1. Computer Vision and Visual Perception

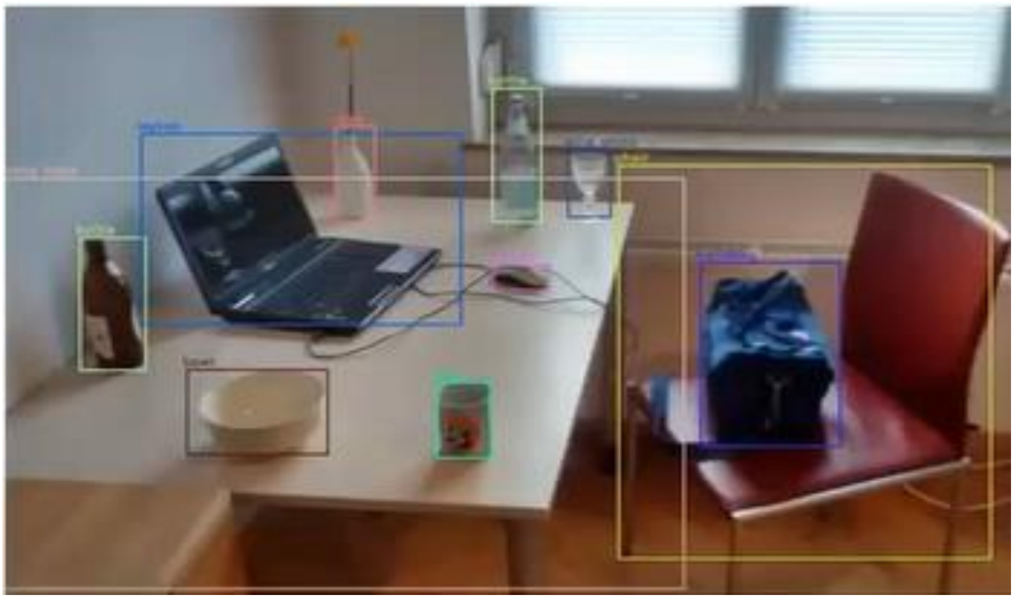
Any AI system can perceive the environment and take actions based on its perceptions.

What is Computer vision?

Computer vision is a field of artificial intelligence (AI) and computer science that focuses on enabling computers to interpret and understand visual information from the world, much like humans do with their eyes and brain.

It involves in the *development of algorithms and techniques* that allow machines to extract meaningful insights and make decisions based on images or video data.

- Ability of Computers
    - to understand and interpret Visual data
        - For example: Images, Videos…
- Automate tasks
    - Which human visual can perform



A human not only identifies the objects (Object detection) in an image but also determines their positions (Localization) within the image.

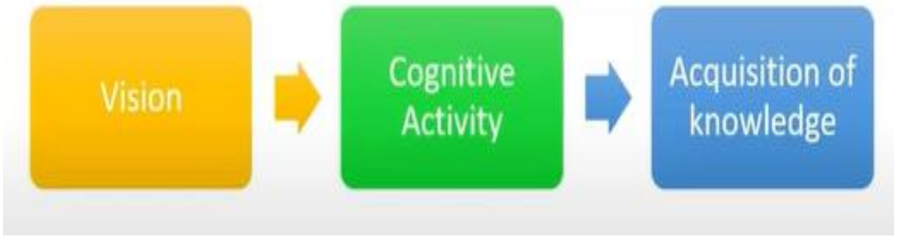
What is visual perception?

"***Visual perception*** is the act of observing patterns and objects through sight or visual input. "

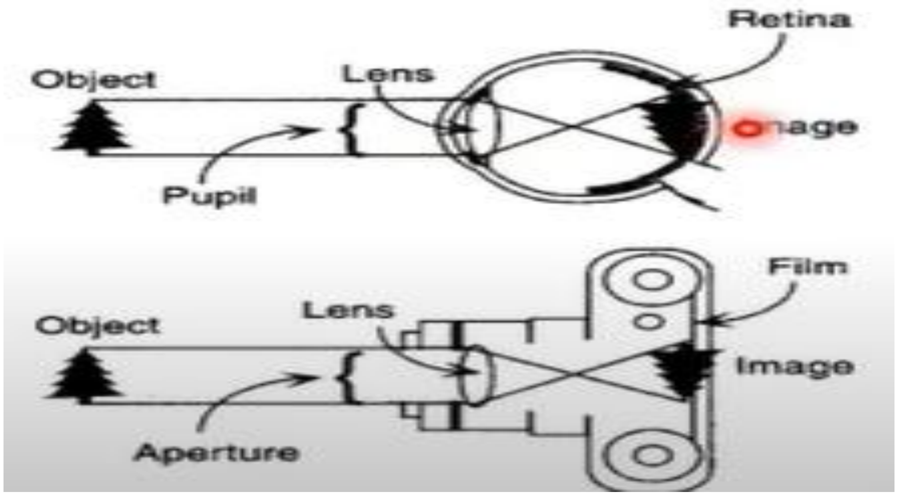
Definition: Process of acquiring knowledge about environmental objects and events by extracting information from the light they emit or reflect (Palmer, 2012)

Visual perception is also defined as the ability to interpret the surrounding environment through vision (photopic vision (daytime vision), colour vision, scotopic vision (night vision), and mesopic vision (twilight vision)) using light in the visible spectrum reflected by objects in the environment.

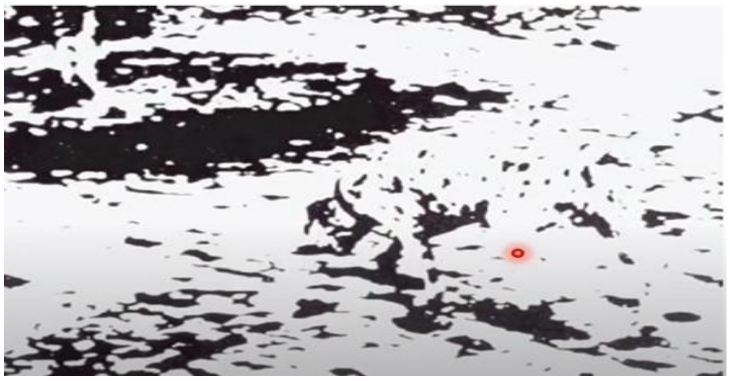
Steps involved in Visual perception:

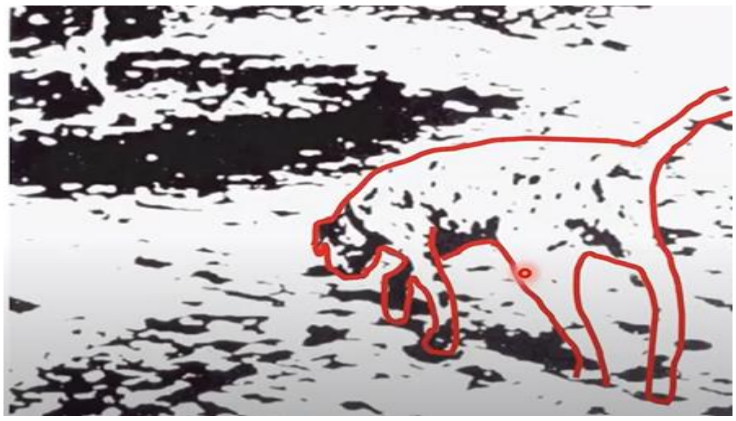


Analogy behind the capture of images by human and by camera:



The whole process of visual perception involves not just seeing, but also accurately interpreting what is seen.





For example, consider autonomous vehicle, visual perception means understanding the surrounding objects and their specific details
—such as pedestrians, or whether there is a particular lane the vehicle needs to be centred in.
—and detecting traffic signs and understanding what they mean.

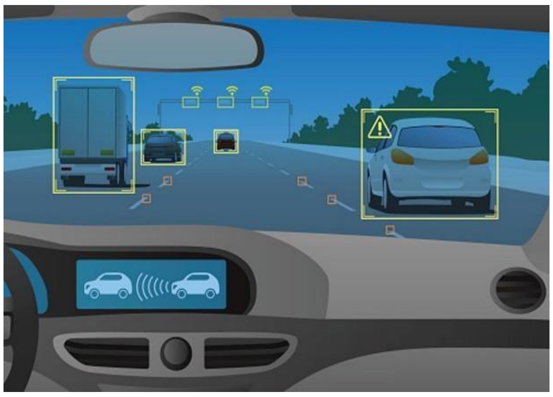


Fig 1a. Visual perception

2. Types of Vision system:

Traditional image-processing techniques were considered as CV systems, but that is not totally accurate. Processing an image by a machine is far different from that machine understanding, what is happening within the image, which is not a trivial task. Image processing is now just a piece of a bigger, more complex system that aims to interpret image content.

a. Human Vision Systems:

The vision system for humans, animals and most living organisms consists of a sensor or an eye to capture the image and a brain to process & interpret the image. The system outputs a prediction of the image components based on the data extracted from the image.

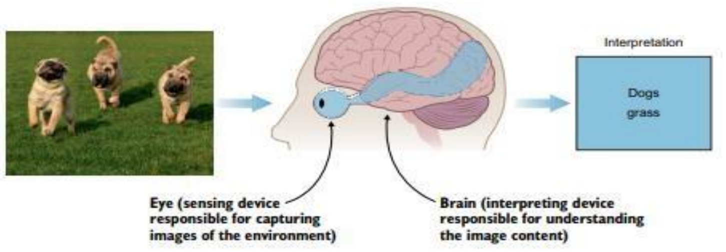

Interpretation
Dogs
grass
Eye (sensing device responsible for capturing images of the environment)
Brain (interpreting device responsible for understanding the image content)

Figure2a: The human vision system uses the eye and brain to sense and interpret an image.

human vision system helps to interpret the image of dogs as shown in figure2a looks at it and directly understands that the image consists of a bunch of dogs (three, to be specific). It comes pretty natural to us to classify and detect objects in this image because we have been trained over the years to identify dogs.

b. AI Vision Systems:

To replicate human vision, two key components are required: a sensing device that mimics the eye's functionality, and a sophisticated algorithm that replicates the brain's ability to interpret and categorize visual content as shown in figure2b.

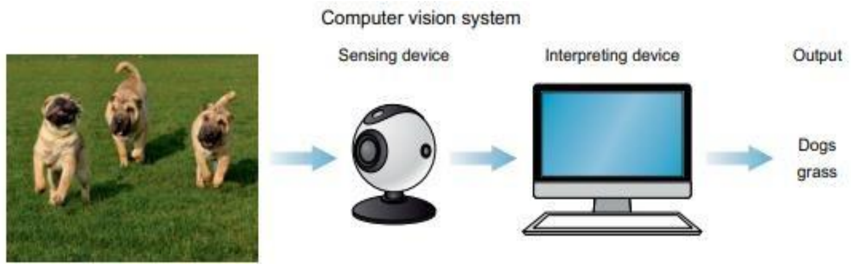
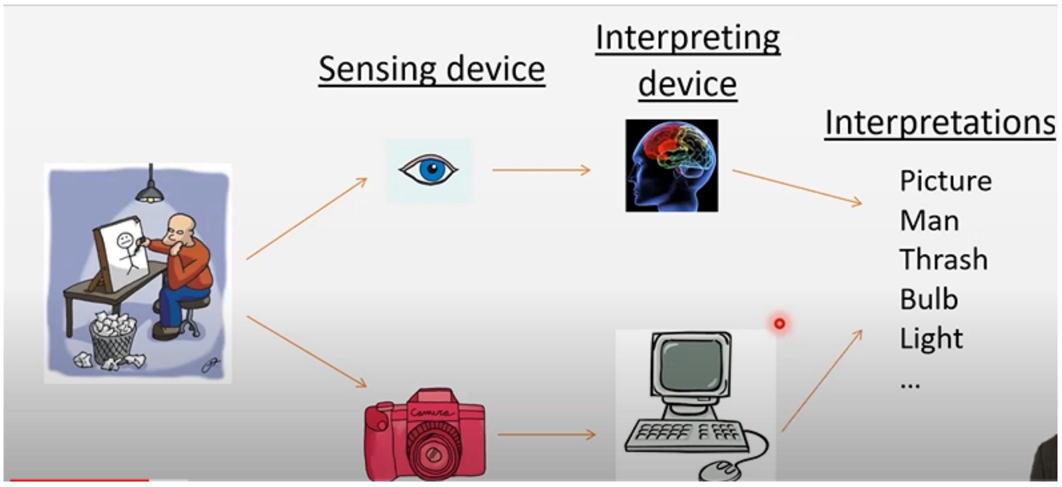


Figure2b: The components of the computer vision system are a sensing device and an interpreting device.

Human Vision vs Computer Vision:



Example in Machine Learning Perspective:

- Learn to identify dogs by seeing labelled images.
  In supervised learning labelled data, data for which target is known.
  Eg: Shown a sample image of a dog and told that it was a dog. The human brain learned to associate the features and say the label: dog.
- Predict the classification for different object, a horse, shown and ask to identify. First, human brain thought it was a dog, because it hadn't seen horses before, and there will be a confusion of horse features with dog features. So, the prediction goes wrong.
- Now the brain starts adjusting the parameters to learn horse features. Now it will understand "Yes, both have four legs, but the horse's legs are longer. Longer legs indicate a horse." So, the brain starts learning until it doesn't make no mistakes. This is called training by trial and error.

Comparison of Image Processing Vs Computer Vision:

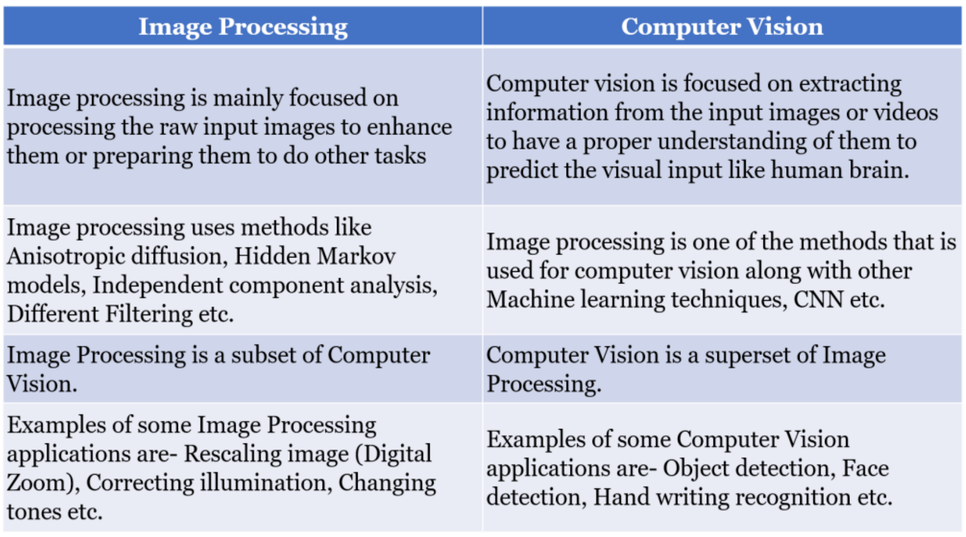


Image Processing	Computer Vision
Image processing is mainly focused on processing the raw input images to enhance them or preparing them to do other tasks	Computer vision is focused on extracting information from the input images or videos to have a proper understanding of them to predict the visual input like human brain.
Image processing uses methods like Anisotropic diffusion, Hidden Markov models, Independent component analysis, Different Filtering etc.	Image processing is one of the methods that is used for computer vision along with other Machine learning techniques, CNN etc.
Image Processing is a subset of Computer Vision.	Computer Vision is a superset of Image Processing.
Examples of some Image Processing applications are- Rescaling image (Digital Zoom), Correcting illumination, Changing tones etc.	Examples of some Computer Vision applications are- Object detection, Face detection, Hand writing recognition etc.

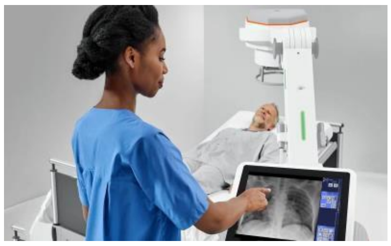
3. Basic components for Human and Machine Vision Systems:

Both human and machine vision systems have two components a ***sensing device*** and an ***interpreting device***. Each is tailored to fulfil a specific task.

a. Sensing devices:

The important aspect in selecting a best sensing device to capture the surroundings of a specific environment, whether that is a camera, radar, X-ray, CT scan, Lidar, or a combination of devices to provide the full scene of an environment to fulfill the task at hand.

Camera


X Ray

Lidar in Drones

For example, the main goal of the *autonomous vehicle (AV)* vision system is to allow the car to understand the environment around it and move from point A to point B safely and in a timely manner. To fulfill this goal, vehicles are equipped vehicles the role of cameras and sensors are to detect 360 degrees of movement—pedestrians, cyclists, vehicles, roadwork, and other objects—from up to three football fields away.

Example2 for sensing devices used in *self-driving cars* is to perceive the surrounding area:

- LIDAR, a radar-like technique, uses invisible pulses of light to create a high resolution 3D map of the surrounding area.
- Cameras can see street signs and road markings but cannot measure distance.
- Radar can measure distance and velocity but cannot see in fine detail.

Medical diagnosis applications use X-rays or CT scans as sensing devices.

b. Interpreting devices

Interpreting devices in vision systems take the output image from the sensing device and learn features and patterns to identify objects. So, we need to build a brain. Thus, we have ANNs and CNNs

Artificial neural networks (ANNs) as shown in **Fig.3. a.** The analogy between the biological neurons and artificial systems has a main processing element, a neuron, with input signals (x1, x2, …, xn) and an output.

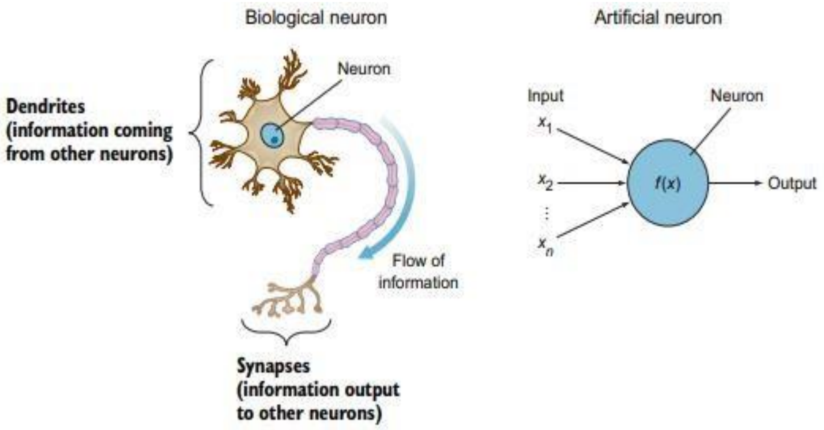


Figure.3. a: Similarities between biological neurons and artificial systems

Another system that connects millions of these neurons stacked in layers that is each neuron is connected to thousands of other neurons, yielding a learning behaviour.

Building a multilayer neural network is called Deep Learning as shown in **Fig.3. a.** The DL methods learn the representations through a sequence of transformations of data through layers of neurons. The different DL architectures are ANNs and Convolutional neural networks CNNs.

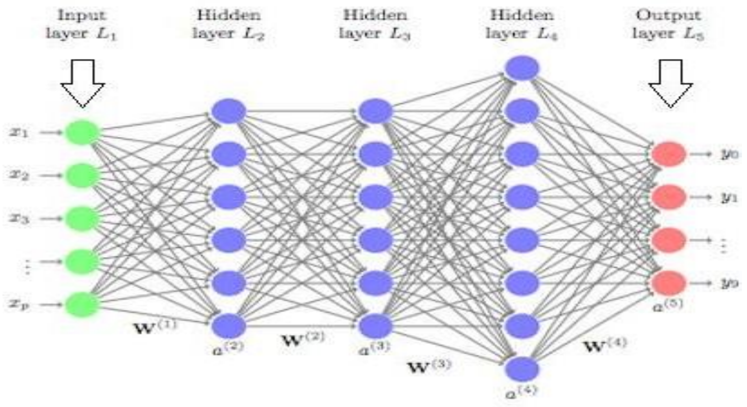


Fig. 3.b: Deep learning involves layers of neurons in a network.

4. Applications of Computer Vision:

Computers are able to recognize human faces in images decades ago, but now AI systems are able to classify objects in photos and videos because of the computational power and the amount of data availability.

The deep neural networks are successful for CV, natural language processing and voice user interface tasks.

AI and DL have managed to achieve best performance on many complex visual perception tasks like image or video search, captioning classification, and object detection.

a. Image classification is the task of assigning to an image a label from a predefined set of categories. A Convolutional neural network is a neural network type that truly shines in processing and classifying images in many different applications:

Lung cancer diagnosis—Lung cancer is a growing problem. The main reason lung

cancer is very dangerous is that when it is diagnosed, it is usually in the middle or of late stages. When diagnosing lung cancer, doctors typically use their eyes to examine CT scan images, looking for small nodules in the lungs. In the early stages, the nodules are usually very small and hard to spot. (figure 4a).

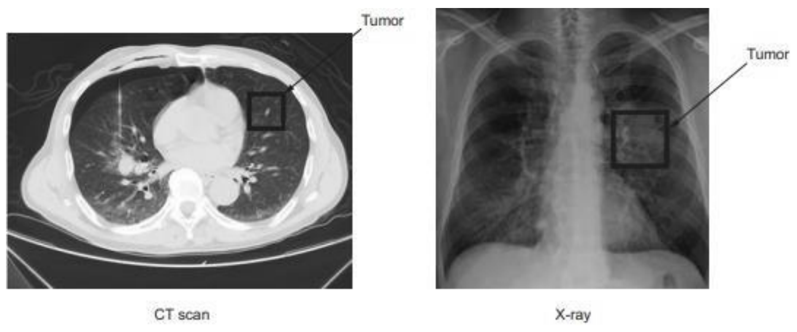


Figure 4a: X-ray images to identify tumors in earlier stages of development.

b. Traffic sign recognition—Traditionally, standard CV methods were employed to detect and classify traffic signs, but this approach required time-consuming manual work to handcraft important features in images. Instead, by applying DL to this problem, we can create a model that reliably classifies traffic signs, learning to identify the most appropriate features for this problem by itself (figure 4b).

Figure4b : Vision systems can detect traffic signs with very high performance.

c. Object detection and localization:

Localization involves determining the object's position and outlining it with a bounding box. On the other hand, object detection goes beyond by identifying and categorizing all objects within the image. Each object is assigned a class label and enclosed with a bounding box. To do that, we can build object detection systems like YOLO (you only look once), SSD (single- shot detector), and Faster R-CNN, which not only classify images but also can locate and detect each object in images that contain multiple objects as shown in (figure 4c).

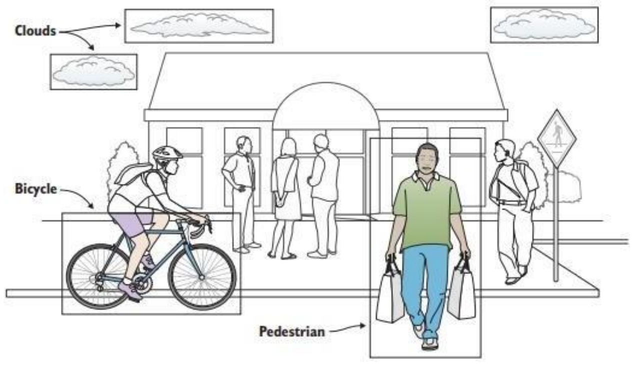


Figure.4c: Deep learning systems can segment objects in an image.

d. Generating art (style transfer)

Neural style transfer, one of the most interesting CV applications, is used to transfer the style from one image to another. Neural style transfer is a method that optimizes three images: a content image, a style reference image (like an artwork), and an input image you want to style. It combines these images to make the input image adopt the content of one image while being artistically transformed to match the style of another image, like a painting. (figure.4d).

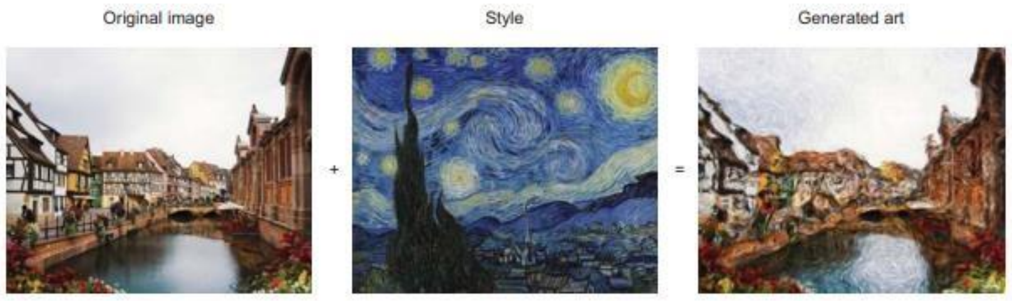


Figure.4d: Style transfer onto the original image, producing a piece of art that feels

e. Creating images

A new DL model that can imagine new things called generative adversarial networks (GANs). GANs are sophisticated DL models that generate stunningly accurate synthesized images of objects, people, and places, among other things. If you give them a set of images, they can make entirely new, realistic-looking images. For example, StackGAN is one of the GAN architecture variations that can use a textual description of an object to generate a high-resolution image of the object matching that description. This is not just running an image search on a database. These “photos” have never been seen before and are totally imaginary (figure 4e)

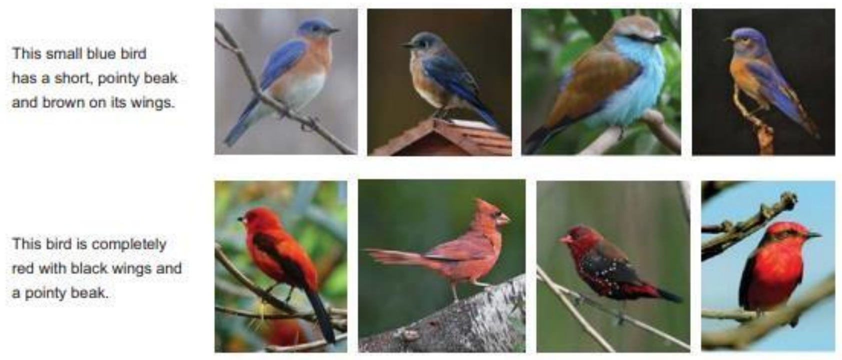
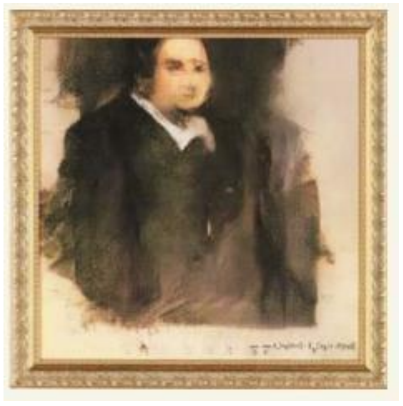

This small blue bird has a short, pointy beak and brown on its wings.
This bird is completely red with black wings and a pointy beak.

Figure 4e: Generative adversarial networks (GANS) can create new, “made-up” images from a set of existing images.



AI-generated artwork featuring a fictional person named Edmond de Belamy sold for $432,500. The artwork was created by a team of three 25-year-old French students using GANs. The network was trained on a dataset of 15,000 portraits painted between the fourteenth and twentieth century’s, and then it created one of its own. The team printed the image, framed it, and signed it with part of a GAN algorithm.

f. Face recognition:

Face recognition (FR) allows us to exactly identify or tag an image of a person. Day-to day applications include searching for celebrities on the web and auto-tagging friends and family in images. Face recognition is a form of fine-grained classification.
Face Recognition system is categorized in two modes:
Face identification involves one-to-many matches that compare a query face image against all the template images in the database to determine the identity of the query face. Another face recognition scenario involves a watchlist check by city authorities, where a query face is matched to a list of suspects (one-to-few matches).

Face verification involves a one-to-one match that compares a query face image against a template face image whose identity is being claimed (figure below)

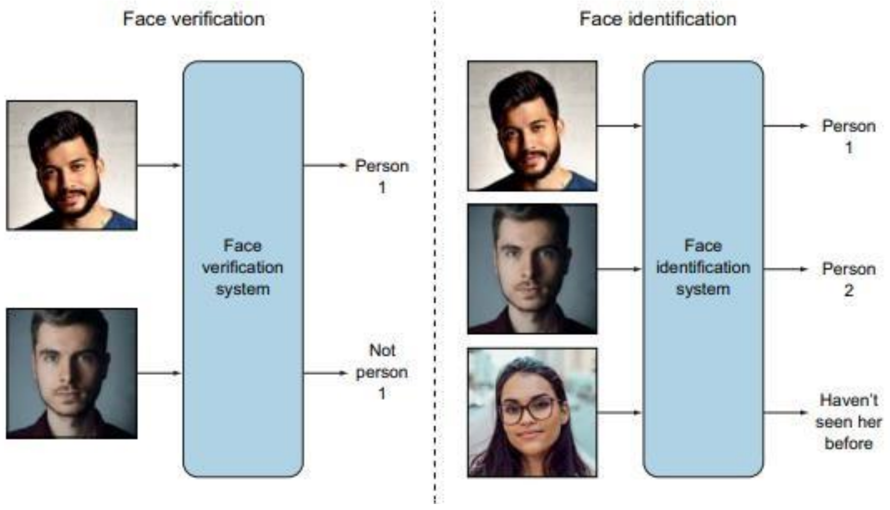


Figure: Example of face verification (left) and face recognition (right)

g. Image recommendation system:

In this task, a user seeks to find similar images with respect to a given query image. Shopping websites provide product suggestions (via images) based on the selection of a particular product, for example, showing a variety of shoes similar to those the user selected. An example of an apparel search is shown in figure below.

Figure: Apparel search.

Computer Vision Pipeline Process

The vision systems are composed of two main components,

- Sensing devices capture the image or signals through camera or sensors.
- The interpreting device is used to process and understand images.

The interpreting process uses "computer vision pipeline" that consists of four steps as shown in **Fig.5.a**

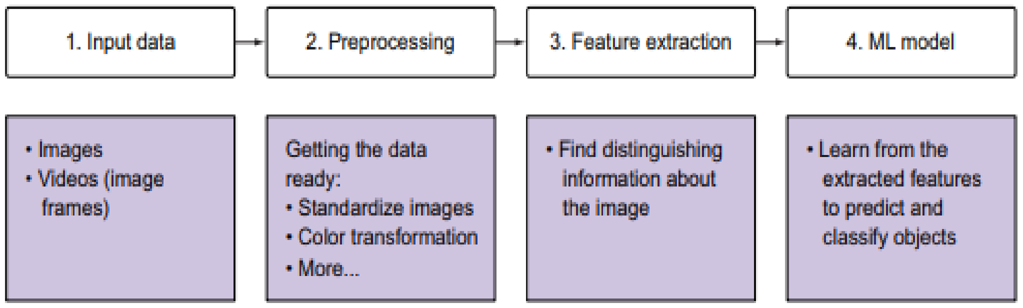


Fig.5.a; The computer vision pipeline

Computer Vision pipeline for image classification:

An image classifier is an algorithm that takes in an image as input and predicts a label or "class" that identifies that image. A class (also called a category) in machine learning is the output category of your data.

Example For a motorcycle image, if we want the model to predict the probability of the object from the following classes: motorcycle, car, and dog as shown in **Fig.5.b.**

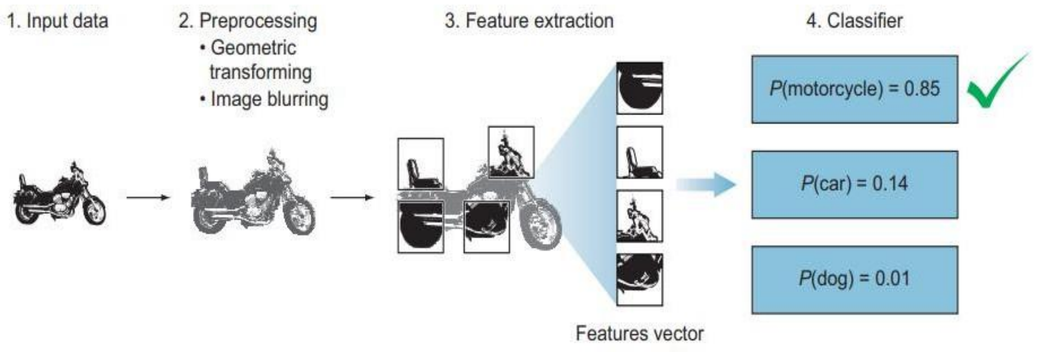


Fig.5.b. Predict the probability of the motorcycle object from the motorcycle, car, and dog classes

h. Input Data (Image):

For Computer Vision applications, the input data is either images or video data.

- The images are of two types, Grayscale (BW) and colour images (RGB) and videos are stacked sequential frames of images.
- An ***image*** can be represented as a function of two variables x and y, which define a two-dimensional area.
- A *digital image* is made of a grid of pixels.
- The pixel is the smallest unit of an image.
- Every image consists of a set of pixels in which their values represent the intensity of light that appears in a given place in the image.
- The image coordinate system is similar to the Cartesian coordinate system:

i) Grayscale Image:

In gray scale images are two-dimensional and lie on the x-y plane. The gray scale image has pixel values range from 0 to 255. Since the pixel value represents the intensity of light, the value 0 represents very dark pixels (black), 255 is very bright (white), and the values in between represent the intensity on the grayscale. **(Fig.5.c)**
Example: motorcycle and the grid of pixel

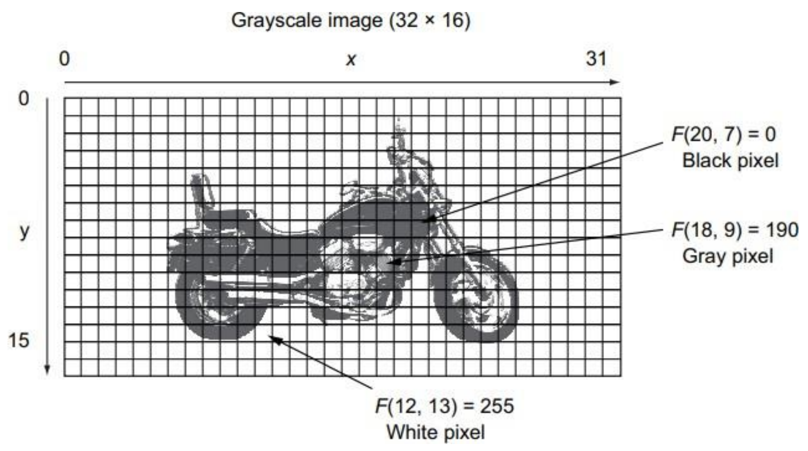


Fig 5.c Images consists of raw building blocks called pixels.

Grayscale => F(x, y) gives the intensity at position (x, y).
The pixel located at x = 12 and y = 13 is white; F(12, 13) = 255.

Mathematically an image is a function of F(x, y) and transforms it to a new image function G(x, y). The image transformation examples:

Application	Transformation
Darken the image.	`G(x, y) = 0.5 * F(x, y)`
Brighten the image.	`G(x, y) = 2 * F(x, y)`
Move an object down 150 pixels.	`G(x, y) = F(x, y + 150)`
Remove the gray in an image to transform the image into black and white.	`G(x, y) = { 0 if F(x, y) < 130, 255 otherwise }`

How computers see images:

When we look at an image, we see objects, landscape, colors, and so on. But that's not the case with computers. Consider figure 1.16. Your human brain can process it and immediately know that it is a picture of a motorcycle. To a computer, the image looks like a 2D matrix of the pixels' values, which represent intensities across the color spectrum. There is no context here, just a massive pile of data.

The image in figure below is of size 24 × 24. This size indicates the width and height of the image: there are 24 pixels horizontally and 24 vertically. That means there is a total of 576 (24 × 24) pixels. If the image is 700 × 500, then the dimensionality of the matrix will be (700, 500), where each pixel in the matrix represents the intensity of brightness in that pixel. Zero represents black, and 255 represents white.

What we see

What computers see

08	02	22	97	38	15	00	40	00	75	04	05	07	78	52	12	50	77	91	08
49	49	99	40	17	81	18	57	60	87	17	40	98	43	69	46	04	56	62	00
81	49	31	73	55	79	14	29	93	71	40	67	53	99	30	03	49	13	36	65
52	90	95	23	04	60	11	42	69	24	48	56	01	32	54	71	37	02	34	91
22	31	14	71	51	67	43	59	41	92	34	54	22	40	40	28	44	33	13	80
24	47	32	60	99	03	45	02	44	75	33	53	78	36	64	20	35	09	12	80
32	98	81	28	64	23	67	10	26	38	40	67	59	54	70	66	18	38	64	70
47	24	20	68	02	62	12	20	95	63	94	39	63	04	49	91	44	49	94	21
24	55	58	05	66	73	99	26	97	17	78	78	94	83	14	88	34	89	63	72
21	36	23	09	75	00	74	44	20	45	35	14	00	41	33	97	34	31	33	95
78	17	53	28	22	75	31	67	15	94	03	80	04	42	16	14	09	53	56	92
16	39	05	42	96	35	31	47	55	58	88	24	00	17	54	24	34	29	85	57
84	56	00	48	35	71	89	07	05	44	44	37	44	40	21	58	51	54	17	58
19	80	61	68	05	94	47	49	28	73	92	13	86	52	17	77	04	09	55	40
04	52	08	83	97	35	99	14	07	97	57	32	16	26	26	79	33	27	98	44
04	36	68	81	57	62	20	72	03	16	33	67	46	55	12	32	43	93	53	69
04	42	14	73	38	25	39	11	24	94	72	18	06	46	29	32	40	62	74	36
20	49	34	41	72	30	23	88	34	62	99	69	82	47	59	85	74	04	34	24
20	23	35	29	78	31	90	01	74	31	49	71	48	86	81	14	23	57	05	54
01	70	54	71	83	51	54	49	16	92	33	48	61	43	51	01	89	19	67	48

Figure 1.16 A computer sees images as matrices of values. The values represent the intensity of the pixels across the color spectrum. For example, grayscale images range between pixel values of 0 for black and 255 for white.

ii) Color images:

In colour images, the value of the pixel is represented by three numbers: the intensity of red, intensity of green, and intensity of blue In an RGB system, **Color image in RGB => F(x, y) = [red (x, y), green (x, y), blue (x, y)].** Color image is having three matrices stacked on top of each other; that's why it's a 3D matrix. The dimensionality of 700 × 700 color images is (700, 700, 3). The first matrix represents the red channel; then each element of that matrix represents an intensity of red color in that pixel, and likewise with green and blue. **(Fig 5.d)**

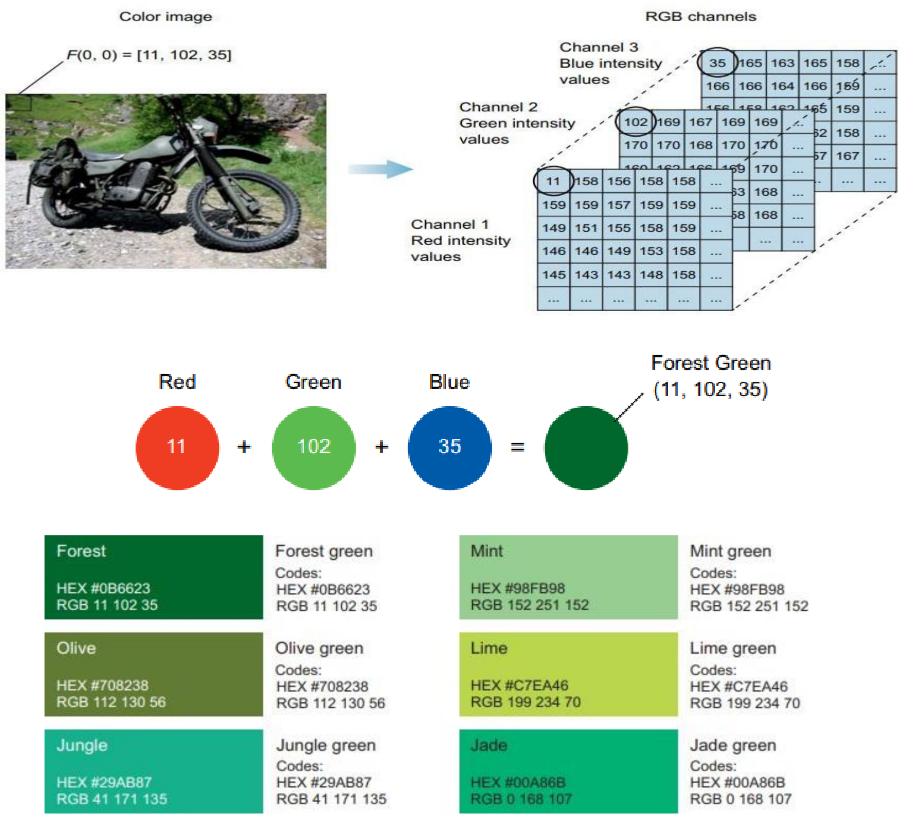

Color image
F(0, 0) = [11, 102, 35]
RGB channels
Channel 3 Blue intensity values
Channel 2 Green intensity values
Channel 1 Red intensity values
Red
Green
Blue
11
102
35
Forest Green (11, 102, 35)
Forest HEX #0B6623 RGB 11 102 35
Forest green Codes: HEX #0B6623 RGB 11 102 35
Mint HEX #98FB98 RGB 152 251 152
Mint green Codes: HEX #98FB98 RGB 152 251 152
Olive HEX #708238 RGB 112 130 56
Olive green Codes: HEX #708238 RGB 112 130 56
Lime HEX #C7EA46 RGB 199 234 70
Lime green Codes: HEX #C7EA46 RGB 199 234 70
Jungle HEX #29AB87 RGB 41 171 135
Jungle green Codes: HEX #29AB87 RGB 41 171 135
Jade HEX #00A86B RGB 0 168 107
Jade green Codes: HEX #00A86B RGB 0 168 107

Fig .5.d. Color images and representation

i. Image Pre-processing:

Image data pre-processing converts image data into a form that allows machine learning algorithms to solve it. It is often used to increase a model's accuracy, as well as reduce its complexity. The techniques used to pre-process image data are image resizing, converting images to grayscale, and image augmentation etc., when the images exist in different formats, i.e., natural, fake, grayscale, etc., we need to standardize them before feeding them into a neural network.

The important steps in image pre-processing techniques are

- Grayscale conversion
- Normalization
- Data Augmentation
- Image standardization

Image Pre-processing:

Image data pre-processing converts image data into a form that allows machine learning algorithms to solve it. It is often used to increase a model's accuracy, as well as reduce its complexity. The techniques used to pre-process image data are image resizing, converting images to grayscale, and image augmentation etc., when the images exist in different formats, i.e., natural, fake, grayscale, etc., we need to standardize them before feeding them into a neural network.

The important steps in image pre-processing techniques are

- Grayscale conversion
- Normalization
- Data Augmentation
- Image standardization

i) Grayscale conversion simply converting images from colored to black and white as shown in **Fig.5.f.** It is normally used to reduced number of pixels that need to be processed and reduce computation complexity in machine learning algorithms.

This could be a not good approach for applications depend on color information because it losses information in conversion. Since most pictures don’t need color to be recognized, it is wise to use grayscale, which reduces the number of pixels in an image, thus, reducing the computations required.

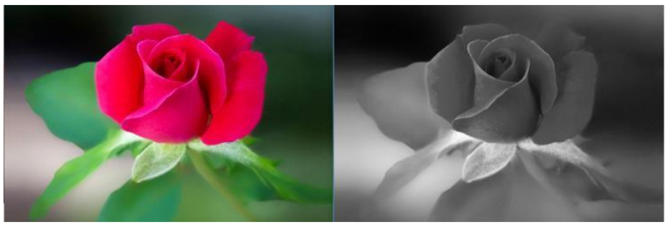


Fig.5.f. color image transformation to gray scale

Converting images to grayscale might not always be practical in solving some problems.

For examples where it would be impractical to use grayscale include: *trafficlights*, *healthcare diagnosis*, *autonomous vehicles*, *agriculture*, etc. The best way to know whether to use it or not depends on your human visual ability to identify an object without color.

ii) Normalization

Image normalization is a typical process in image processing that changes the range of pixel intensity values.
For example, when we perform a function that produces a normalization of an input image (grayscale or RGB). Then, we understand a representation of the range of values of the scale of the image represented between 0 and 255. i.e., very dark images become clearer.
Linear normalization of a digital image adjusts the pixel intensity values to a common scale, typically to improve contrast or prepare the image for further processing. The formula for linear normalization is often expressed as:

$$I' = \frac{I - I_{\min}}{I_{\max} - I_{\min}} \times (new_{\max} - new_{\min}) + new_{\min}$$

Where:

- I is the original pixel intensity.
- $I_{\min}$ and $I_{\max}$ are the minimum and maximum pixel intensity values in the original image, respectively.
- $new_{\min}$ and $new_{\max}$ are the desired minimum and maximum intensity values in the normalized image.
- I' is the normalized pixel intensity.

Linear normalization used in digital image processing to rescale pixel intensity values is given by the formula:

$$\text{Output\_channel} = 255 \times \frac{\text{Input\_channel} - \min}{\max - \min}$$

- **Output_channel**: The normalized pixel intensity in the output image, scaled to a range of 0 to 255, which is common for 8-bit images.
- **Input_channel**: The original pixel intensity value in the input image.
- **min**: The minimum intensity value in the input image, which could be based on the entire image or a specific channel.
- **max**: The maximum intensity value in the input image.

This formula scales the input pixel values such that the minimum value becomes 0 and the maximum value becomes 255, enhancing contrast and detail visibility in the resulting image.

For grayscale image, normalize using one channel and color images normalize a RGB (3 channels)

Examples:

The left image depicts the original image is too dark and results clear after the normalization process.

Example 2: The left image depicts the right-side original image is very bright results with better contrast after the normalization process.

iii) Data augmentation:

Data augmentation helps in preventing a neural network from learning irrelevant features. This results in better model performance.

Data augmentation is the process of making minor alterations to existing data to increase its diversity without collecting new data. It is a technique used for enlarging a dataset.

There are two types of augmentation are *Offline augmentation* is used for small datasets. *Online augmentation* is used for large datasets. It is normally applied in real-time.

Standard data augmentation techniques include horizontal & vertical flipping, rotation, cropping, shearing, etc. as shown in **Fig.5.h**

Shifting is the process of shifting image pixels horizontally or vertically.

Flipping reverses, the rows or

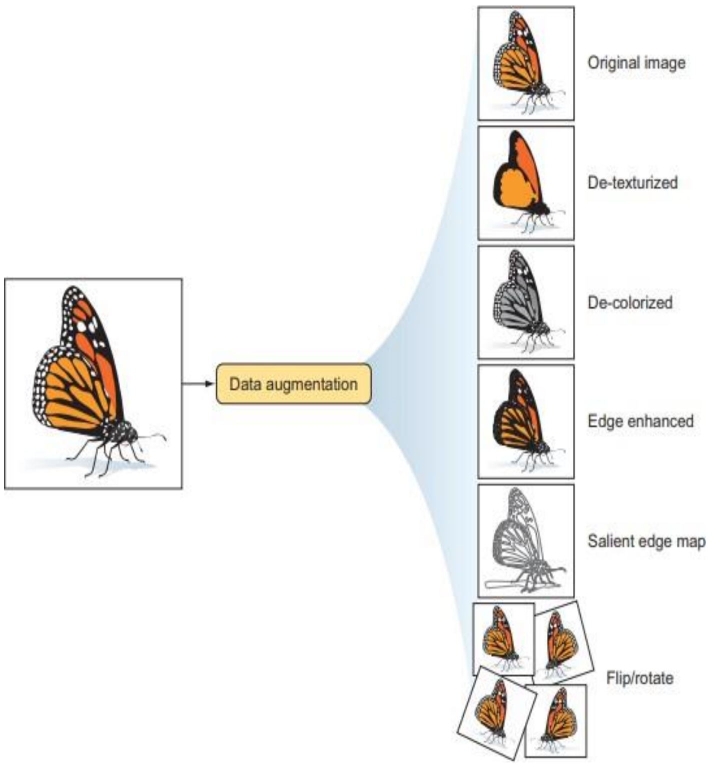
columns of pixels in either vertical or horizontal cases, respectively.

Rotation process involves rotating an image by a specified degree.

Changing brightness is the process of increasing or decreasing image contrast.

Cropping is the process of creating a random subset of an original image which is then resized to the size of the original image.

Scaling image can be scaled either inward or outward. When scaling an image outward, the image becomes more significant than the original and vise versa.



iv) Standardizing the images:

Standardization is a method that scales and pre-processes images to have similar heights and widths. It re-scales data to have a standard deviation of 1 (unit variance) and a mean of 0. Standardization helps to improve the quality and consistency of data. one important constraint that exists in some ML algorithms, such as CNNs, is the need to resize the images in your dataset to unified dimensions. This implies that your images must be pre-processed and scaled to have identical widths and heights before being fed to the learning algorithm

C. Feature extraction

Features are parts or patterns of an object in an image that help to identify image. The entire DL model works around the idea of extracting useful features that clearly define the objects in the image.
A raw data (image) is transformed into a feature vector using learning algorithm, which can learn the characteristics of the object.

For example — a square has 4 corners and 4 edges, they can be called features of the square, and they help us humans identify it’s a square. Features include properties like corners, edges, regions of interest points, ridges, etc.

Example: when we feed the raw input image of a motorcycle into a feature extraction algorithm. the extraction algorithm produces a vector that contains a list of features as shown in figure below. This feature vector is a 1D array that makes a robust representation of the object**.**

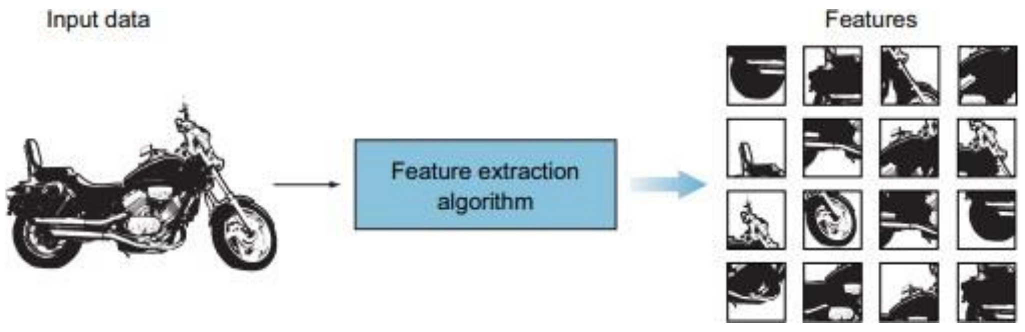


Fig.5. j. Feature Extraction

Feature extraction for Traditional Machine Learning:

The Process relies on domain knowledge (or partner with domain experts) to extract features that make ML algorithms work better. Feeding the produced features to a classifier like a support vector machine (SVM) or AdaBoost to predict the output (**Fig 5.l**).

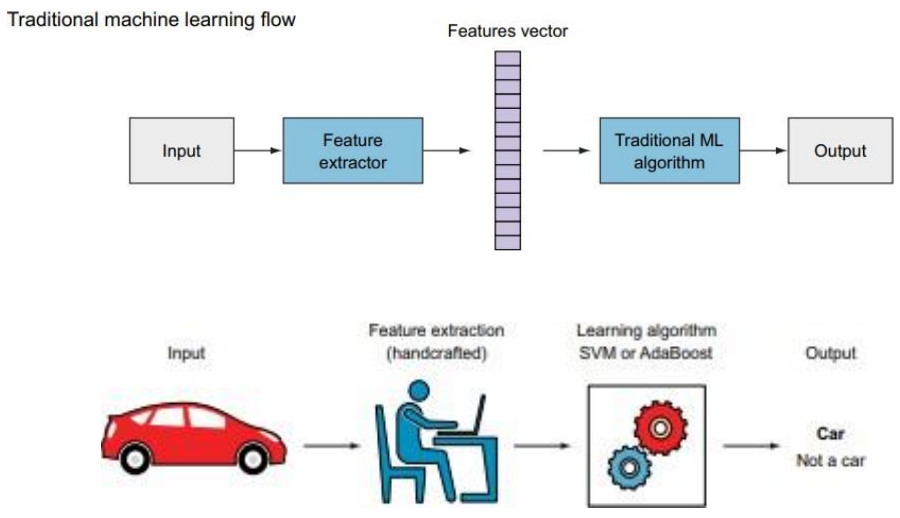
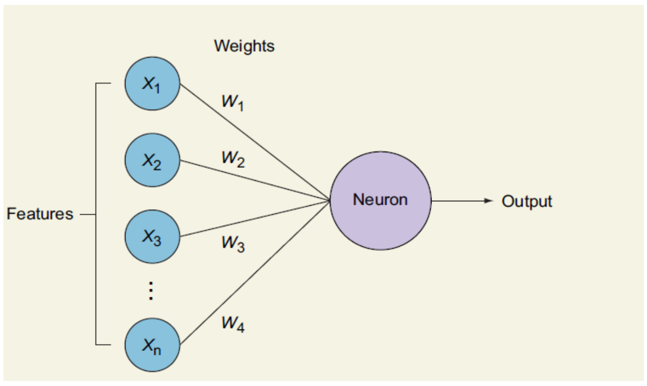

Traditional machine learning flow
Features vector
Input
Feature extractor
Traditional ML algorithm
Output
Input
Feature extraction (handcrafted)
Learning algorithm SVM or AdaBoost
Output
Car
Not a car

Fig 5.l: ***Traditional machine learning algorithms require handcrafted feature extraction.***

Feature extraction in Deep Learning:

In deep Learning, no need to manually extract features from the image. The network extract features automatically and learns their importance on the output by applying weights to its connections.



When a raw image is fed to the network, while passing through the network layers, identifies features/ patterns within the image as shown in figure (**Fig 5.m**). Neural networks can be thought of as feature extractors plus classifiers that are end-to-end trainable, as opposed to traditional ML models that use handcrafted features.

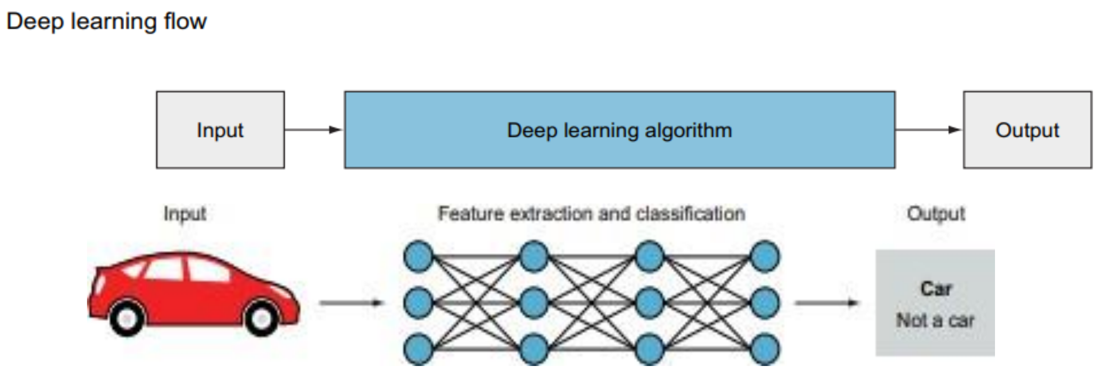


Fig.5.m : A DNN the input image through its layers to automatically extract features

No Free Lunch Theorem:

The No Free Lunch Theorem is often thrown around in the field of optimization and machine learning, often with little understanding of what it means or implies.

The theorem states that all optimization algorithms perform equally well when their performance is averaged across all possible problems.

It implies that there is no single best optimization algorithm. Because of the close relationship between optimization, search, and machine learning, it also implies that there is no single best machine learning algorithm for predictive modeling problems such as classification and regression.

- The no free lunch theorem suggests the performance of all optimization algorithms are identical, under some specific constraints.
- There is probably no single best optimization algorithm or machine learning algorithm.
- The practical implications of the theorem may be limited given we are interested in a small subset of all possible objective functions.

Classification model:

The features vectors are fed into a classification model predicts the class of the image.

- First you see a wheel feature; could this be a car, a motorcycle, or a dog? Clearly it is not a dog, because dogs don't have wheels (at least, normal dogs, not robots). Then this could be an image of a car or a motorcycle.
- You move on to the next feature, the headlights. There is a higher probability that this is a motorcycle than a car.
- The next feature is rear mudguards—again, there is a higher probability that it is a motorcycle.
- The object has only two wheels; this is closer to a motorcycle.
- And you keep going through all the features like the body shape, pedal, and so on, until you arrive at a best guess of the object in the image.

The output of this process is the probability of each class., the dog has the lowest probability, 1%, whereas there is an 85% probability that this is a motorcycle. Although the model was able to predict the right class with the highest probability, it is still a little having confusion between cars and motorcycles because it was predicted as 14% chance of car. Since it is a motorcycle, we can say that our ML classification algorithm is 85% accurate.

The different approaches used to improve the accuracy of our model follow either step:

- acquire more training images, or
- more processing to remove noise, or
- extract better features, or
- change the classifier algorithm and tune some hyperparameters, or
- even allow more training time.

5. Classifier learning algorithm:

The classification task is done by two ways:

- By traditional ML algorithms like SVMs which might give result for some problems.

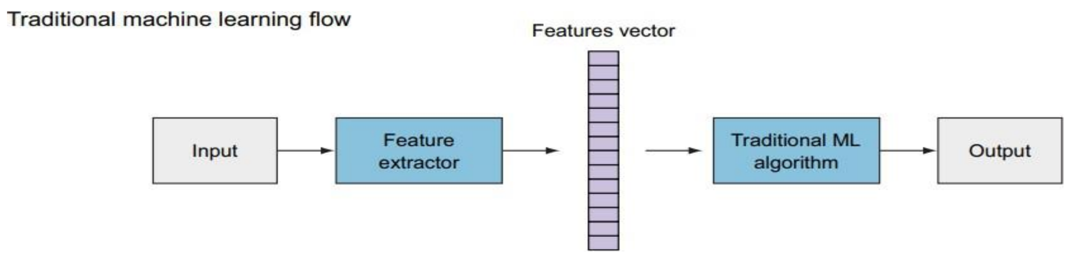


Fig : Traditional Classifier learning algorithm

- Deep neural network algorithms like CNNs give best results to classifying images for most complex problems.

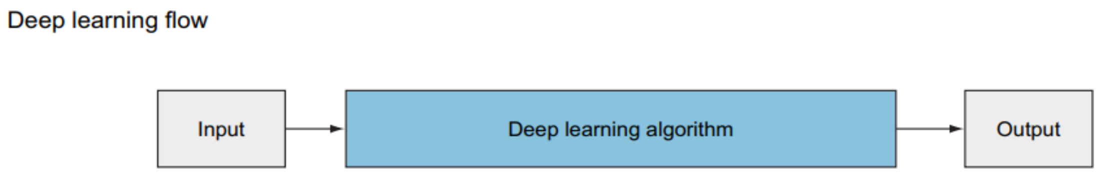


Fig : Deep NN Classifier learning algorithm

Deep Learning Classifier:

The neural networks classifier automatically extracts useful features from your dataset(images), to predicts the class labels for new images.

The input images are passed through the hidden layers of the deep neural **network** to learn their features layer by layer. The deeper network with more hidden layers learns the features of the dataset will prone the model to overfitting and reduces the accuracy of the model. The last layer of the neural network is fully connected usually acts as the classifier that outputs the class label.

To improve the accuracy of the model, different approaches are used:

- acquire more training images, or
- more processing to remove noise, or
- extract better features, or
- change the classifier algorithm and tune some hyperparameters, or
- even allow more training time.

Deeplearning Flow:

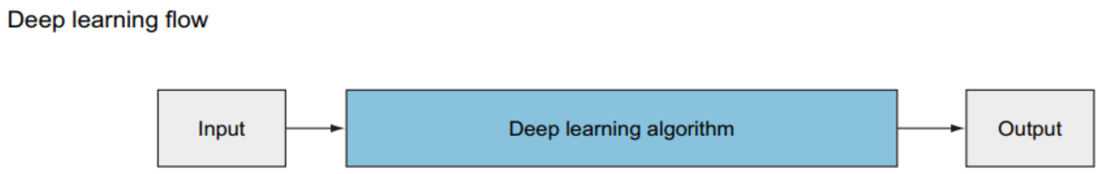


Fig : Deep NN Classifier learning algorithm

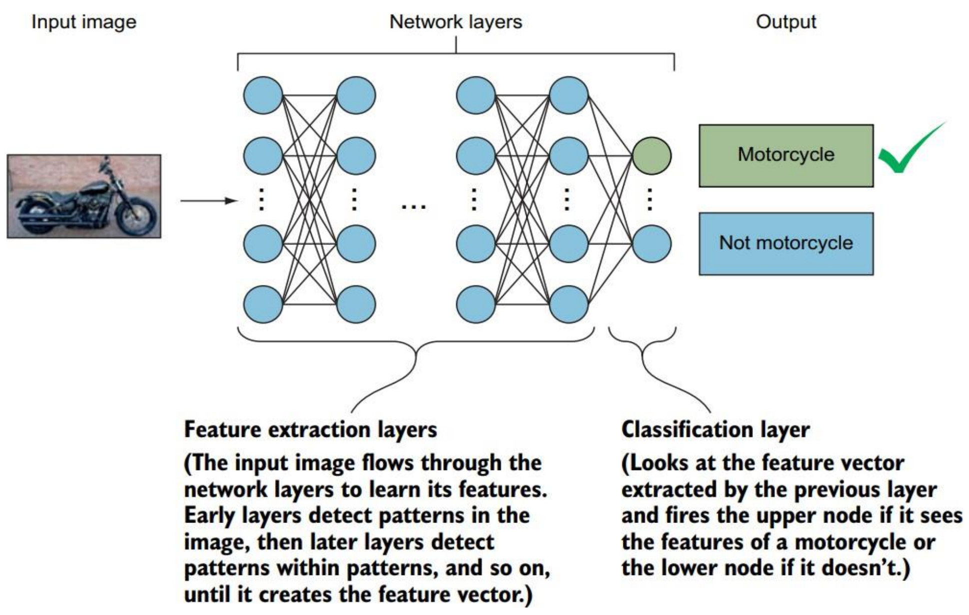

Input image
Network layers
Output
Motorcycle
Not motorcycle
Feature extraction layers
(The input image flows through the network layers to learn its features. Early layers detect patterns in the image, then later layers detect patterns within patterns, and so on, until it creates the feature vector.)
Classification layer
(Looks at the feature vector extracted by the previous layer and fires the upper node if it sees the features of a motorcycle or the lower node if it doesn't.)

Fig : Deep NN Classifier learning algorithm

PART-B

Deep learning and neural networks:

Understanding perceptrons, Multilayer perceptrons, Activation functions, The feed forward process, Optimization algorithms, Back propagation.

1. Understanding perceptrons:

Let's take a look at the artificial neural network (ANN) diagram:

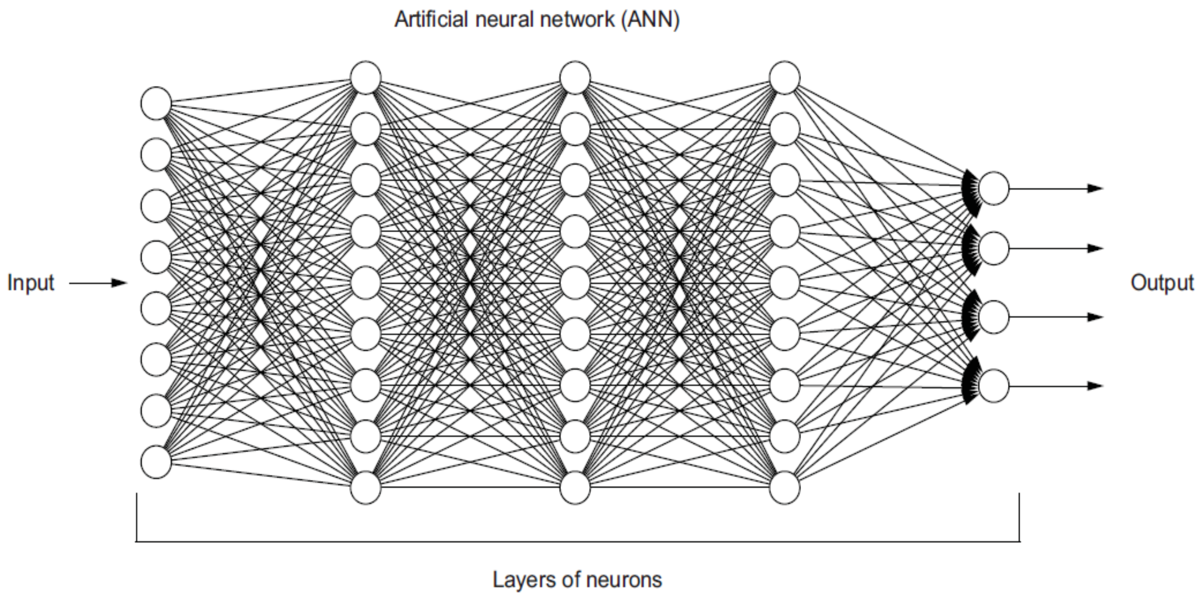
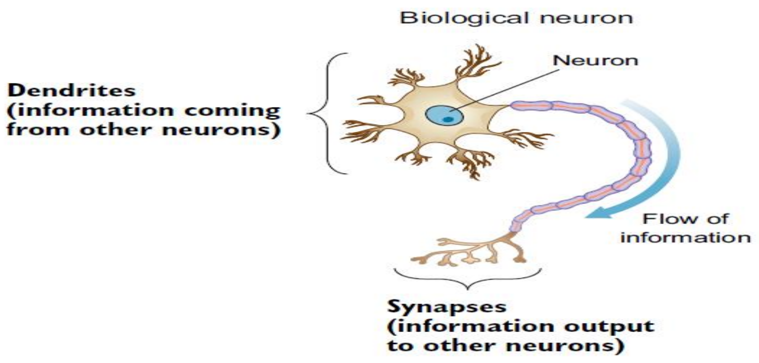


Figure 2.2 An artificial neural network consists of layers of nodes, or neurons connected with edges.

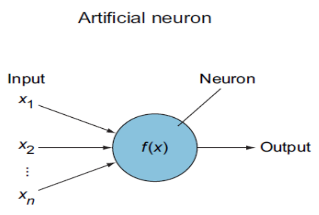
You can see that ANNs consist of many neurons that are structured in layers to perform some kind of calculations and predict an output. This architecture can be also called a *multilayer perceptron*, which is more intuitive because it implies that the network consists of perceptrons structured in multiple layers.

What is a perceptron?

The simplest neural network is the perceptron, which consists of a single neuron. Conceptually, the perceptron functions in a manner similar to a biological neuron as shown below.


Biological neuron
Neuron
Dendrites (information coming from other neurons)
Flow of information
Synapses (information output to other neurons)

A biological neuron receives electrical signals from its *dendrites*, modulates the electrical signals in various amounts, and then fires an output signal through its *synapses* only when the total strength of the input signals exceeds a certain threshold. The output is then fed to another neuron, and so forth. To model the biological neuron phenomenon, the artificial neuron performs two consecutive functions: it calculates the *weighted sum* of the inputs to represent the total strength of the input signals, and it applies a *step function* to the result to determine whether to fire the output 1 if the signal exceeds a certain threshold or 0 if the signal doesn't exceed the threshold.



Not all input features are equally useful or important. To represent that, each input node is assigned a weight value, called its *connection weight*, to reflect its importance.

Not all input features are equally important (or useful) features. Each input feature ($x1$) is assigned its own weight ($w1$) that reflects its importance in the decision-making process. Inputs assigned greater weight have a greater effect on the output. If the weight is high, it amplifies the input signal; and if the weight is low, it diminishes the input signal. In common representations of neural networks, the weights are represented by lines or edges from the input node to the perceptron. For example, if you are predicting a house price based on a set of features like size, neighborhood, and number of rooms, there are three input features ($x1$, $x2$, and $x3$). Each of these inputs will have a different weight value that represents its effect on the final decision. For example, if the size of the house has double the effect on the price compared with the neighborhood, and the neighborhood has double the effect compared with the number of rooms, you will see weights something like 8, 4, and 2, respectively.

Perceptron is a supervised deep learning model used for binary classification tasks. Performs two consecutive functions: it calculates the weighted sum of the inputs to represent the total strength of the input signals, and it applies a step function to the result to determine whether to fire the output 1 if the signal exceeds a certain threshold or 0 if the signal doesn't exceed the threshold.

a. Perceptron architecture: The perceptron is as shown in Fig. 6.a.

Mr. Frank Rosenblatt invented the perceptron model as a binary classifier which contains three main components. These are as follows:

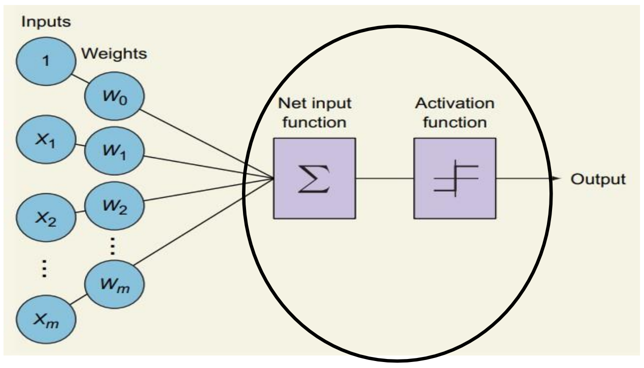


Fig. 6.a. Single Layer Perceptron

The Perceptron consists of:

Input vector denoted by uppercase X (x1, x2, . . ., xn) fed to the neuron .

bias (b) is an extra weight used while learning and adjusting the neuron to minimize the cost function

Weights vector—Each x1 is assigned a weight value w1 that represents its importance to distinguish between different input data points.

Neuron functions—The calculations performed within the neuron to modulate the input signals: the weighted sum and step activation function.

Output—controlled by the type of activation function you choose for your network. There are different activation functions eg. a step function, the output is either 0 or 1. Other activation functions produce probability output or float numbers. The output node represents the perceptron prediction.

b. When to use Perceptron and How do they learn?

The single perceptron works with *linearly separable data*. This means the training data should be separated by a straight line as shown in **Fig 6.b**

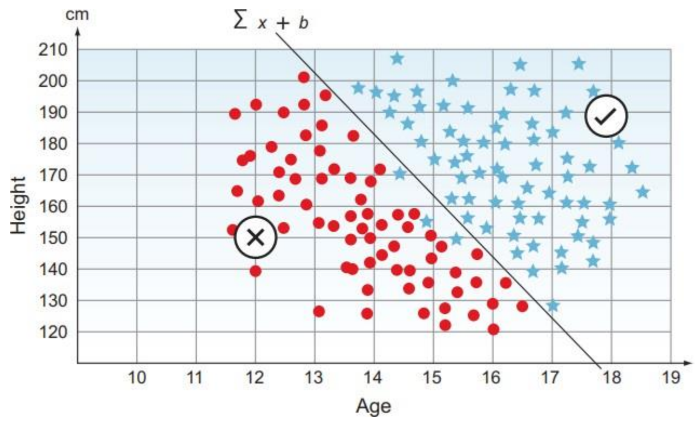


Fig 6.b. plot for best fit between Age and height

How does the perceptron learn?

The perceptron uses trial and error to learn from its mistakes. It uses the weights as knobs by tuning their values up and down until the network is trained. That is the Weights are tuned up and down during the learning process to optimize the value of the loss function.

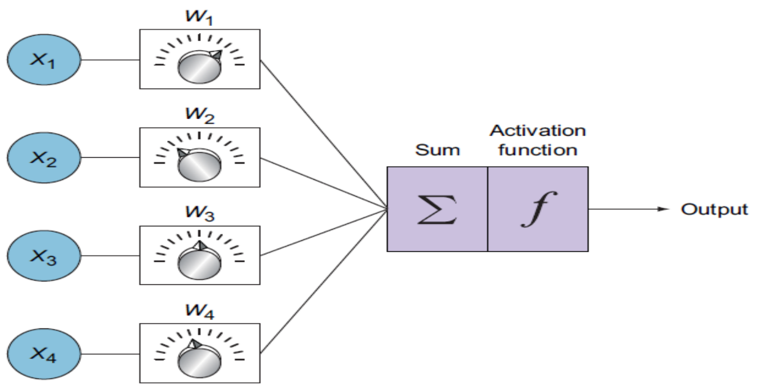


Figure 2.6 Weights are tuned up and down during the learning process to optimize the value of the loss function.

WEIGHTED SUM FUNCTION

Also known as a *linear combination*, the weighted sum function is the sum of all inputs multiplied by their weights, and then added to a bias term. This function produces a straight line represented in the following equation:

$$z = \sum x_i \cdot w_i + b \text{ (bias)}$$

$$z = x_1 \cdot w_1 + x_2 \cdot w_2 + x_3 \cdot w_3 + \cdots + x_n \cdot w_n + b$$

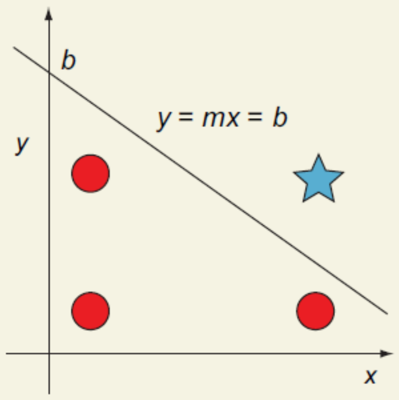
Here is how we implement the weighted sum in Python:

```
z = np.dot(w.T,X) + b
```

X is the input vector (uppercase X), w is the weights vector, and b is the y-intercept.

What is a bias in the perceptron, and why do we add it?

Let's brush up our memory on some linear algebra concepts to help understand what's happening under the hood. Here is the function of the straight line:



The equation of a straight line

The function of a straight line is represented by the equation ($y = mx + b$), where b is the y-intercept. To be able to define a line, you need two things: the slope of the line and a point on the line. The bias is that point on the y-axis. Bias allows you to move the line up and down on the y-axis to better fit the prediction with the data. Without the bias (b), the line always has to go through the origin point (0,0), and you will get a poorer fit. To visualize the importance of bias, look at the graph in the above figure and try to separate the circles from the star using a line that passes through the origin (0,0). It is not possible.

Steps of perceptron learning

1 The neuron calculates the weighted sum and applies the activation function to make a prediction yˆ . This is called the feed forward process.

$$z = x_1 \cdot w_1 + x_2 \cdot w_2 + x_3 \cdot w_3 + \cdots \ + x_n \cdot w_n + b$$

$$z = \sum x_i \cdot w_i + b \text{ (bias)}$$

$$\hat{y} = \text{activation}\left(\sum x_i \cdot w_i + b\right)$$

2 It compares the output prediction with the correct label to calculate the error

$$\text{error} = y - \hat{y}$$

3 It then updates the weight. If the prediction is too high, it adjusts the weight to make a lower prediction the next time, and vice versa.

4 Repeat!

This process is repeated many times, and the neuron continues to update the weights to improve its predictions until step 2 produces a very small error (close to zero),which means the neuron's prediction is very close to the correct value. At this point, we can stop the training and save the weight values that yielded the best results to apply to future cases where the outcome is unknown.

How to Train a perceptron model to predict whether a player will be accepted into the college squad?

The first step is to collect all the data from previous years and train the perceptron to predict whether players will be accepted based on only two features (height and weight). The trained perceptron will find the best weights and bias values to produce the straight line that best separates the accepted from non-accepted (best fit).

The line has this equation: z = height · w1 + age · w2 + b. After the training is complete on the training data, we can start using the perceptron to predict with new players. When we get a player who is 150 cm in height and 12 years old, we compute the previous equation with the values (150, 12)as shown in **Fig .6.c**

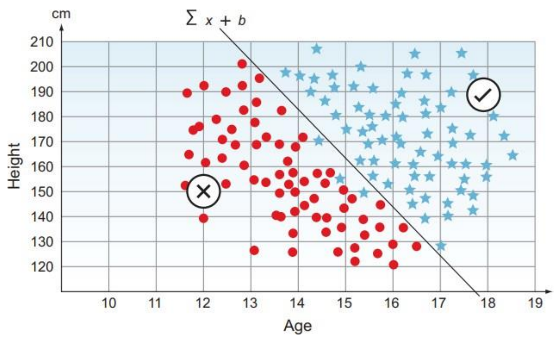


Fig 6.c plot for best fit between Age and height

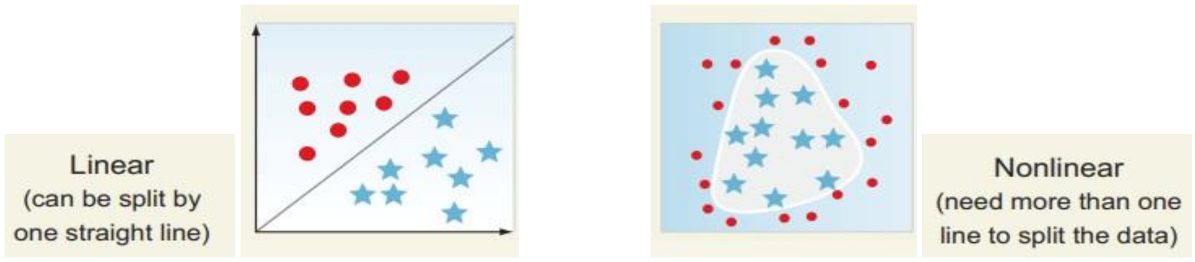
2. Multi layer Perceptrons:

The deep Learning model uses neural network to extract features automatically by passing the input through hidden layers. Based on the depth, number of hidden layers and I/O capabilities neural network models are classified into different types. They are

1. Perceptron
2. Feed Forward Networks
3. Multi-Layer Perceptron
4. Convolutional Neural Networks
5. AdvancedCNN- LNet,AlexNet,VGG..Etc.,

The single perceptron works fine because our data was linearly separable. This means the training data can be separated by a straight line. But in real time isn't always that simple.

What happens when we have a more complex dataset that cannot be separated by a straight line (nonlinear dataset fig 7.a.))? We use Multi layer Perceptrons .



Linear
(can be split by one straight line)
Nonlinear
(need more than one line to split the data)

Fig .7.a Linear Data Vs Non Linear data

Linear vs. nonlinear problems

- *Linear datasets*—The data can be split with a single straight line.
- *Nonlinear datasets*—The data cannot be split with a single straight line. We need more than one line to form a shape that splits the data.

i) Multi layer Perceptron Architecture:

A very common neural network architecture is to stack the neurons in layers on top of each other, called hidden layers. Each layer has n number of neurons. Layers are connected to each other by weight connections. This leads to the multilayer perceptron (MLP) architecture in the **Fig 7.b.**

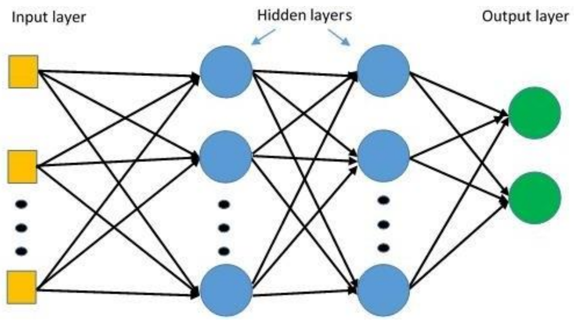


Fig 7.b. Multi Layer Perceptron(ANN)

The main components of the neural network architecture are

Input Layer

It is the initial or starting layer of the Multilayer perceptron. It takes input from the training data set and forwards it to the hidden layer. There are n input nodes in the input layer. The number of input nodes depends on the number of dataset features. Each input vector variable is distributed to each of the nodes of the hidden layer.

Hidden Layer

It is the heart of all Artificial neural networks. This layer comprises all computations of the neural network. The edges of the hidden layer have weights multiplied by the node values. This layer uses the activation function. There can be one or two hidden layers in the model. Several hidden layer nodes should be accurate as few nodes in the hidden layer make the model unable to work efficiently with complex data. More nodes will result in an overfitting problem.

Output Layer

This layer gives the estimated output of the Neural Network as shown in fig 7.d. The number of nodes in the output layer depends on the type of problem. For a single targeted variable, use one node. N classification problem, ANN uses N nodes in the output layer.

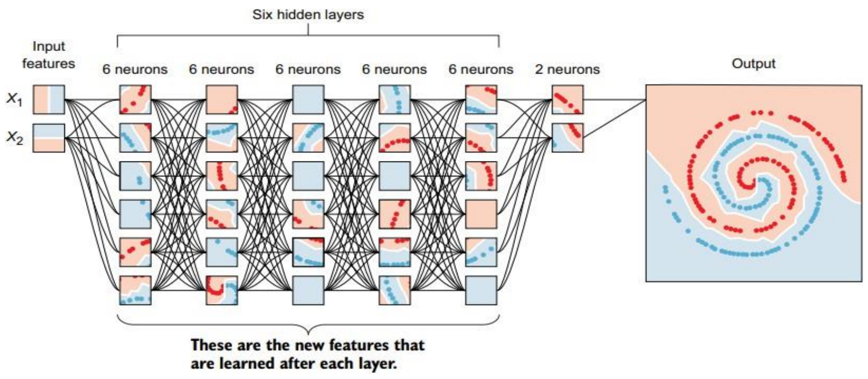
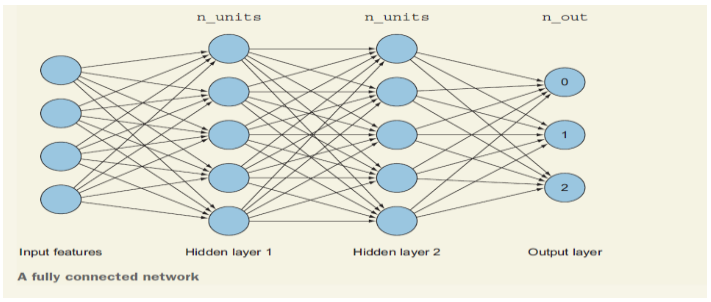


Fig.7.d. ANN with estimated output

How does a multilayer perceptron work?

1. The input node represents the feature of the dataset.
2. Each input node passes the vector input value to the hidden layer.
3. In the hidden layer, each edge has some weight multiplied by the input variable. All the production values from the hidden nodes are summed together. To generate the output
4. The activation function is used in the hidden layer to identify the active nodes.
5. The output is passed to the output layer.
6. Calculate the difference between predicted and actual output at the output layer.
7. The model uses backpropagation after calculating the predicted output.

Fully connected layers:
It is important to call out that the layers in classical MLP network architectures are fully connected to the next hidden layer. In the following figure, notice that each node in a layer is connected to all nodes in the previous layer. This is called a *fully connected network*.



ii) Some Hyper Parameters in a neural network are

Number of hidden layers—The general idea is that the more neurons you have, the better your network will learn the training data. But if you have too many neurons, this might lead to a phenomenon called overfitting: the network learned the training set so much that it memorized it instead of learning its features. Thus, it will fail to generalize. To get the appropriate number of layers, start with a small network, and observe the network performance. Then start adding layers until you get satisfying results.

Activation function—There are many types of activation functions, the most popular being ReLU and softmax. It is recommended that you use ReLU activation in the hidden layers and Softmax for the output layer.

Error function—Measures how far the network's prediction is from the true label. Mean square error is the metrics for regression problems, and cross-entropy is metric for classification problems.

Optimizer—Optimization algorithms are used to find the optimum weight values that minimize the error. There are several optimizers - batch gradient descent, stochastic gradient descent, and mini-batch gradient descent. Adam and RMSprop are two other popular optimizers.

Batch size—Mini-batch size is the number of sub-samples given to the network, after which parameter update happens. Bigger batch sizes learn faster but require more memory space. A good default for batch size might be 32. Also try 64, 128, 256, and so on.

Number of epochs—The number of times the entire training dataset is shown to the network while training. Increase the number of epochs until the validation accuracy starts decreasing even when training accuracy is increasing (overfitting).

Learning rate—One of the optimizer's input parameters that we tune is learning rate. If lr is too small is guaranteed to reach the minimum error (if you train for infinity time). A learning rate that is too big speeds up the learning but is not guaranteed to find the minimum error. The default lr value of the optimizer in most DL libraries is a reasonable start to get decent results.

2. Activation functions:

Activation functions are also referred to as *transfer functions* or *nonlinearities* because they transform the linear combination of a weighted sum into a nonlinear model. An activation function is placed at the end of each perceptron to decide whether to activate this neuron.

- The purpose of the activation function is to introduce nonlinearity into the network.
- Without it, a multilayer perceptron will perform similarly to a single perceptron no matter how many layers we add.
- Activation functions are needed to restrict the output value to a certain finite value.

A neuron in a neural network is a function that calculates the output of the neuron based on its inputs and the weights on individual inputs. Nontrivial problems can be solved only using a nonlinear activation function. Popular types of activation functions functionality is as shown in Fig 8.a

1. Binary Step Function
2. Linear Function
3. Sigmoid
4. Softmax
5. Tanh

ReLU

6. Leaky ReLU .....Etc.,

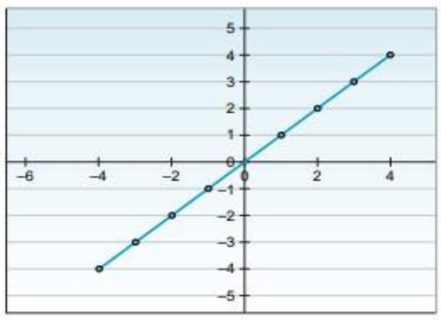
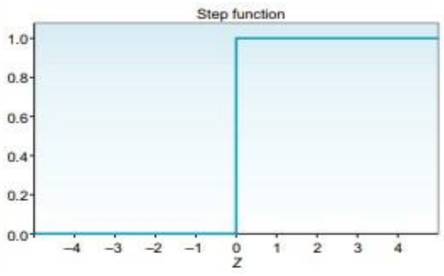
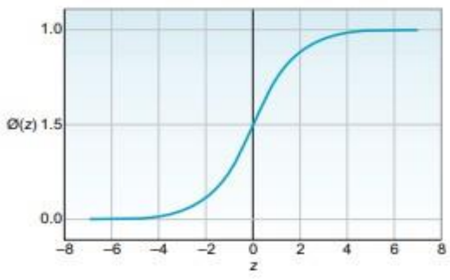
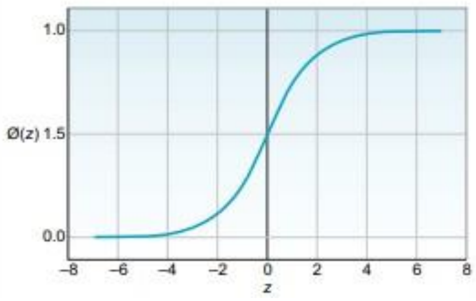
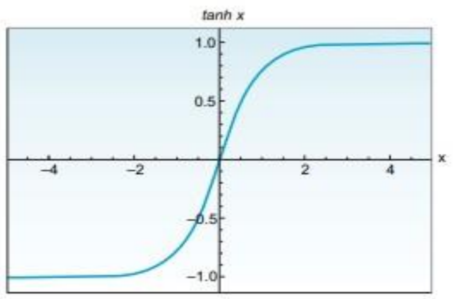
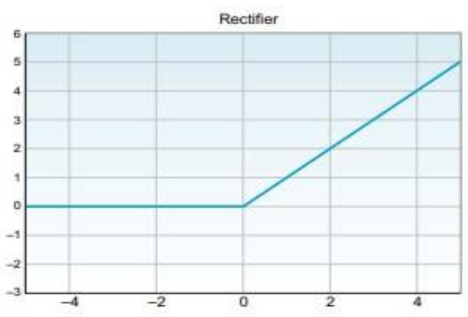
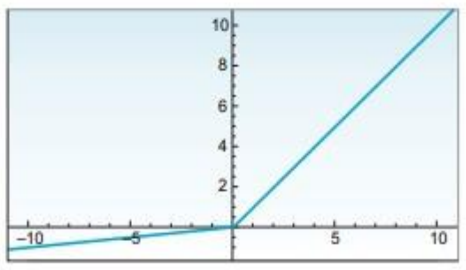
Activation function	Description	Plot	Equation
Linear transfer function (identity function)	The signal passes through it unchanged. It remains a linear function. Almost never used.		$f(x) = x$
Heaviside step function (binary classifier)	Produces a binary output of 0 or 1. Mainly used in binary classification to give a discrete value.		$\text{output} = \begin{cases} 0 \text{ if } w \cdot x + b \leq 0 \\ 1 \text{ if } w \cdot x + b > 0 \end{cases}$
Sigmoid/ logistic function	Squishes all the values to a probability between 0 and 1, which reduces extreme values or outliers in the data. Usually used to classify two classes.		$\sigma(z) = \frac{1}{1+e^{-z}}$
Softmax function	A generalization of the sigmoid function. Used to obtain classification probabilities when we have more than two classes.		$\sigma(x_j) = \frac{e^{x_j}}{\sum_i e^{x_i}}$
Hyperbolic tangent function (tanh)	Squishes all values to the range of –1 to 1. Tanh almost always works better than the sigmoid function in hidden layers.		$\tanh(x) = \frac{\sinh(x)}{\cosh(x)} = \frac{e^x - e^{-x}}{e^x + e^{-x}}$

Fig 8.a Popular types of activation Functions

Rectified linear unit (ReLU)	Activates a node only if the input is above zero. Always recommended for hidden layers. Better than tanh.		$f(x) = \max(0, x)$
Leaky ReLU	Instead of having the function be zero when $x < 0$, leaky ReLU introduces a small negative slope (around 0.01) when (x) is negative.		$f(x) = \max(0.01x, x)$

3. Training a MLP model

Training a neural network involves showing the network many examples (a training dataset); the network makes predictions through feedforward calculations and compares them with the correct labels to calculate the error. Finally, the neural network needs to adjust the weights (on all edges) until it gets the minimum error value, which means maximum accuracy. To get Max accuracy we need to do is build algorithms that find the optimum weights.

The Training neural network is done in three main steps:

- Feed forward Process
- Calculate Error,
- Optimize Weights

a. Feed forward Process:

The neural network learns by implementing the complete forward-pass calculations to produce a prediction output. The process of computing the linear combination and applying the activation function is called feed forward. The term feed forward is used to imply the forward direction in which the information flows from the input layer through the hidden layers, all the way to the output layer.

This process happens through the implementation of two consecutive functions:

i. the weighted sum and
ii. the activation function.

In short, the forward pass is the calculations through the layers to make a prediction.

Let's us consider a simple three-layer neural network as shown in figure Fig.9.a ,its components:

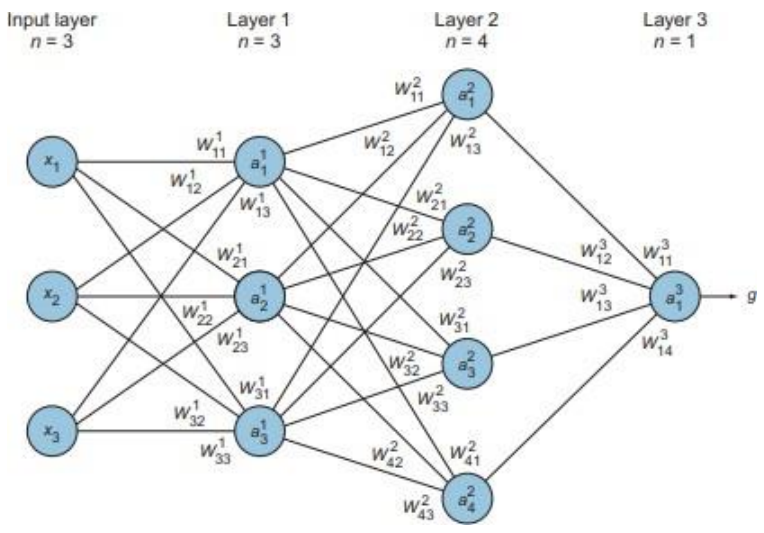


Fig9.A. Three-Layer Neural Network

Layers—This network consists of an input layer with three input features, and three hidden layers with 3, 4, 1 neurons in each layer

Weights and biases (w, b)—The edges between nodes are assigned random weights denoted as $W_{ab}^{(n)}$, where (n) indicates the layer number and (ab) indicates the weighted edge connecting the ath neuron in layer (n) to the b th neuron in the previous layer (n – 1). For example, $W_{23}^{(2)}$ is the weight that connects the second node in layer 2 to the third node in layer 1 (a_2^2 to a_1^3). (Note that you can see different denotations of $W_{ab}^{(n)}$ in other DL literature, which is fine as long as you follow one convention for your entire network.) The biases are treated similarly to weights because they are randomly initialized, and their values are learned during the training process. So, for convenience, from this point forward we are going to represent the basis with the same notation that we gave for the weights (w). In DL literature, you will mostly find all weights and biases represented as (w) for simplicity.

Activation functions σ(x)—In this example, we are using the sigmoid function σ(x) as an activation function.

Node values (a)—We will calculate the weighted sum, apply the activation function, and assign this value to the node a_m^n, where n is the layer number and m is the node index in the layer. For example, a_2^3 means node number 2 in layer 3.

Feed forward calculations:

Step1: start the feed forward calculations

$$a_1^{(1)} = \sigma\left(W_{11}^{(1)}x_1 + W_{12}^{(1)}x_2 + W_{13}^{(1)}x_3\right)$$
$$a_2^{(1)} = \sigma\left(W_{21}^{(1)}x_1 + W_{22}^{(1)}x_2 + W_{23}^{(1)}x_3\right)$$
$$a_3^{(1)} = \sigma\left(W_{31}^{(1)}x_1 + W_{32}^{(1)}x_2 + W_{33}^{(1)}x_3\right)$$

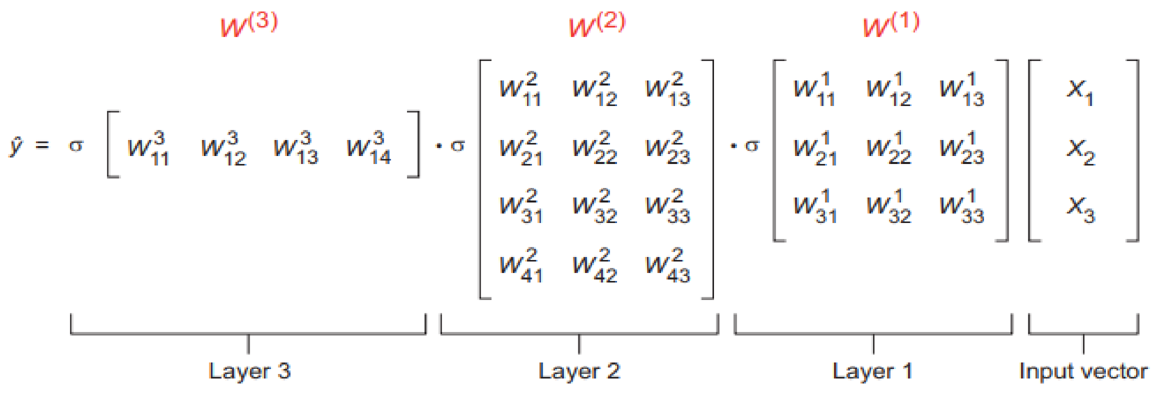
Step2: do the same calcul ations for layer 2

$$a_1^{(2)}, a_2^{(2)}, a_3^{(2)}, \text{ and } a_4^{(2)}$$

Step3: all the way to the output prediction in layer 3

$$\hat{Y} = a_1^{(3)} = \sigma\left(W_{11}^{(3)}a_1^2 + W_{12}^{(3)}a_2^2 + W_{13}^{(3)}a_3^2 + W_{14}^{(3)}a_4^2\right)$$

$$\hat{y} = \sigma \cdot W^{(3)} \cdot \sigma \cdot W^{(2)} \cdot \sigma \cdot W^{(1)} \cdot (x)$$


W(3)
W(2)
W(1)
ŷ = σ [w³₁₁ w³₁₂ w³₁₃ w³₁₄] · σ [w²₁₁ w²₁₂ w²₁₃; w²₂₁ w²₂₂ w²₂₃; w²₃₁ w²₃₂ w²₃₃; w²₄₁ w²₄₂ w²₄₃] · σ [w¹₁₁ w¹₁₂ w¹₁₃; w¹₂₁ w¹₂₂ w¹₂₃; w¹₃₁ w¹₃₂ w¹₃₃] [x₁; x₂; x₃]
Layer 3
Layer 2
Layer 1
Input vector

b. Error Function:

The error function is a measure of how “wrong” the neural network prediction is with respect to the expected output (the label). It quantifies how far we are from the correct solution. example, if we have a high loss, then our model is not doing a good job. The smaller the loss, the better the job the model is doing. The larger the loss, the more our model needs to be trained to increase its accuracy.
The error in simplest form, is calculated by comparing the prediction yˆ and the actual label y.

$$\text{error} = |\hat{y} - y|$$

Types of error Functions:

Different loss functions will give different errors for the same prediction, and thus have a considerable effect on the performance of the model. two most commonly used loss functions: mean squared error (and its variations), usually used for regression problems, and cross-entropy, used for classification problems.

a. Mean squared error (MSE) is commonly used in regression problems that require the output to be a real value (like house pricing). Instead of just comparing the prediction output with the label (yˆi – yi), the error is squared and averaged over the number of data points, as you see in this equation

$$E(W, b) = \frac{1}{N}\sum_{i=1}^{N} (\hat{y}_i - y_i)^2$$

MSE is a good choice for a few reasons. The square ensures the error is always positive, and larger errors are penalized more than smaller errors. MSE is quite sensitive to outliers, since it squares the error value. This might not be an issue for the specific problem that you are solving. In fact, this sensitivity to outliers might be beneficial in some cases. For example, if you are predicting a stock price, you would want to take outliers into account, and sensitivity to outliers would be a good thing. In other scenarios, you wouldn't want to build a model that is skewed by outliers, such as predicting a house price in a city. In that case, you are more interested in the median and less in the mean.

b. mean absolute error (MAE) was developed just for this purpose. It averages the absolute error over the entire dataset without taking the square of the error.

$$E(W, b) = \frac{1}{N}\sum_{i=1}^{N} |\hat{y}_i - y_i|$$

c. Cross-entropy is commonly used in classification problems because it quantifies the difference between two probability distributions.
For example, suppose that for a specific training instance, we are trying to classify a dog image out of three possible classes (dogs, cats, fish). The true distribution for this training instance is as follows:

Probability(cat)	P(dog)	P(fish)
0.0	1.0	0.0

We can interpret this "true" distribution to mean that the training instance has 0% probability of being class A, 100% probability of being class B, and 0% probability of being class C. Now, suppose our machine learning algorithm predicts the following probability distribution:

Probability(cat)	P(dog)	P(fish)
0.2	0.3	0.5

How close is the predicted distribution to the true distribution? That is what the crossentropy loss function determines. We can use this formula:

$$E(W, b) = -\sum_{i=1}^{m} \hat{y}_i \log(p_i)$$

where (y) is the target probability, (p) is the predicted probability, and (m) is the number of classes. The sum is over the three classes: cat, dog. the loss is 1.2

```
E = - (0.0 * log(0.2) + 1.0 * log(0.3) + 0.0 * log(0.5)) = 1.2
```

To calculate the cross-entropy error across all the training examples (n), we use this general formula:

$$E(W, b) = -\sum_{i=1}^{n}\sum_{i=1}^{m} \hat{y}_{ij} \log(p_{ij})$$

Optimization algorithms:

Training a neural network involves showing the network many examples (a training dataset); the network makes predictions through feedforward calculations and compares them with the correct labels to calculate the error. Finally, the neural network needs to adjust the weights (on all edges) until it gets the minimum error value, which means maximum accuracy. Now, all we need to do is build algorithms that can find the optimum weights for us.

<u>What is optimization?</u>

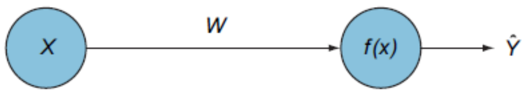
Optimization is a way of framing a problem to maximize or minimize some value. The best thing about computing an error function is that we turn the neural network into an optimization problem where our goal is to minimize the error.

Suppose you want to optimize your commute from home to work. First, you need to define the metric that you are optimizing (the error function). Maybe you want to optimize the cost of the commute, or the time, or the distance. Then, based on that specific loss function, you work on minimizing its value by changing some parameters.

Changing the parameters to minimize (or maximize) a value is called optimization. If you choose the loss function to be the cost, maybe you will choose a longer commute that will take two hours, or (hypothetically) you might walk for five hours to minimize the cost. On the other hand, if you want to optimize the time spent commuting, maybe you will spend $50 to take a cab that will decrease the commute time to 20 minutes.

Based on the loss function you defined, you can start changing your parameters to get the results you want.

Let's look at the space that we are trying to optimize:



In a neural network of the simplest form, a perceptron with one input, we have only one weight. We can easily plot the error (that we are trying to minimize) with respect to this weight, represented by the 2D curve in figure below

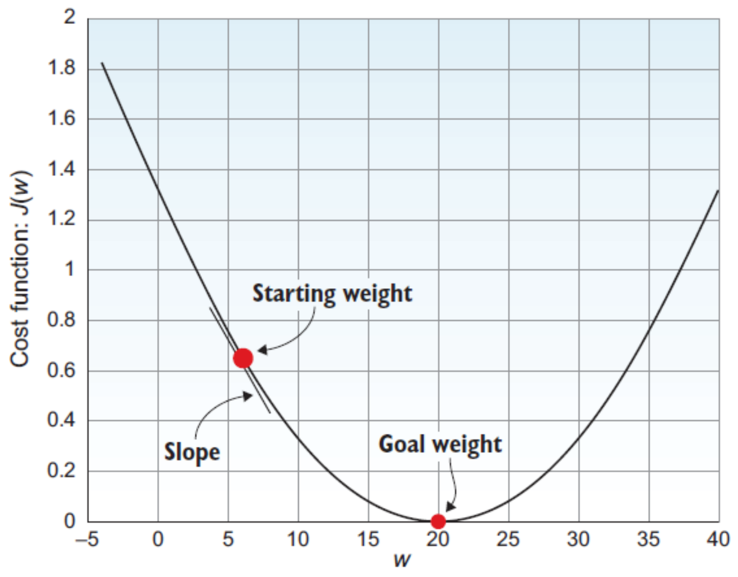


Figure 2.26 The error function with respect to its weight for a single perceptron is a 2D curve.

If we graph all the possible values of the two weights, we get a 3D plane of the error. Your network will probably have hundreds or thousands of weights (because each edge in your network has its own weight value).

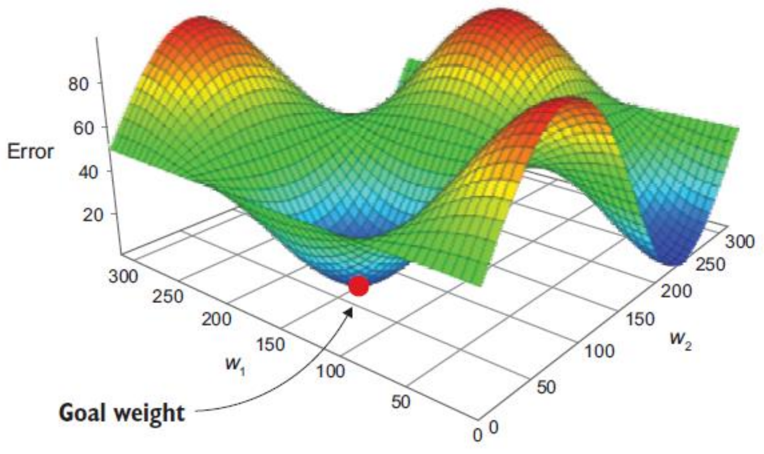


Figure 2.27 Graphing all possible values of two weights gives a 3D error plane.

In order to optimize the model, our goal is to search this space to find the best weights that will achieve the lowest possible error.

Why do we need an optimization algorithm?

Can't we just brute-force through a lot of weight values until we get the minimum error? Suppose we used a brute-force approach where we just tried a lot of different possible weights (say 1,000 values) and found the weight that produced the minimum error.

This approach might work when we have very few inputs and only one or two neurons in our network. Suppose we want to predict house prices based on only four features (inputs) and one hidden layer of five neurons (see figure below)

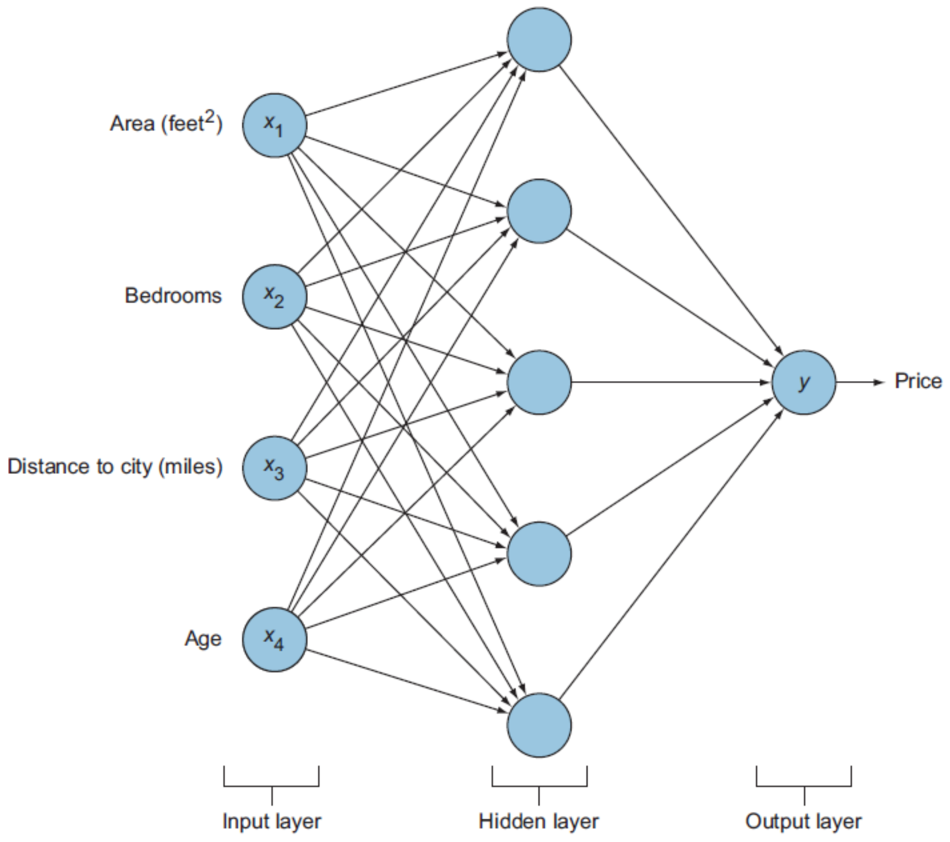


Figure 2.28 If we want to predict house prices based on only four features (inputs) and one hidden layer of five neurons, we'll have 20 edges (weights) from the input to the hidden layer, plus 5 weights from the hidden layer to the output prediction.

As you can see, we have 20 edges (weights) from the input to the hidden layer, plus 5 weights from the hidden layer to the output prediction, totaling 25 weight variables that need to be adjusted for optimum values. To brute-force our way through a simple neural network of this size, if we are trying 1,000 different values for each weight, then we will have a total of 1075 combinations:

1,000 × 1,000 × . . . × 1,000 = 1,00025 = 1075 combinations.

The brute-forcing through the optimization process is not the answer. Now, let's study the most popular optimization algorithm for neural networks: gradient descent. Gradient descent has several variations: batch gradient descent (BGD), stochastic gradient descent (SGD), and mini-batch GD (MB-GD).

Batch gradient descent:

The general definition of a gradient (also known as a derivative) is that it is the function that tells you the slope or rate of change of the line that is tangent to the curve at any given point. It is just a fancy term for the slope or steepness of the curve (figure below).

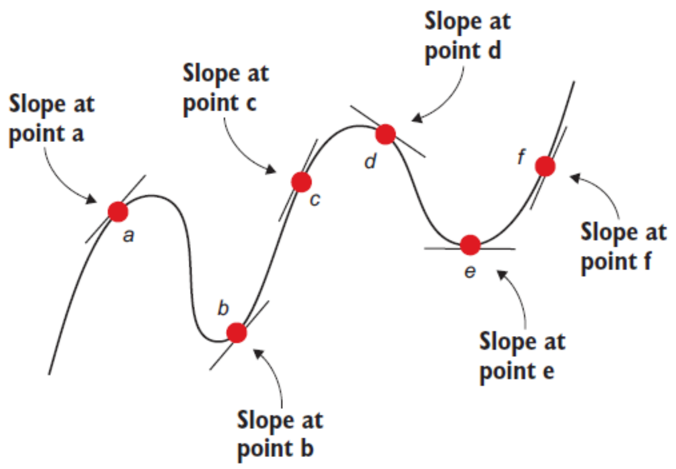


Figure 2.29 A gradient is the function that describes the rate of change of the line that is tangent to a curve at any given point.

Gradient descent simply means updating the weights iteratively to descend the slope of the error curve until we get to the point with minimum error. Let's take a look at the error function that we introduced earlier with respect to the weights. At the initial weight point, we calculate the derivative of the error function to get the slope (direction) of the next step. We keep repeating this process to take steps down the curve until we reach the minimum error (figure below).

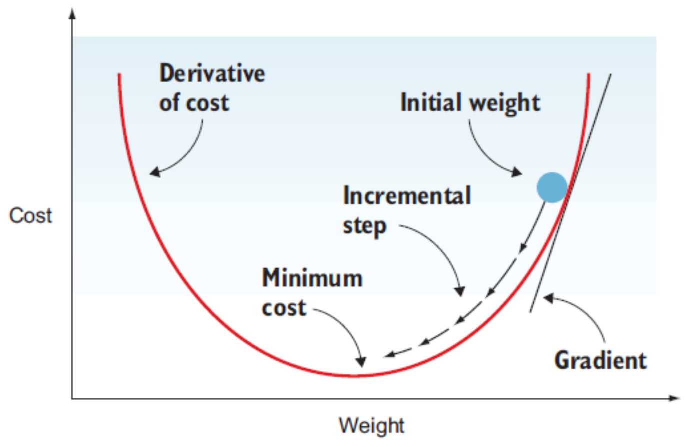


Figure 2.30 Gradient descent takes incremental steps to descend the error function.

HOW DOES GRADIENT DESCENT WORK?

To visualize how gradient descent works, let's plot the error function in a 3D graph (figure below) and go through the process step by step.

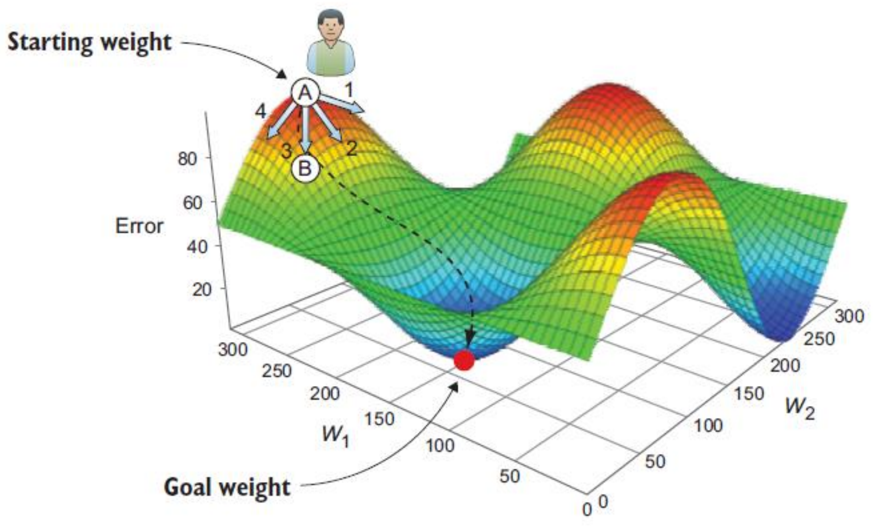


Figure 2.31 The random initial weight (starting weight) is at point A. We descend the error mountain to the w_1 and w_2 weight values that produce the minimum error value.

The random initial weight (starting weight) is at point A, and our goal is to descend this error mountain to the goal *w*1 and *w*2 weight values, which produce the minimum error value. The way we do that is by taking a series of steps *down* the curve until we get the minimum error. In order to descend the error mountain, we need to determine two things for each step:

- The step direction (gradient)
- The step size (learning rate)

THE DIRECTION (GRADIENT)

Suppose you are standing on the top of the error mountain at point A. To get to the bottom, you need to determine the step direction that results in the deepest descent (has the steepest slope). And what is the slope, again? It is the derivative of the curve. So if you are standing on top of that mountain, you need to look at all the directions around you and find out which direction will result in the deepest descent (1, 2, 3, or 4, for example). Let's say it is direction 3; we choose that way. This brings us to point B, and we restart the process (calculate feedforward and error) and find the direction of deepest descent, and so forth, until we get to the bottom of the mountain.

This process is called *gradient descent*. By taking the derivative of the error with respect to the weight ($\frac{dE}{dW}$), we get the direction that we should take. Now there's one thing left. The gradient only determines the direction. How large should the size of the step be? It could be a 1-foot step or a 100-foot jump. This is what we need to determine next.

THE STEP SIZE (LEARNING RATE α)

The learning rate is the size of each step the network takes when it descends the error mountain, and it is usually denoted by the Greek letter alpha (α). It is one of the most important hyperparameters that you tune when you train your neural network (more on that later). A larger learning rate means the network will learn faster (since it is descending the mountain with larger steps), and smaller steps mean slower learning.

Well, this sounds simple enough. Let's use large learning rates and complete the neural network training in minutes instead of waiting for hours. Right? Not quite. Let's take a look at what could happen if we set a very large learning rate value. In figure below, you are starting at point A. When you take a large step in the direction of the arrow, instead of descending the error mountain, you end up at point B, on the other side. Then another large step takes you to C, and so forth. The error will keep oscillating and will never descend. We will talk more later about tuning the learning rate and how to determine if the error is oscillating.

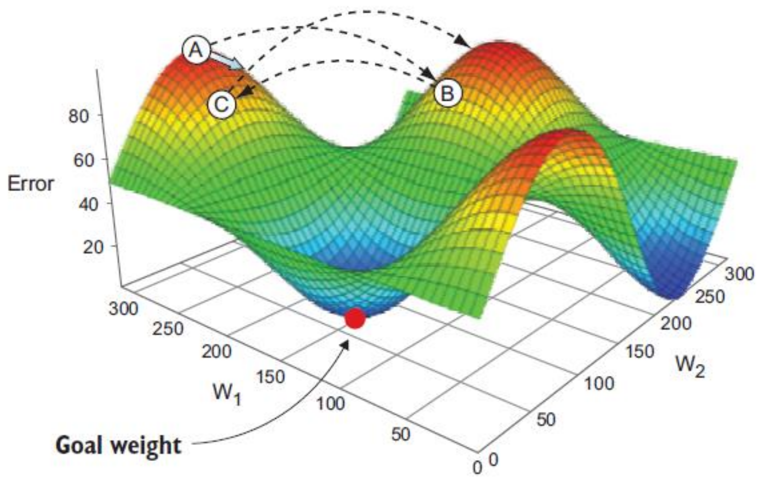


Figure 2.32 Setting a very large learning rate causes the error to oscillate and never descend.

But for now, you need to know this: if you use a very small learning rate, the network will eventually descend the mountain and will get to the minimum error. But this training will take longer (maybe weeks or months). On the other hand, if you use a very large learning rate, the network might keep oscillating and never train. So we usually initialize the learning rate value to 0.1 or 0.01 and see how the network performs, and then tune it further.

PUTTING DIRECTION AND STEP TOGETHER

By multiplying the direction (derivative) by the step size (learning rate), we get the change of the weight for each step:

$$\Delta w_i = -\alpha \frac{dE}{dw_i}$$

We add the minus sign because the derivative always calculates the slope in the upward direction. Since we need to descend the mountain, we go in the opposite direction of the slope:

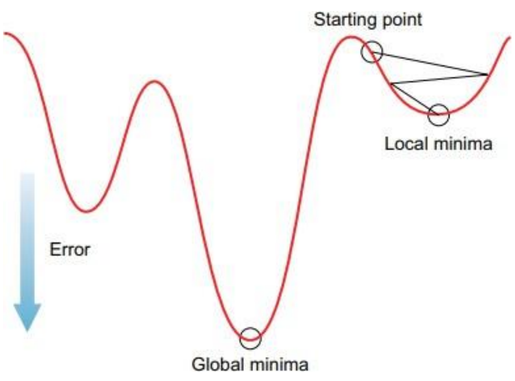
$$w_{\text{next-step}} = w_{\text{current}} + \Delta w$$

i) Batch GD updates the weights after computing the gradient of all the training data. This can be computationally very expensive when the data is huge. It doesn't scale well. Batch Gradient descent is a very powerful algorithm to get to the minimum error. But it has two major pitfalls. First, not all cost functions look like the simple bowls we saw earlier. There may be holes, ridges, and all sorts of irregular terrain that make reaching the minimum error very difficult. Consider figure below, where the error function is a little more complex and has ups and downs. Complex error functions are represented by more complex curves with many local minima values. Our goal is to reach the global minimum value.

PITFALLS OF BATCH GRADIENT DESCENT:

Gradient descent is a very powerful algorithm to get to the minimum error. But it has two major pitfalls.

First, not all cost functions look like the simple bowls we saw earlier. There may be holes, ridges, and all sorts of irregular terrain that make reaching the minimum error very difficult. Consider figure 2.33, where the error function is a little more complex and has ups and downs.



Remember that during weight initialization, the starting point is randomly selected. What if the starting point of the gradient descent algorithm is as shown in this figure? The error will start descending the small mountain on the right and will indeed reach a minimum value. But this minimum value, called the *local minima*, is not the lowest possible error value for this error function. It is the minimum value for the local mountain where the algorithm randomly started. Instead, we want to get to the lowest possible error value, the *global minima*.

Second, batch gradient descent uses the entire training set to compute the gradients at every step. Remember this loss function?

$$L(W, b) = \frac{1}{N}\sum_{i=1}^{N} (\hat{y}_i - y_i)^2$$

This means that if your training set (*N*) has 100,000,000 (100 million) records, the algorithm needs to sum over 100 million records just to take *one step*. That is computationally very expensive and slow. And this is why this algorithm is also called *batch gradient descent*—because it uses the entire training data in one batch. One possible approach to solving these two problems is stochastic gradient descent.

Stochastic gradient descent:

In stochastic gradient descent, the algorithm randomly selects data points and goes through the gradient descent one data point at a time (figure 2.34). This provides many different weight starting points and descends all the mountains to calculate their local minimas. Then the minimum value of all these local minimas is the global minima. This sounds very intuitive; that is the concept behind the SGD algorithm.

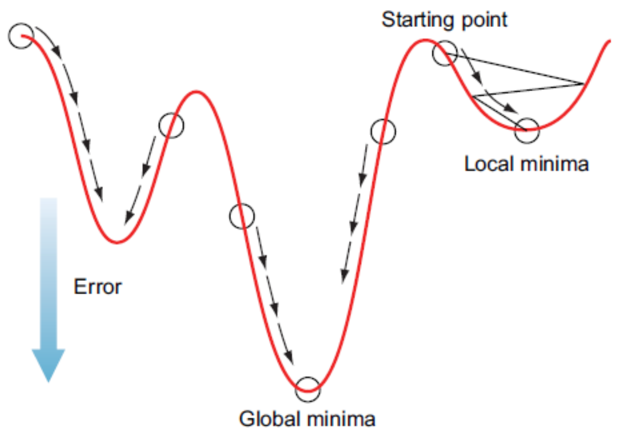
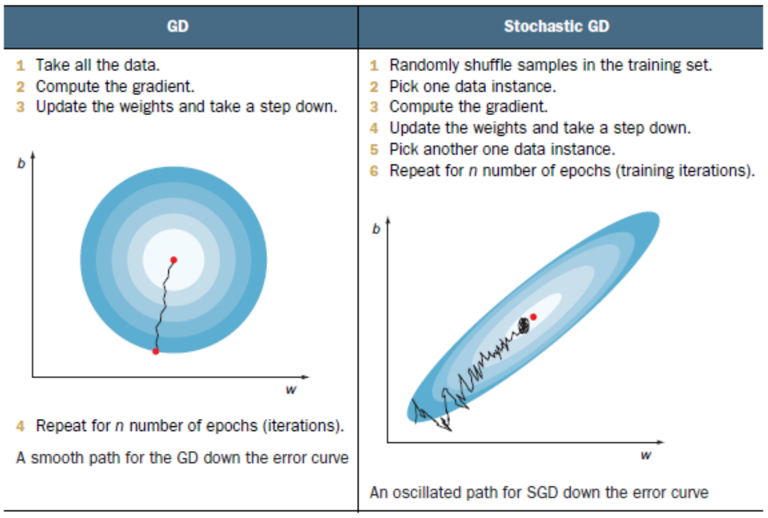


Figure 2.34 The stochastic gradient descent algorithm randomly selects data points across the curve and descends all of them to find the local minima.

Stochastic is just a fancy word for *random*. Stochastic gradient descent is probably the most-used optimization algorithm for machine learning in general and for deep learning in particular. While gradient descent measures the loss and gradient over the full training set to take one step toward the minimum, SGD *randomly* picks *one instance* in the training set for each one step and calculates the gradient based only on that single instance. Let's take a look at the pseudocode of both GD and SGD to get a better understanding of the differences between these algorithms:



GD	Stochastic GD
1 Take all the data. 2 Compute the gradient. 3 Update the weights and take a step down.	1 Randomly shuffle samples in the training set. 2 Pick one data instance. 3 Compute the gradient. 4 Update the weights and take a step down. 5 Pick another one data instance. 6 Repeat for *n* number of epochs (training iterations).
4 Repeat for *n* number of epochs (iterations). A smooth path for the GD down the error curve	An oscillated path for SGD down the error curve

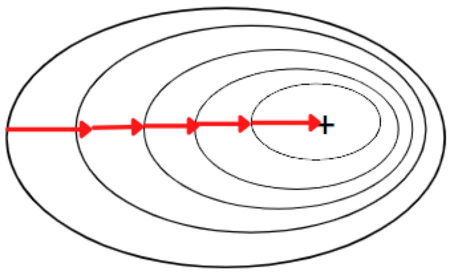
Because we take a step after we compute the gradient for the entire training data in batch GD, you can see that the path down the error is smooth and almost a straight line. In contrast, due to the stochastic (random) nature of SGD, you see the path toward the global cost minimum is not direct but may zigzag if we visualize the cost surface in a 2D space. That is because in SGD, every iteration tries to better fit just a single training example, which makes it a lot faster but does not guarantee that every step takes us a step down the curve. It will arrive close to the global minimum and, once it gets there, it will continue to bounce around, never settling down. In practice, this isn't a problem because ending up very close to the global minimum is good enough for most practical purposes. SGD almost always performs better and faster than batch GD.

2.6.4 Mini-batch gradient descent

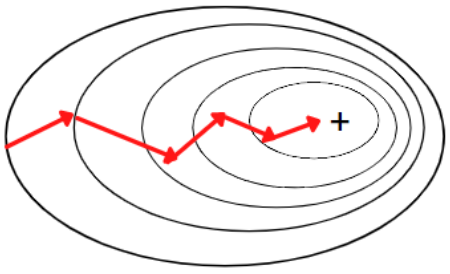
Mini-batch gradient descent (MB-GD) is a compromise between BGD and SGD. Instead of computing the gradient from one sample (SGD) or all samples (BGD), we divide the training sample into *mini-batches* from which to compute the gradient (a common mini-batch size is *k* = 256). MB-GD converges in fewer iterations than BGD because we update the weights more frequently;

however, MB-GD lets us use vectorized operations, which typically result in a computational performance gain over SGD.

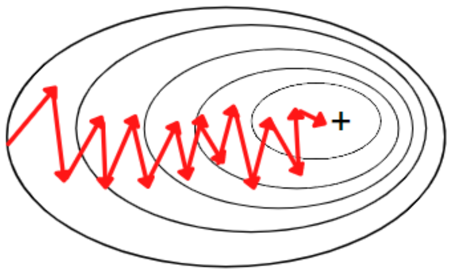
Batch Gradient Descent



Mini-Batch Gradient Descent



Stochastic Gradient Descent



Vanilla Gradient Descent	**Stochastic Gradient Descent**	**Mini Batch Gradient Decent**
Computes gradient using the whole Training sample	Computes gradient using a single Training sample	Computes gradient using the Subset of Training sample
Slow and computationally expensive algorithm	Faster and less computationally expensive than Vanilla GD	Computation time is lesser than SGD Computation cost is lesser than Vanilla Gradient Descent
Not suggested for huge training samples.	Can be used for large training samples. But it maybe slow when datasets are huge.	Can be used for large training samples and it is also faster than SGD.
Cost Function reduces smoothly	Lot of variations in cost function	Smoother cost function as compared to SGD
Gives optimal solution given sufficient time to converge.	Gives good solution but not optimal.	Gives optimal solution in less time compared to SGD
No random shuffling of points are required.	The data sample should be in a random order, and this is why we want to shuffle the training set for every epoch.	The Data sample is Shuffled in a random order and then divided into batches.
Can't escape shallow local minima easily.	SGD can escape shallow local minima more easily.	Mini Batch can escape local minima easily compared to vanilla Gradient Descent

Vanilla Gradient Descent	**Stochastic Gradient Descent**	**Mini Batch Gradient Decent**
Convergence is slow.	Reaches the convergence much faster.	At times mini batch can reach convergence faster than SGD.
Vanilla Gradient descent is generally used for Small databases that fit into computer memory	SGD is a basis of more advanced stochastic algorithms used in training artificial neural networks	Mini batch gradient descent is most commonly used in practical applications

Gradient descent takeaways:

There is a lot going on here, so let's sum it up, shall we? Here is how gradient descent is summarized in my head:

Three types: **batch, stochastic, and mini-batch.**

All follow the same concept:

– Find the direction of the steepest slope: the derivative of the error with respect to the weight .

– Set the learning rate (or step size). The algorithm will compute the slope, but you will set the learning rate as a hyperparameter that you will tune by trial and error.

– Start the learning rate at 0.01, and then go down to 0.001, 0.0001, 0.00001. The lower you set your learning rate, the more guaranteed you are to descend to the minimum error (if you train for an infinite time). Since we don't have infinite time, 0.01 is a reasonable start, and then we go down from there.

 Batch GD updates the weights after computing the gradient of *all* the training data. This can be computationally very expensive when the data is huge. It doesn't scale well.

 Stochastic GD updates the weights after computing the gradient of a single instance of the training data. SGD is faster than BGD and usually reaches very close to the global minimum.

 Mini-batch GD is a compromise between batch and stochastic, using neither all the data nor a single instance. Instead, it takes a group of training instances (called a mini-batch), computes the gradient on them and updates the weights, and then repeats until it processes all the training data. In most cases, MB-GD is a good starting point.

– batch_size is a hyperparameter that you will tune. This will come up again in the hyperparameter-tuning section in chapter 4. But typically, you can start experimenting with batch_size = 32, 64, 128, 256.

– Don't get *batch_size* confused with *epochs*. An *epoch* is the full cycle over all the training data. The batch is the number of training samples in the group for which we are computing the gradient. For example, if we have 1,000 samples in our training data and set batch_size = 256, then epoch 1 =

batch 1 of 256 samples plus batch 2 (256 samples) plus batch 3 (256 samples) plus batch 4 (232 samples).

What is Backpropagation?

Backpropagation is the core of how neural networks learn. Up until this point, you learned that training a neural network typically happens by the repetition of the following three steps:

- **Feedforward:** get the linear combination (weighted sum), and apply the activation function to get the output prediction ($\hat{y}$):

$$\hat{y} = \sigma \cdot W^{(3)} \cdot \sigma \cdot W^{(2)} \cdot \sigma \cdot W^{(1)} \cdot (x)$$

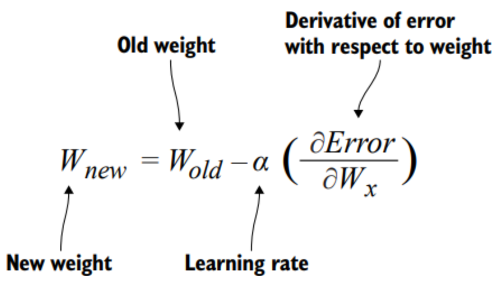
- Compare the prediction with the label to **calculate the error** or loss function:

$$E(W, b) = |\hat{y}i - yi|$$

- Use a **gradient descent optimization** algorithm to compute the Δw that optimizes the error function:

$$\Delta wi = -\alpha \, dE/dw_i$$

- **Backpropagate** the Δw through the network to update the weights:


Old weight
Derivative of error with respect to weight
$W_{new} = W_{old} - \alpha \left(\frac{\partial Error}{\partial W_x}\right)$
New weight
Learning rate

Backpropagation, or backward pass, means propagating derivatives of the error with respect to each specific weight dE/dW_i .

from the last layer (output) back to the first layer (inputs) to adjust weights. By propagating the change in weights Δw backward from the prediction node ($\hat{y}$) all the way through the hidden layers and back to the input layer, the weights get updated:

$$(w_{next-step} = w_{current} + \Delta w)$$

This will take the error one step down the error mountain. Then the cycle starts again (steps 1 to 3) to update the weights and take the error another step down, until we get to the minimum error.

Backpropagation might sound clearer when we have only one weight. We simply adjust the weight by adding the Δw to the old weight $w_{new} = w - \alpha \, dE/dw_i$.

But it gets complicated when we have a multilayer perceptron (MLP) network with many weight variables. To make this clearer, consider the scenario in figure below.

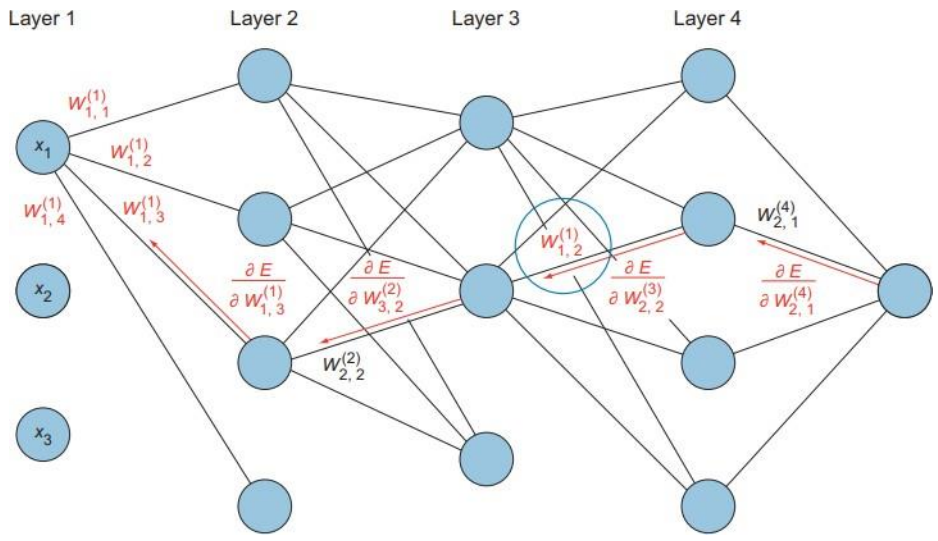

Layer 1
Layer 2
Layer 3
Layer 4
x_1
x_2
x_3
$w_{1,1}^{(1)}$
$w_{1,2}^{(1)}$
$w_{1,4}^{(1)}$
$w_{1,3}^{(1)}$
$\frac{\partial E}{\partial w_{1,3}^{(1)}}$
$\frac{\partial E}{\partial w_{3,2}^{(2)}}$
$w_{2,2}^{(2)}$
$w_{1,2}^{(1)}$
$\frac{\partial E}{\partial w_{2,2}^{(3)}}$
$w_{2,1}^{(4)}$
$\frac{\partial E}{\partial w_{2,1}^{(4)}}$

Figure 10.h. Backpropagation in MLP network with many weights' variables

How do we compute the change of the total error with respect to dE/dw$_{13}$?

How much will the total error change when we change the parameter w$_{13}$?

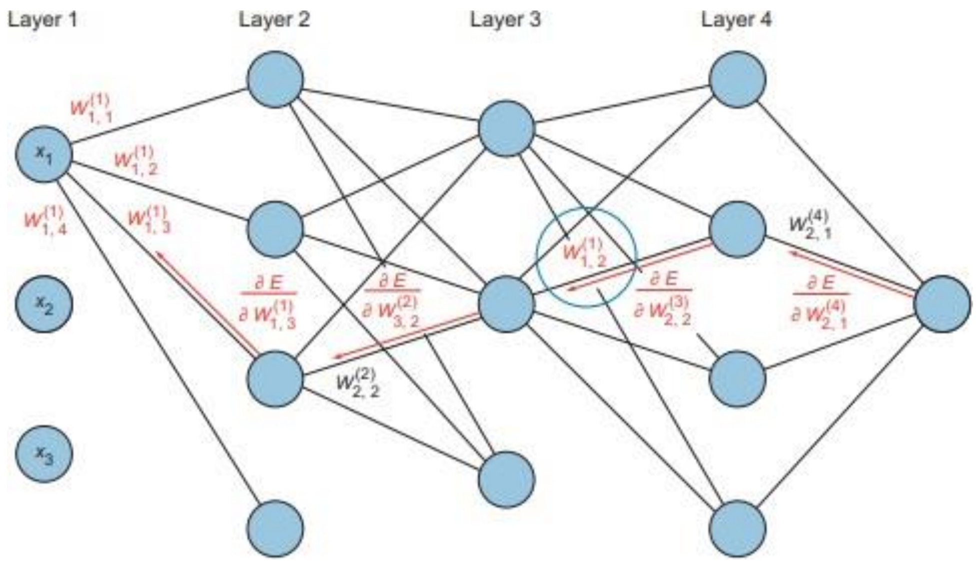
how to compute by applying the derivative rules on the error function.

That is straightforward because w21 is directly connected to the error function. But to compute the derivatives of the total error with respect to the weights all the way back to the input, we need a calculus rule called the **chain rule.**

Figure 10.i shows how backpropagation uses the chain rule to flow the gradients in the backward direction through the network. Let's apply the chain rule to calculate the derivative of the error with respect to the third weight on the first input w$_{1,3}$ (1) , where the (1) means layer 1, and w$_{1,3}$ means node number 1 and weight number 3:

$$\frac{dE}{dw_{1,3}^{(1)}} = \frac{dE}{dw_{2,1}^{(4)}} \times \frac{dw_{2,1}^{(4)}}{dw_{2,2}^{(3)}} \times \frac{dw_{2,2}^{(3)}}{dw_{3,2}^{(2)}} \times \frac{dw_{3,2}^{(2)}}{dw_{1,3}^{(1)}}$$

---→Chain rule


Layer 1
Layer 2
Layer 3
Layer 4
$w_{1,1}^{(1)}$
$w_{1,2}^{(1)}$
$w_{1,4}^{(1)}$
$w_{1,3}^{(1)}$
x_1
x_2
x_3
$\frac{\partial E}{\partial w_{1,3}^{(1)}}$
$\frac{\partial E}{\partial w_{3,2}^{(2)}}$
$w_{2,2}^{(2)}$
$w_{1,2}^{(1)}$
$\frac{\partial E}{\partial w_{2,2}^{(3)}}$
$w_{2,1}^{(4)}$
$\frac{\partial E}{\partial w_{2,1}^{(4)}}$

The error back propagated to the edge w$_{1,3}$ $^{(1)}$ = effect of error on edge 4 · effect on edge 3 · effect on edge 2 · effect on target edge.

Chain rule in derivatives:

Back again to calculus. Remember the derivative rules that we listed earlier? One of the most important rules is the chain rule. Let's dive deep into it to see how it is implemented in backpropagation:

Chain Rule: $f(g(x)) = f'(g(x))g'(x)$

The chain rule is a formula for calculating the derivatives of functions that are composed of functions inside other functions. It is also called the *outside-inside rule*.

Look at this:

$$\frac{d}{dx} f(g(x)) = \frac{d}{dx} \text{outside function} \times \frac{d}{dx} \text{inside function}$$
$$= \frac{d}{dx} f(g(x)) \times \frac{d}{dx} g(x)$$

The chain rule says, “When composing functions, the derivatives just multiply.” That is going to be very useful for us when implementing backpropagation, because feedforwarding is just composing a bunch of functions, and backpropagation is taking the derivative at each piece of this function.

To implement the chain rule in backpropagation, all we are going to do is multiply a bunch of partial derivatives to get the effect of errors all the way back to the input. Here is how it works—but first, remember that our goal is to propagate the error backward all the way to the input layer. So in the following example, we want to calculate , which is the effect of total error on input (x):

$$\begin{tikzcd} X & A \arrow[l] & B \arrow[l] & E \arrow[l] \end{tikzcd}$$

$$\frac{dE}{dx} = \frac{dE}{dB} \cdot \frac{dB}{dA} \cdot \frac{dA}{dx}$$

All we do here is multiply the upstream gradient by the local gradient all the way until we get to the target value.

Thus, the backpropagation technique is used by neural networks to update the weights to solve the best fit problem.