31}^{(1)} x_1 + \Theta_{32}^{(1)} x_2 + \Theta_{33}^{(1)} x_3)$$

And by multiplying hidden layer vector with weights matrix θ for the second layer($A*\theta$) we get output for the hypothesis function:

$$h_\Theta(x) = a_1^{(3)} = g(\Theta_{10}^{(2)} a_0^{(2)} + \Theta_{11}^{(2)} a_1^{(2)} + \Theta_{12}^{(2)} a_2^{(2)} + \Theta_{13}^{(2)} a_3^{(2)})$$

This example is with only one hidden layer and 4 nodes there. If we try to generalize for Neural Network with multiple hidden layers and multiple nodes in each of the layers we would get next formula.

$$\mathrm{a}_n^L = \left[\sigma\left(\sum_m \theta_{nm}^L \left[\cdots\left[\sigma\left(\sum_j \theta_{kj}^2 \left[\sigma\left(\sum_i \theta_{ji}^1 x_i + b_j^1\right)\right] + b_k^2\right)\right]\cdots\right]_m + b_n^L\right)\right]_n$$

Where we have *L* layers with *n* nodes and *L-1* layer with *m* nodes.

Appropriate problems for neural network learning

Neural networks are broadly used for real world business problems such as sales forecasting, customer research, data validation, and risk management.

Marketing

Target marketing involves market segmentation, where we divide the market into distinct groups of customers with different consumer behavior.

Neural networks are well-equipped to carry this out by segmenting customers according to basic characteristics including demographics, economic status, location, purchase patterns, and attitude towards a product. Unsupervised neural networks can be used to automatically group and segment customers based on the similarity of their characteristics, while supervised neural networks can be trained to learn the boundaries between customer segments based on a group of customers.

Retail & Sales

Neural networks have the ability to simultaneously consider multiple variables such as market demand for a product, a customer's income, population, and product price. Forecasting of sales in supermarkets can be of great advantage here.

If there is a relationship between two products over time, say within 3–4 months of buying a printer the customer returns to buy a new cartridge, then retailers can use this information to contact the customer, decreasing the chance that the customer will purchase the product from a competitor.

Banking & Finance

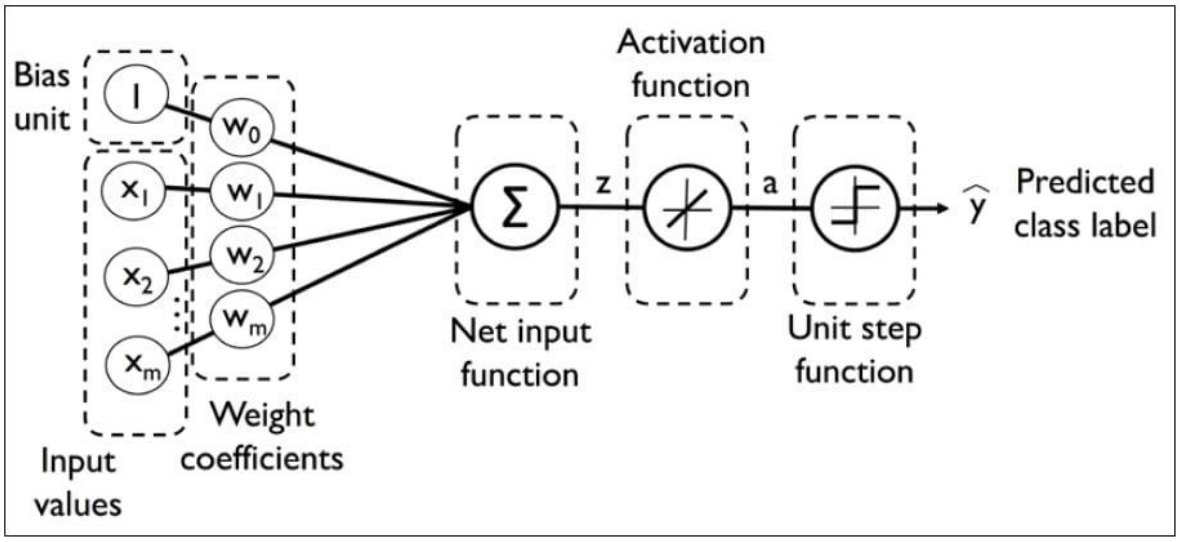
Neural networks have been applied successfully to problems like derivative securities pricing and hedging, futures price forecasting, exchange rate forecasting, and stock performance. Traditionally, statistical techniques have driven the software. These days, however, neural networks are the underlying technologies driving decision making.

Medicine

It is a trending research area in medicine and it is believed that they will receive an extensive application to biomedical systems in the next few years. At the moment, the research is mostly on modeling parts of the human body and recognizing diseases from various scans.

Single-layer ANN - A RECAP

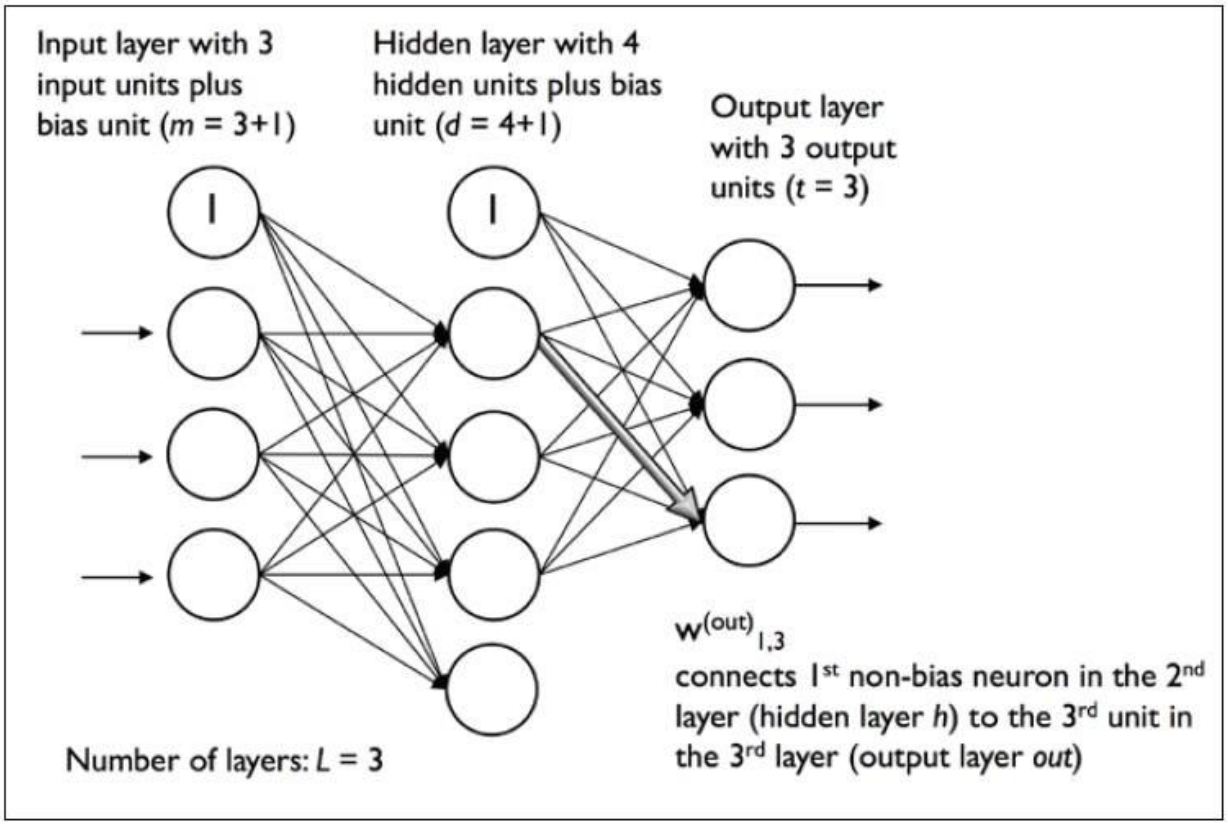
Perceptron rule and Adaline rule were used to train a single-layer neural network.



Weights are updated based on a unit function in perceptron rule or on a linear function in Adaline Rule.

Multi-layer ANN

A fully connected multi-layer neural network is called a Multilayer Perceptron (MLP).



It has 3 layers including one hidden layer. If it has more than 1 hidden layer, it is called a deep ANN.

An MLP is a typical example of a feedforward artificial neural network.

In this figure, the i_{th} activation unit in the l_{th} layer is denoted as $a_{i(l)}$.

The number of layers and the number of neurons are referred to as hyperparameters of a neural network, and these need tuning. Cross-validation techniques must be used to find ideal values for these.

The weight adjustment training is done via backpropagation. Deeper neural networks are better at processing data. However, deeper layers can lead to vanishing gradient problem. Special algorithms are required to solve this issue.

Notations

In the representation below:

$$a^{(in)} = \begin{bmatrix} a_0^{(in)} \\ a_1^{(in)} \\ \vdots \\ a_m^{(in)} \end{bmatrix} = \begin{bmatrix} 1 \\ x_1^{(in)} \\ \vdots \\ x_m^{(in)} \end{bmatrix}$$

- $a_{i(in)}$ refers to the i_{th} value in the input layer
- $a_{i(h)}$ refers to the i_{th} unit in the hidden layer
- $a_{i(out)}$ refers to the i_{th} unit in the output layer
- $a_{o(in)}$ is simply the bias unit and is equal to 1; it will have the corresponding weight w_0
- The weight coefficient from layer l to layer l+1 is represented by $w_{k,j(l)}$

A simplified view of the multilayer is presented here.

This image shows a fully connected three-layer neural network with 3 input neurons and 3 output neurons. A bias term is added to the input vector.

Forward Propagation

In the following topics, we discuss the forward propagation in detail.

MLP Learning Procedure

The MLP learning procedure is as follows :

- Starting with the input layer, propagate data forward to the output layer. This step is the forward propagation.
- Based on the output, calculate the error (the difference between the predicted and known outcome). The error needs to be minimized.
- Backpropagate the error. Find its derivative with respect to each weight in the network, and update the model.

Repeat the three steps given above over multiple epochs to learn ideal weights.

Finally, the output is taken via a threshold function to obtain the predicted class labels.

Forward Propagation in MLP

In the first step, calculate the activation unit al(h) of the hidden layer.

$$z_1^{(h)} = a_0^{(in)} w_{0,1}^{(h)} + a_1^{(in)} w_{1.1}^{(h)} + \cdots + a_m^{(in)} w_{m,1}^{(h)}$$

$$a_1^{(h)} = \phi\left(z_1^{(h)}\right)$$

Activation unit is the result of applying an activation function φ to the z value. It must be differentiable to be able to learn weights using gradient descent.

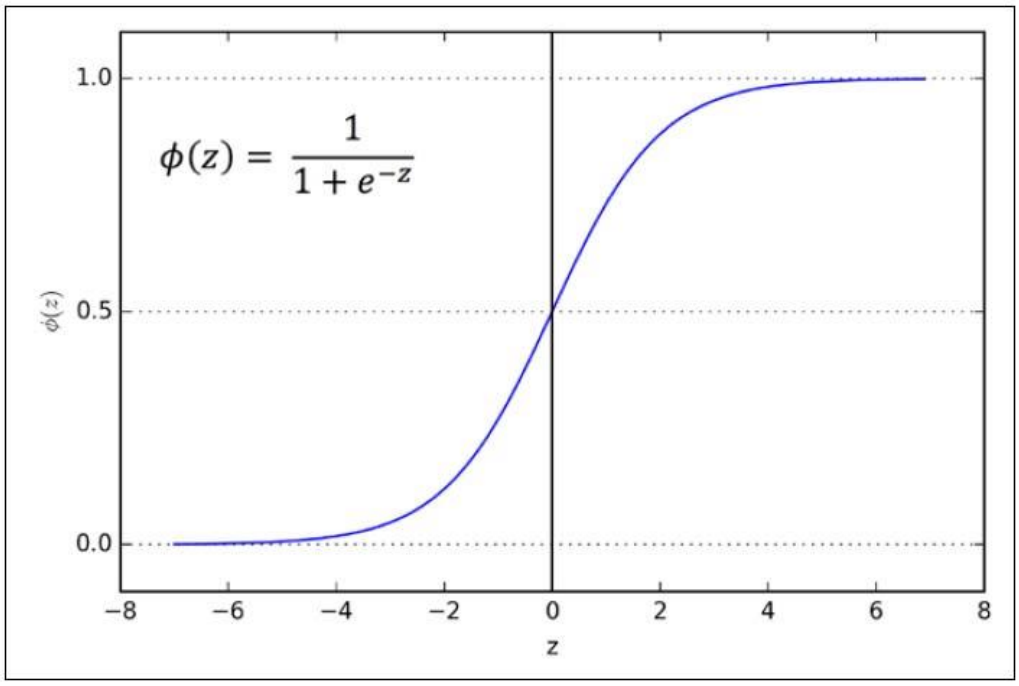
The activation function φ is often the sigmoid (logistic) function.

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

It allows nonlinearity needed to solve complex problems like image processing.

Sigmoid Curve

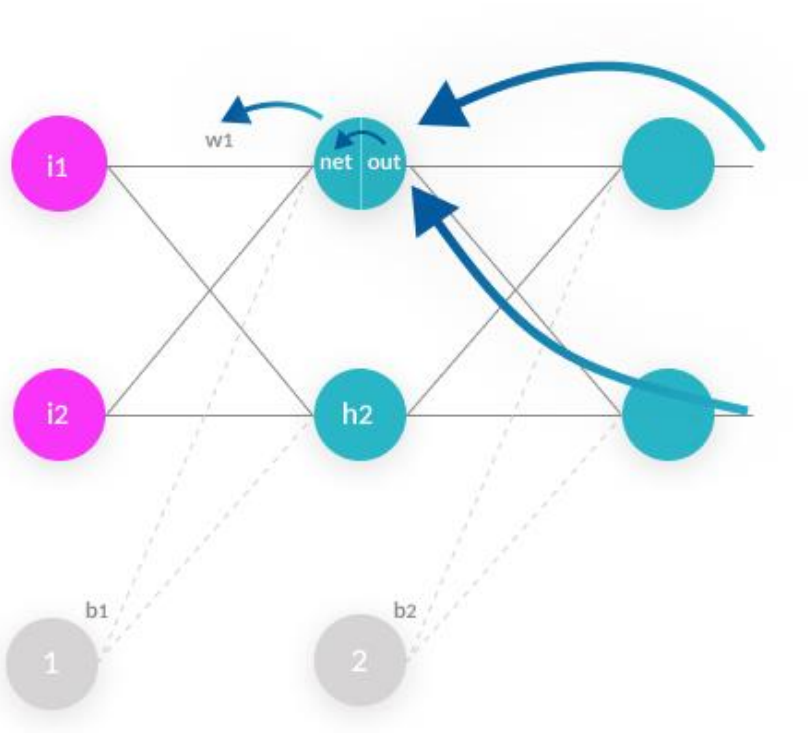
The sigmoid curve is an S-shaped curve.



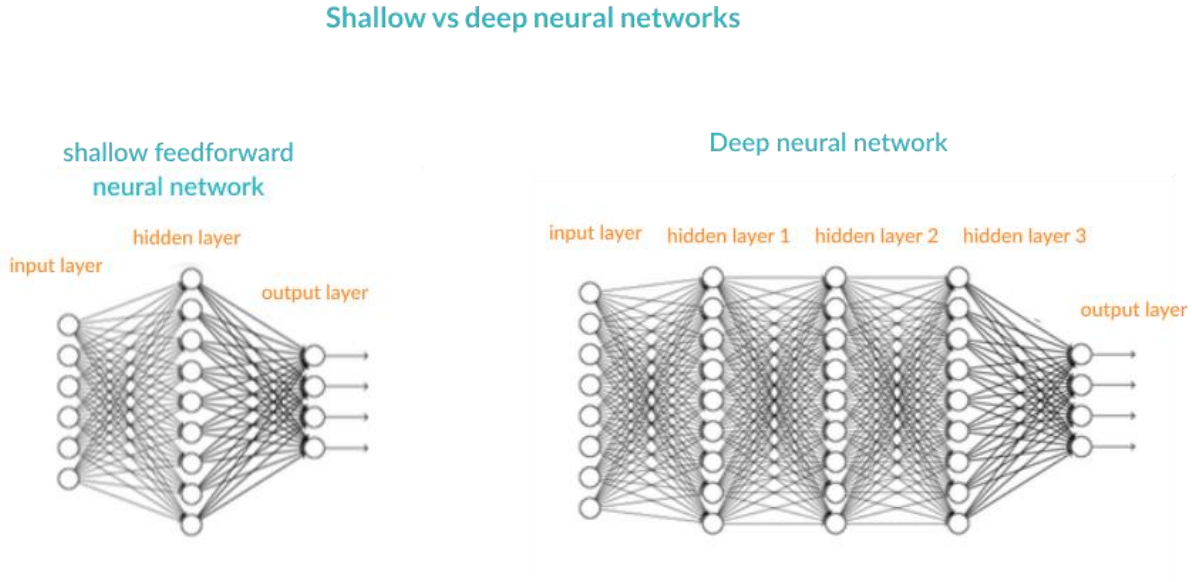
What is backpropagation?

Backpropagation is an algorithm commonly used to train neural networks. When the neural network is initialized, weights are set for its individual elements, called neurons. Inputs are loaded, they are passed through the network of neurons, and the network provides an output for each one, given the initial weights. Backpropagation helps to adjust the weights of the neurons so that the result comes

closer and closer to the known true result.



Artificial Neural Networks (ANN) are a mathematical construct that ties together a large number of simple elements, called neurons, each of which can make simple mathematical decisions. Together, the neurons can tackle complex problems and questions, and provide surprisingly accurate answers. A shallow neural network has three layers of neurons that process inputs and generate outputs. A Deep Neural Network (DNN) has two or more "hidden layers" of neurons that process inputs. According to Goodfellow, Bengio and Courville, and other experts, while shallow neural networks can tackle equally complex problems, deep learning networks are more accurate and improve in accuracy as more neuron layers are added.



ANN and DNN Concepts Relevant to Backpropagation

Here are several neural network concepts that are important to know before learning about backpropagation:

- 

  ### Inputs

  Source data fed into the neural network, with the goal of making a decision or prediction about the data. The data is broken down into binary signals, to allow it to be processed by single neurons—for example an image is input as individual pixels.

- 

  ### Training Set

  A set of outputs for which the correct outputs are known, which can be used to train the neural networks.

- 

  ### Outputs

  The output of the neural network can be a real value between 0 and 1, a boolean, or a discrete value (for example, a category ID).

- 

  ### Activation Function

  Each neuron accepts part of the input and passes it through the activation function. Commonly used functions are the sigmoid function, tanh and ReLu. Modern activation functions normalize the output to a given range, to ensure the model has stable convergence.

- 

Weight Space

Each neuron is given a numeric weight. The weights, applied to the activation function, determine each neuron's output. In training of a deep learning model, the objective is to discover the weights that can generate the most accurate output.

- 

Initialization

Setting the weights at the beginning, before the model is trained. A typical strategy in neural networks is to initialize the weights randomly, and then start optimizing from there. Xavier optimization is another approach which makes sure weights are "just right" to ensure enough signal passes through all layers of the network.

- 

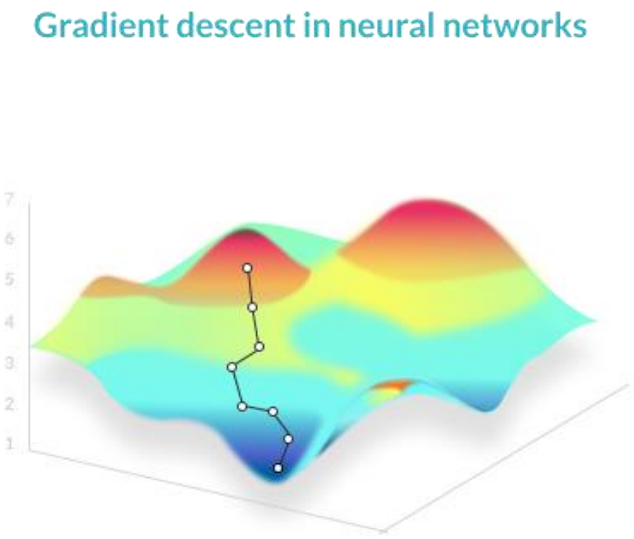
Forward Pass

The forward pass tries out the model by taking the inputs, passing them through the network and allowing each neuron to react to a fraction of the input, and eventually generating an output.

- 

Gradient Descent

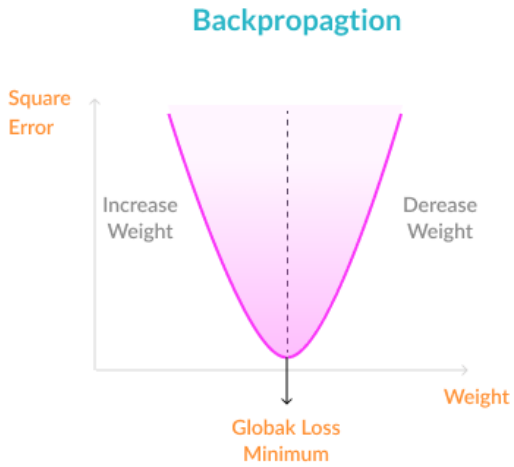
A mathematical technique that modifies the parameters of a function to descend from a high value of a function to a low value, by looking at the derivatives of the function with respect to each of its parameters, and seeing which step, via which parameter, is the next best step to minimize the function. Applying gradient descent to the error function helps find weights that achieve lower and lower error values, making the model gradually more accurate.



6 Stages of Neural Network Learning

Generally speaking, neural network or deep learning model training occurs in six stages:

1. **Initialization**—initial weights are applied to all the neurons.
2. **Forward propagation**—the inputs from a training set are passed through the neural network and an output is computed.
3. **Error function**—because we are working with a training set, the correct output is known. An error function is defined, which captures the delta between the correct output and the actual output of the model, given the current model weights (in other words, "how far off" is the model from the correct result).
4. **Backpropagation**—the objective of backpropagation is to change the weights for the neurons, in order to bring the error function to a minimum.



5. **Weight update—**weights are changed to the optimal values according to the results of the backpropagation algorithm.
6. **Iterate until convergence**—because the weights are updated a small delta step at a time, several iterations are required in order for the network to learn. After each iteration, the gradient descent force updates the weights towards less and less global loss function. The

amount of iterations needed to converge depends on the learning rate, the network meta-parameters, and the optimization method used.

At the end of this process, the model is ready to make predictions for unknown input data. New data can be fed to the model, a forward pass is performed, and the model generates its prediction.

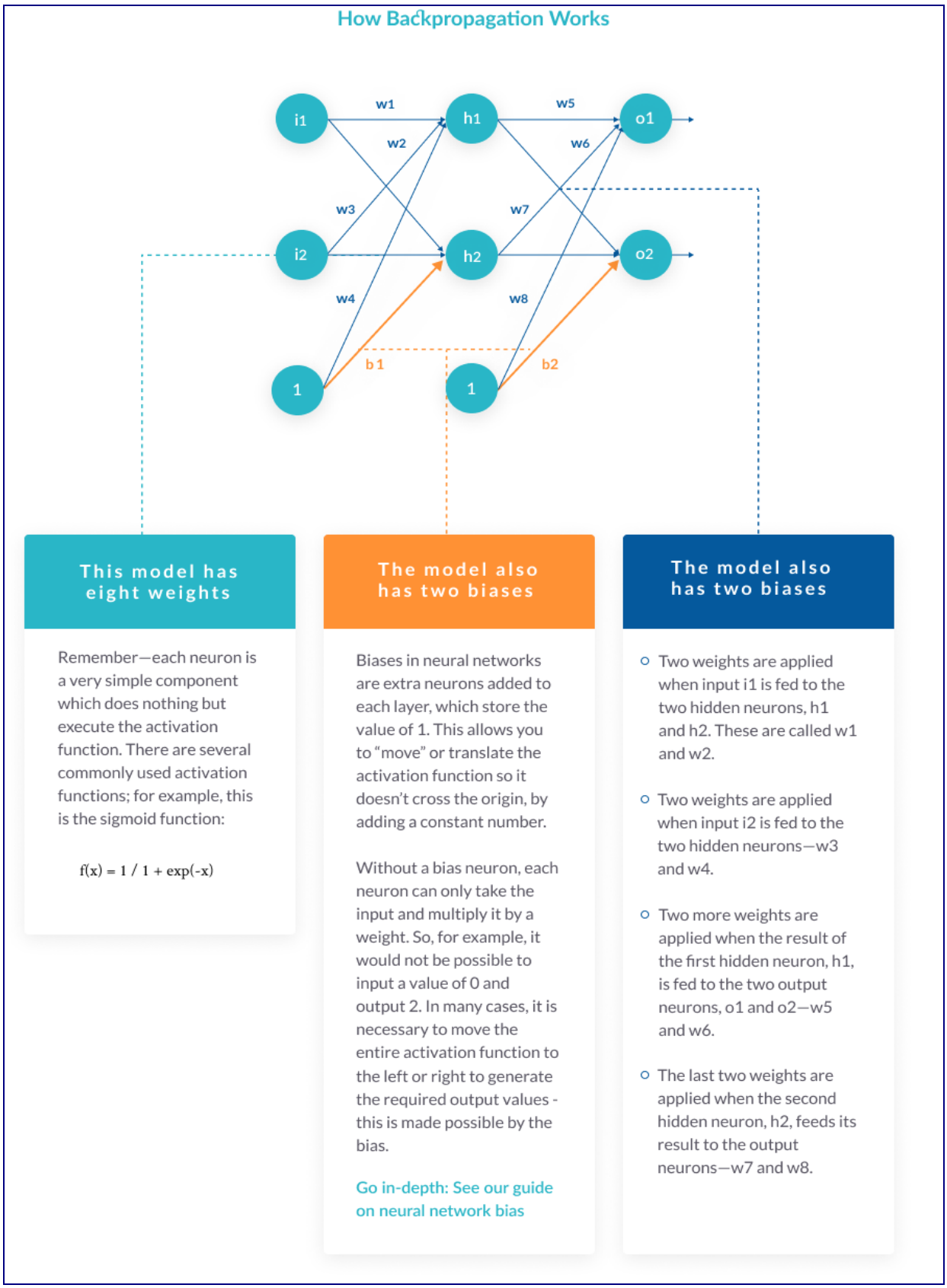
Why Do We Need Backpropagation in Neural Networks?

In the six stages of learning we presented above, step #4 can be done by any optimization function that can reduce the size of the error in the model. For example, you could do a brute force search to try to find the weight values that bring the error function to a minimum.

Brute force or other inefficient methods could work for a small example model. But in a realistic deep learning model which could have as its output, for example, 600X400 pixels of an image, with 3-8 hidden layers of neurons processing those pixels, you can easily reach a model with millions of weights. This is why a more efficient optimization function is needed.

Backpropagation is simply an algorithm which performs a highly efficient search for the optimal weight values, using the gradient descent technique. It allows you to bring the error functions to a minimum with low computational resources, even in large, realistic models.

How Backpropagation Works



The image above is a very simple neural network model with two inputs (i1 and i2), which can be real values between 0 and

1, two hidden neurons (h1 and h2), and two output neurons (o1 and o2).

The model also has two biases

Biases in neural networks are extra neurons added to each layer, which store the value of 1. This allows you to "move" or translate the activation function so it doesn't cross the origin, by adding a constant number.

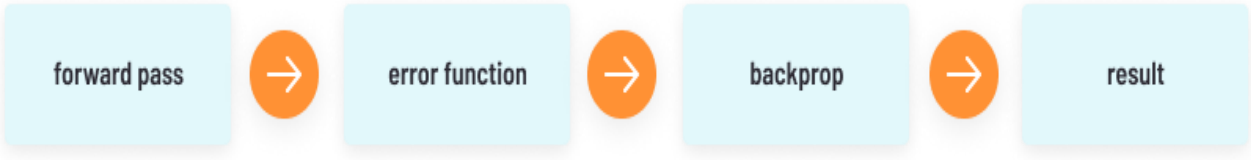
Without a bias neuron, each neuron can only take the input and multiply it by a weight. So, for example, it would not be possible to input a value of 0 and output 2. In many cases, it is necessary to move the entire activation function to the left or right to generate the required output values – this is made possible by the bias.

Go in-depth: see our guide on neural network bias

What the neurons do

Remember—each neuron is a very simple component which does nothing but executes the activation function. There are several commonly used activation functions; for example, this is the sigmoid function:

f(x) = 1 / 1 + exp(-x)



The forward pass

Our simple neural network works by:

1. Taking each of the two inputs
2. Multiplying by the first-layer weights—w1,2,3,4
3. Adding bias
4. Applying the activation function for neurons h1 and h2
5. Taking the output of h1 and h2, multiplying by the second layer weights—w5,6,7,8
6. This is the output.

To take a concrete example, say the first input i1 is 0.1, the weight going into the first neuron, w1, is 0.27, the second input i2 is 0.2, the weight from the second weight to the first neuron, w3, is 0.57, and the first layer bias b1 is 0.4.

The input of the first neuron h1 is combined from the two inputs, i1 and i2:

(i1 * w1) + (i2 * w2) + b1 = (0.1 * 0.27) + (0.2 * 0.57) + (0.4 * 1) = 0.541

Feeding this into the activation function of neuron h1:

f(0.541) = 1 / (1 + exp(-0.541)) = **0.632**

Now, given some other weights w2 and w4 and the second input i2, you can follow a similar calculation to get an output for the second neuron in the hidden layer, h2.

The final step is to take the outputs of neurons h1 and h2, multiply them by the weights w5,6,7,8, and feed them to the same activation function of neurons o1 and o2 (exactly the same calculation as above).

The result is the final output of the neural network—let's say **the final outputs are 0.735 for o1 and 0.455 for o2**.

We'll also assume that the **correct output values are 0.5 for o1 and 0.5 for o2** (these are assumed correct values because in supervised learning, each data point had its truth value).

The error function

The error function For simplicity, we'll use the Mean Squared Error function. For the first output, the error is the correct output value minus the actual output of the neural network:

0.5—0.735 = -0.235

For the second output:

0.5—0.455 = 0.045

Now we'll calculate the Mean Squared Error:

MSE(o1) = ½ $(-0.235)^2$ = 0.0276

MSE(o2) = ½ $(0.045)^2$ = 0.001

The Total Error is the sum of the two errors:

Total Error = 0.0276 + 0.001 = **0.0286**

This is the number we need to minimize with backpropagation.

Backpropagation with gradient descent

The backpropagation algorithm calculates how much the final output values, o1 and o2, are affected by each of the weights. To do this, it calculates partial derivatives, going back from the error function to the neuron that carried a specific weight.

For example, weight w6, going from hidden neuron h1 to output neuron o2, affected our model as follows:

neuron h1 with weight w6 → affects total input of neuron o2 → affects output o2 → affects total errors

Backpropagation goes in the opposite direction:

total errors → affected by output o2 → affected by total input of neuron o2 → affected by neuron h1 with weight w6
The algorithm calculates three derivatives:

The algorithm calculates three derivatives:

- The derivative of total errors with respect to output o2
- The derivative of output o2 with respect to total input of neuron o2
- Total input of neuron o2 with respect to neuron h1 with weight w6

This gives us complete traceability from the total errors, all the way back to the weight w6.

Using the Leibniz Chain Rule, it is possible to calculate, based on the above three derivatives, what is the optimal value of w6 that minimizes the error function. **In other words, what is the "best" weight w6 that will make the neural network most accurate?**

Similarly, the algorithm calculates an optimal value for each of the 8 weights. Or, in a realistic model, for each of thousands or millions of weights used for all neurons in the model.

End result of backpropagation

The backpropagation algorithm results in a set of optimal weights, like this:

Optimal w1 = 0.355

Optimal w2 = 0.476

Optimal w3 = 0.233

Optimal w4 = 0.674

Optimal w5 = 0.142

Optimal w6 = 0.967

Optimal w7 = 0.319

Optimal w8 = 0.658

You can update the weights to these values, and start using the neural network to make predictions for new inputs.

How Often Are the Weights Updated?

There are three options for updating weights during backpropagation:

- **Updating after every sample in training set**—running a forward pass for every sample, calculating optimal weights and updating. The downside is that this can be time-consuming for large training sets, and outliers can throw off the model and result in the selection of inappropriate weights.

-

Updating in batch—dividing training samples into several large batches, running a forward pass on all training samples in a batch, and then calculating backpropagation on all the samples together. Training is performed iteratively on each of the batches. This makes the model more resistant to outliers and variance in the training set.

- 

Randomized mini-batches—a compromise between the first two approaches is to randomly select small batches from the training data, and run forward pass and backpropagation on each batch, iteratively. This avoids a biased selection of samples in each batch, which can lead to the of a local optimum.