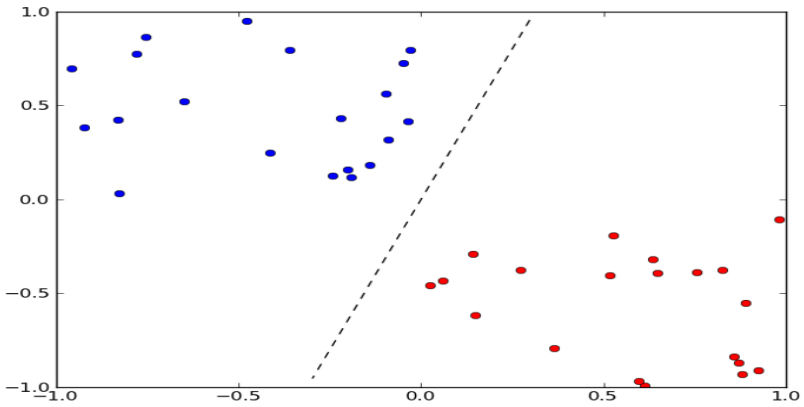
- What is perceptron
- How does it work?
- Activation functions
- Bias
- Neural networks
- Single layer perceptron (treshold function, fully connected)
- Multilayer perceptron
- Feedforward propogation
- Backprop (lost function error function and how it works)
- Lınear regression
- Cnn
- Ovefitting/underfitting
- Machine learning (and types)
- MNIST dataset
- CIFAR 10
- CIFAR 100

What is perceptron

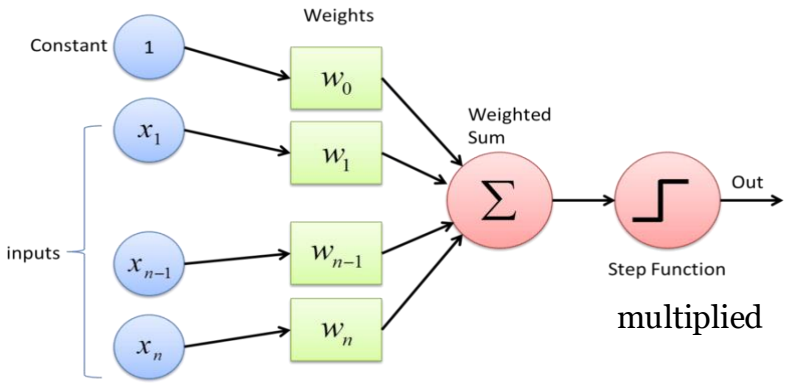
Perceptron is a linear classifier (binary). Also, it is used in supervised learning. It helps to classify the given input data. Therefore, it is also known as a Linear Binary Classifier.



How it works?

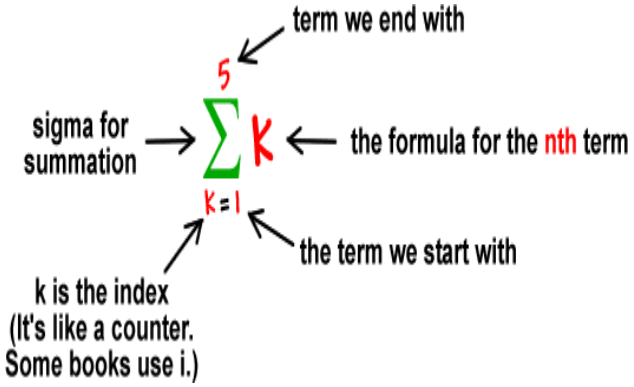
The perceptron consists of 4 parts .

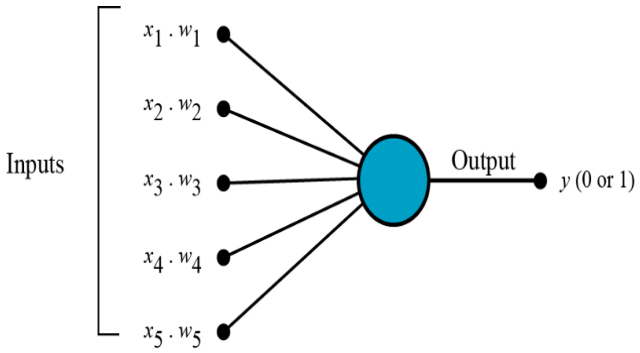
1. Input values or One input layer
2. Weights and Bias
3. Net sum
4. Activation Function



The perceptron works on these simple steps.all inputs x are multiplied with their weight w let's call it K. ***Add*** all the multiplied values and call them ***Weighted Sum.***

And finally ***Apply*** that weighted sum to the correct ***Activation Function.***





So what is ACTIVATION FUNCTION ? AND WHY IT IS USED?

It's just a thing (**node**) that you add to the output end of any neural network. It is also known as **Transfer Function**. It can also be attached in between two Neural Networks.

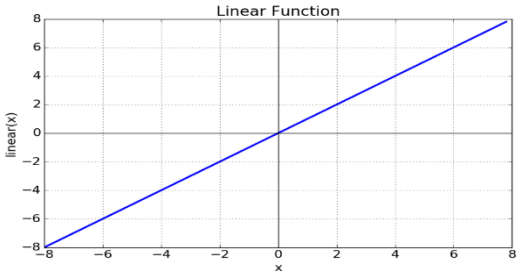
It is used to determine the output of neural network like yes or no. It maps the resulting values in between 0 to 1 or -1 to 1 etc. (depending upon the function).

There are two types of Activation functions

- Linear activation function
- Non-linear activation functions

Linear or identity activation function

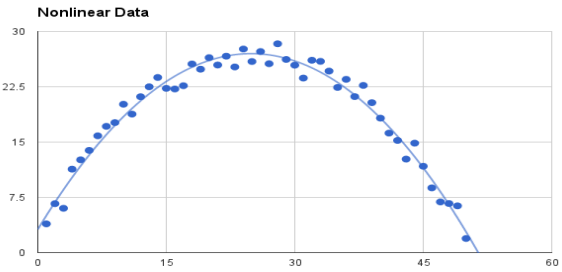
As you can see the function is a line or linear.Therefore, the output of the functions will not be confined between any range.



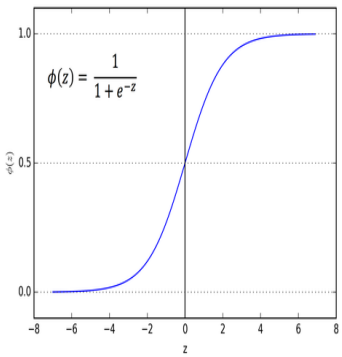
Equation : f(x)=x

Range : infinity to infinity

Non-linear activation functions



The Nonlinear Activation Functions are the most used activation functions. The Nonlinear Activation Functions are mainly divided on the basis of their **range or curves**.

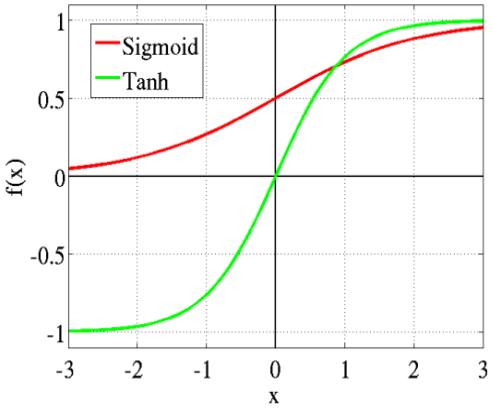


Sigmoid or Logistic Activation Function

The Sigmoid Function curve looks like a S-shape.

The main reason why we use sigmoid function is because it exists between **(0 to 1).** Therefore, it is especially used for models where we have to **predict the probability** as an output.Since probability of anything exists only between the range of **0 and 1,** sigmoid is the right choice.

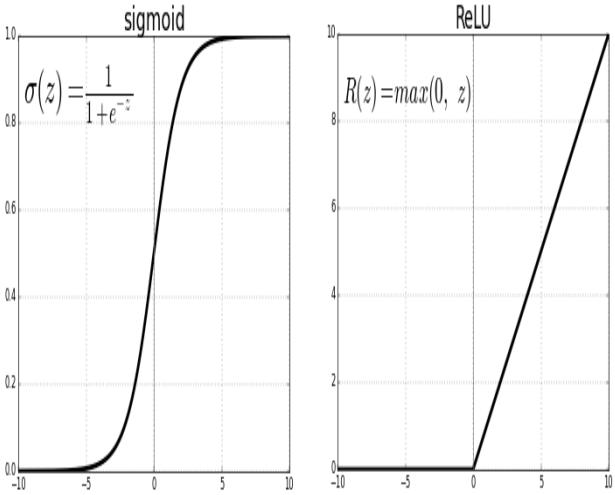
Tanh or hyperbolic tangent Activation Function



tanh is also like logistic sigmoid but better. The range of the tanh function is from (-1 to 1). tanh is also sigmoidal (s - shaped). The advantage is that the negative inputs will be mapped strongly negative and the zero inputs will be mapped near zero in the tanh graph.

Both tanh and logistic sigmoid activation functions are used in feed-forward nets.

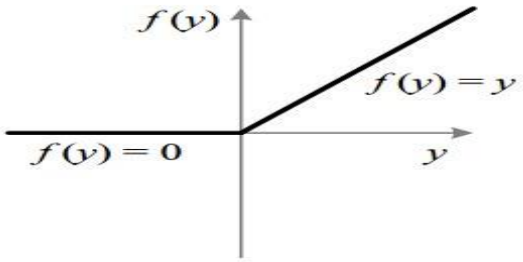
3. ReLU (Rectified Linear Unit) Activation Function

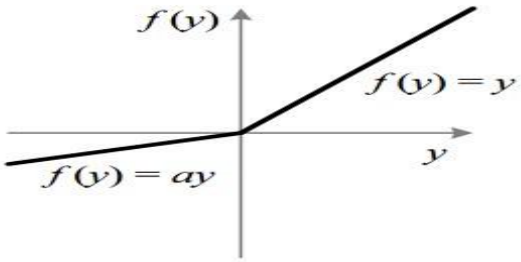


The ReLU is the most used activation function in the world right now.Since, it is used in almost all the convolutional neural networks or deep learning. As you can see, the ReLU is half rectified (from bottom). f(z) is zero when z is less than zero and f(z) is equal to z when z is above or equal to zero. **Range:** [0 to infinity)

4.Leaky ReLU

The leak helps to increase the range of the ReLU function. Usually, the value of **a** is 0.01 or so.When **a is not 0.01** then it is called **Randomized ReLU**.Therefore the **range** of the Leaky ReLU is (-infinity to infinity).





Bias

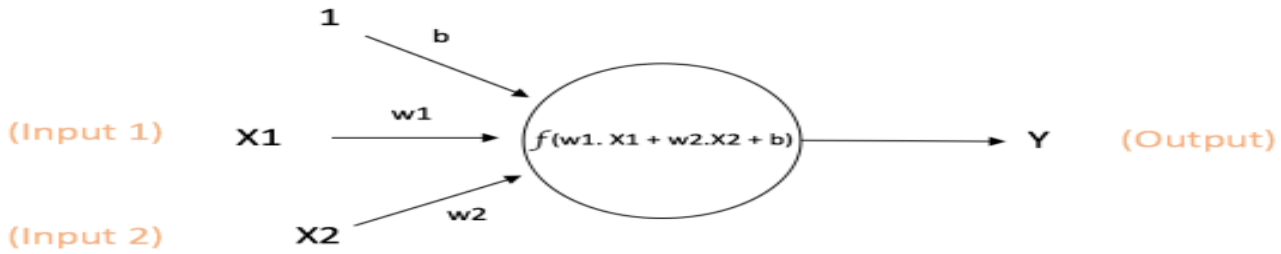
Importance of Bias: The main function of Bias is to provide every node with a trainable constant value (in addition to the normal inputs that the node receives). *A bias* value allows you to shift the activation function curve up or down.

Neural network(overview)

An Artificial Neural Network (ANN) is a computational model that is inspired by the way biological neural networks in the human brain process information.

A Single Neuron

The basic unit of computation in a neural network is the **neuron**, often called a **node** or **unit**. It receives input from some other nodes, or from an external source and computes an output. Each input has an associated **weight** (w), which is assigned on the basis of its relative importance to other inputs. The node applies a function ***f*** (defined below) to the weighted sum of its inputs as shown in Figure 1 below:



Output of neuron = Y= $f(w1.\ X1 + w2.X2 + b)$

The above network takes numerical inputs **X1** and **X2** and has weights **w1** and **w2** associated with those inputs. Additionally, there is another input **1** with weight **b** (called the **Bias**) associated with it.

Activation function and bias yuxarida

Feedforward neural network

The feedforward neural network was the first and simplest type of artificial neural network devised . It contains multiple neurons (nodes) arranged in **layers**. Nodes from adjacent layers have **connections** or **edges** between them. All these connections have **weights** associated with them.

An example of a feedforward neural network is shown in Figure 3.

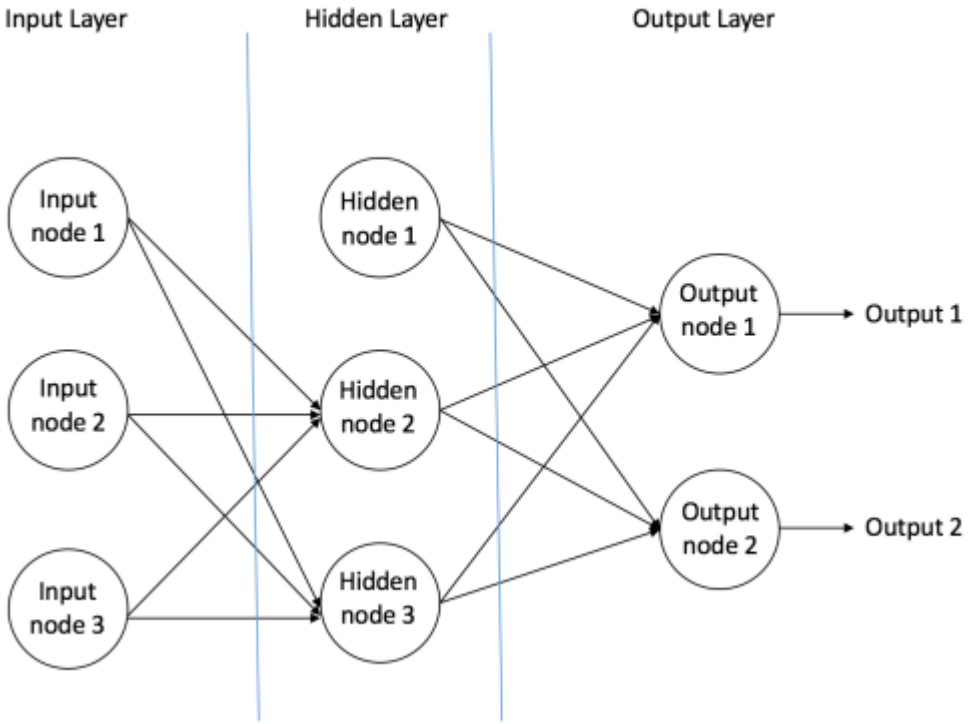


Figure 3: an example of feedforward neural network

A feedforward neural network can consist of three types of nodes:

1. **Input Nodes –** The Input nodes provide information from the outside world to the network and are together referred to as the "Input Layer". No computation is performed in any of the Input nodes – they just pass on the information to the hidden nodes.
2. **Hidden Nodes –** The Hidden nodes have no direct connection with the outside world (hence the name "hidden"). They perform computations and transfer information from the input nodes to the output nodes. A collection of hidden nodes forms a "Hidden Layer". While a feedforward network will only have a single input layer and a single output layer, it can have zero or multiple Hidden Layers.
3. **Output Nodes –** The Output nodes are collectively referred to as the "Output Layer" and are responsible for computations and transferring information from the network to the outside world.

In a feedforward network, the information moves in only one direction – forward – from the input nodes, through the hidden nodes (if any) and to the output nodes. There are no cycles or loops in the network (this property of feed forward networks is different from Recurrent Neural Networks in which the connections between the nodes form a cycle).

Two examples of feedforward networks are given below:

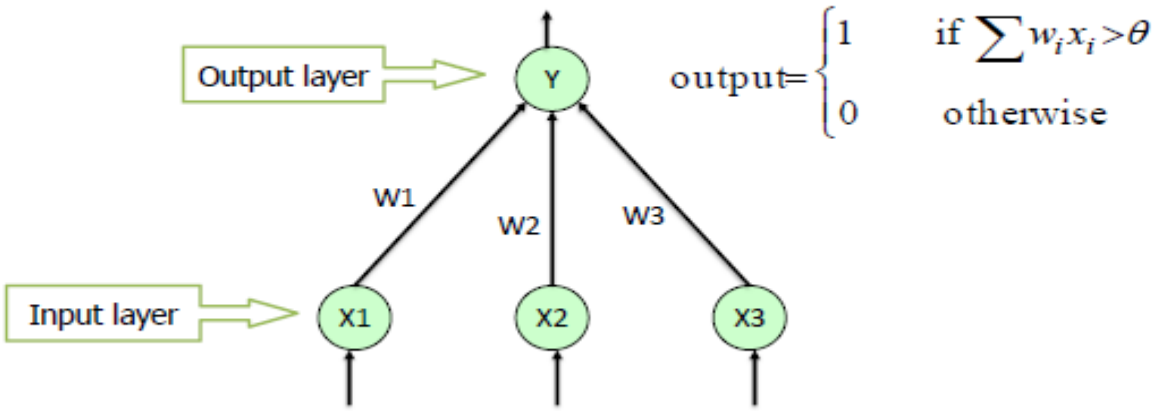
1. **Single Layer Perceptron** – This is the simplest feedforward neural network and does not contain any hidden layer.

2. **Multi Layer Perceptron** – A Multi Layer Perceptron has one or more hidden layers. We will only discuss Multi Layer Perceptrons below since they are more useful than Single Layer Perceptons for practical applications today.

Single layer perceptron

A single layer perceptron (**SLP**) is a feed-forward network based on a threshold transfer function. SLP is the simplest type of artificial neural networks and can only classify linearly separable cases with a binary target (1 , 0).

Single Layer Perceptron



Algorithm

The single layer perceptron does not have a priori knowledge, so the initial weights are assigned randomly. SLP sum the weighted inputs and if the sum is above the threshold (some predetermined value), SLP is said to be activated (output=1).

Output

$$w_1 x_1 + w_2 x_2 + \ldots + w_n x_n > \theta \Rightarrow 1$$

$$w_1 x_1 + w_2 x_2 + \ldots + w_n x_n \leq \theta \Rightarrow 0$$

The input values are presented to the perceptron, and if the predicted output is the same as the desired output, the the performance is considered satisfactory and no changes to the weights are made. However, if the output does n match the desired output, then the weights need to be changed to reduce the error.

Perceptron Weights Adjustment

$$\Delta w = \eta \times d \times x$$

$d \rightarrow$ Predicted output - Desired output

$\eta \rightarrow$ Learning rate, usually less than 1

$x \rightarrow$ Input data

Because SLP is a linear classifier and if the cases are not linearly separable the learning process will never reach a po where all the cases are classified properly. The most famous example of the inability of perceptron to solve problem with linearly non-separable cases is the XOR problem.

However, a multi-layer perceptron using the backpropagation algorithm can successfully classify the XOR data.

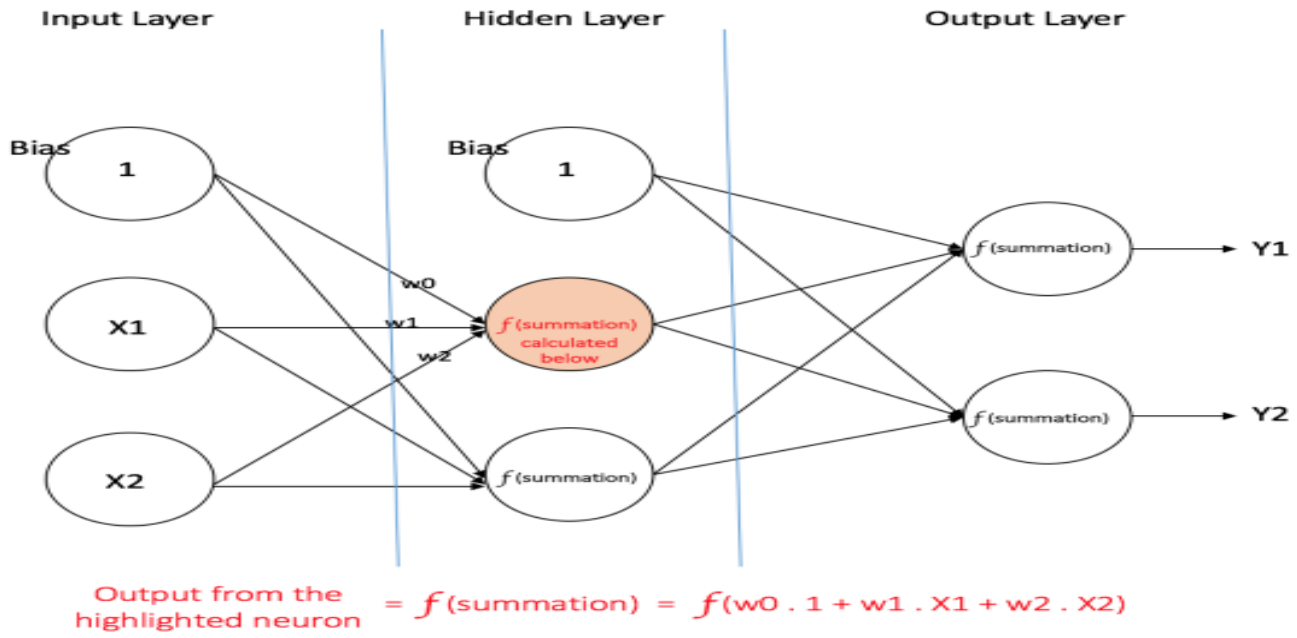
Multi layer perceptron

A Multi Layer Perceptron (MLP) contains one or more hidden layers (apart from one input and one output layer). While a single layer perceptron can only learn linear functions, a multi layer perceptron can also learn non – linear functions.

Figure 4 shows a multi layer perceptron with a single hidden layer. Note that all connections have weights associated with them, but only three weights (w0, w1, w2) are shown in the figure.

Input Layer: The Input layer has three nodes. The Bias node has a value of 1. The other two nodes take X1 and X2 as external inputs (which are numerical values depending upon the input dataset). As discussed above, no computation is performed in the Input layer, so the outputs from nodes in the Input layer are 1, X1 and X2 respectively, which are fed into the Hidden Layer.

Hidden Layer: The Hidden layer also has three nodes with the Bias node having an output of 1. The output of the other two nodes in the Hidden layer depends on the outputs from the Input layer (1, X1, X2) as well as the weights associated with the connections (edges). Figure 4 shows the output calculation for one of the hidden nodes (highlighted). Similarly, the output from other hidden node can be calculated. Remember that ***f*** refers to the activation function. These outputs are then fed to the nodes in the Output layer.



Output Layer: The Output layer has two nodes which take inputs from the Hidden layer and perform similar computations as shown for the highlighted hidden node. The values calculated (Y1 and Y2) as a result of these computations act as outputs of the Multi Layer Perceptron.

Given a set of features **X = (x1, x2, ...)** and a target **y**, a Multi Layer Perceptron can learn the relationship between the features and the target, for either classification or regression.

Lets take an example to understand Multi Layer Perceptrons better. Suppose we have the following student-marks dataset:

Hours Studied	Mid Term Marks	Final Term Result
35	67	1 (Pass)
12	75	0 (Fail)
16	89	1 (Pass)
45	56	1 (Pass)
10	90	0 (Fail)

The two input columns show the number of hours the student has studied and the mid term marks obtained by the student. The Final Result column can have two values 1 or 0 indicating whether the student passed in

the final term. For example, we can see that if the student studied 35 hours and had obtained 67 marks in the mid term, he / she ended up passing the final term.

Now, suppose, we want to predict whether a student studying 25 hours and having 70 marks in the mid term will pass the final term.

Hours Studied	Mid Term Marks	Final Term Result
25	70	?

This is a binary classification problem where a multi layer perceptron can learn from the given examples (training data) and make an informed prediction given a new data point. We will see below how a multi layer perceptron learns such relationships.

Back-propogation algorithm

The process by which a Multi Layer Perceptron learns is called the Backpropagation algorithm.

Backward Propagation of Errors, often abbreviated as BackProp is one of the several ways in which an artificial neural network (ANN) can be trained. It is a supervised training scheme, which means, it learns from labeled training data (there is a supervisor, to guide its learning).

To put in simple terms, BackProp is like "**learning from mistakes**". The supervisor ***corrects***the ANN whenever it makes mistakes.

An ANN consists of nodes in different layers; input layer, intermediate hidden layer(s) and the output layer. The connections between nodes of adjacent layers have "weights" associated with them. The goal of learning is to assign correct weights for these edges. Given an input vector, these weights determine what the output vector is.

In supervised learning, the training set is labeled. This means, for some given inputs, we know the desired/expected output (label).

The basic procedure:

1. A training sample is presented and propagated forward through the network.
2. The output error is calculated, typically the mean squared error:
3. Network error is minimized using a method called stochastic gradient descent.

$$E = \frac{1}{2}(t - y)^2$$

Where *t* is the target value and *y* is the actual network output. Other error calculations are also acceptable, but the MSE is a good choice

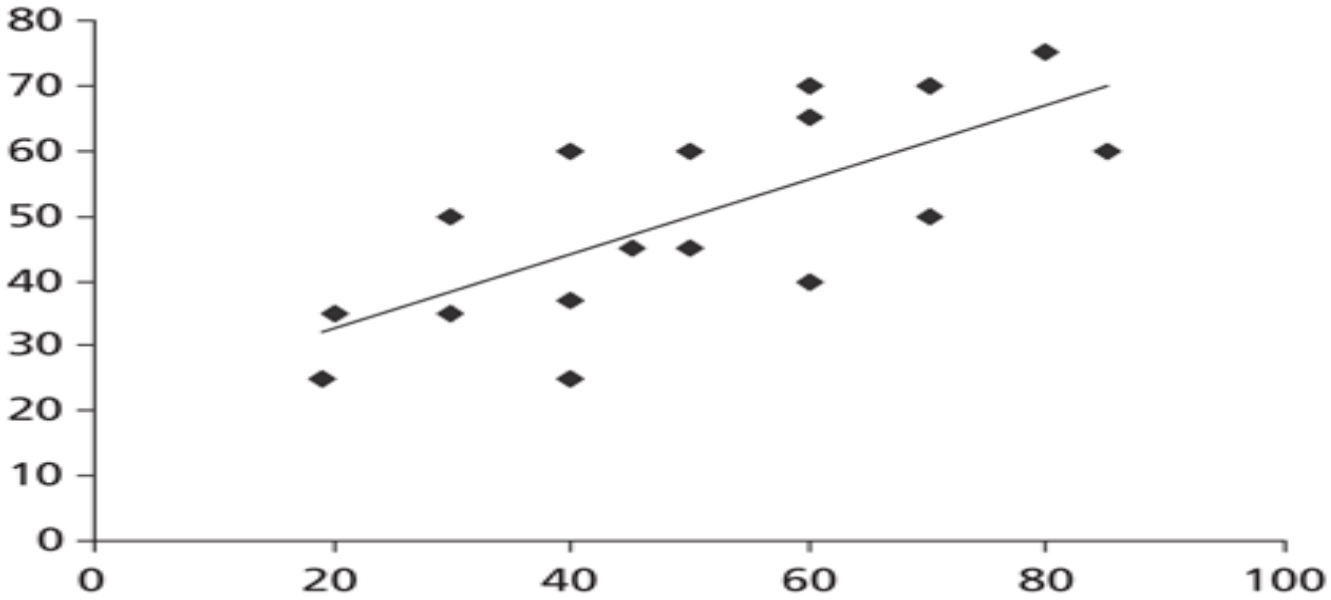
Thankfully, backpropagation provides a method for updating each weight between two neurons with respect to the output error. The derivation itself is quite complicated, but the weight update for a given node has the following (simple) form:

$$\Delta w_i = -\alpha \frac{\partial E}{\partial w_i}$$

Where *E* is the output error, and *w_i* is the weight of input *i* to the neuron.
Essentially, the goal is to move in the direction of the gradient with respect to weight *i*.

Linear regression

Linear regression is the method to find pattern with “the best fit line” in your data. Looks like something this (for any x or y axis)



In this example i will be using **load_boston** dataset. İt is the boston housing pricing dataset from scikit learn.

Full code ☺

```
import pandas as pd
import pprint
from sklearn.datasets import load_boston
from sklearn import linear_model
from sklearn.model_selection import train_test_split

boston = load_boston()
# pprint.pprint(boston)

df_x = pd.DataFrame(boston.data, columns = boston.feature_names)
df_y = pd.DataFrame(boston.target)

model = linear_model.LinearRegression()

x_train, x_test, y_train, y_test = train_test_split(df_x, df_y, test_size = 0.2, random_state = 4 )
model.fit(x_train,y_train)

result=model.predict(x_test)
print(result[5] , y_test )

```

The above code is divided into 5 steps.

1. **Load** and **preety print** the dataset.

```
from sklearn.datasets import load_boston
boston = load_boston()

import pprint
pprint.pprint(boston)
```

from sklearn.datasets **import** load_boston

load = load_boston()

First line above imports the *load_boston* dataset from the *datasets* collection which is present in the *sklearn* library.Second line just puts the *load_boston* in the *load* variable for further use.

import pprint

pprint.pprint(boston)

Importing the *preetprint* library, its used to make our output look preety and understandable and then we are just preety printing it.

2. Create the **DataFrame** .

```
import pandas as pd

df_x = pd.DataFrame(boston.data, columns = boston.feature_names)
df_y = pd.DataFrame(boston.target)
```

In this we will create two *DataFrame* using the *pandas* library*. Dataframe* is just a fancy word for making the sets or arrays of data. The *df_x* contains the data or features of the houses with the *columns = boston. features_names* which is also an array present in the dataset for more clear understanding and the *df_y* contains the target prices respectively**.**

3. Selecting the **Model** (i.e. **LinearRegression**)

```
from sklearn import linear_model

model = linear_model.LinearRegression()
```

we are importing the *LinearRegression()* model from the sklearn library and giving the model to the variable *model* for further use in the code.

4. Splitting **Test** and **Train** Datasets with **randomness.**

```
from sklearn.model_selection import train_test_split

x_train, x_test, y_train, y_test = train_test_split(df_x, df_y, test_size = 0.2, random_state = 4 )
```

Here, we are using the *train_test_split* from the sklearn library. This function in above code requires four parameters the *data* DataFrame, *target* DataFrame, test_size, random_state (Just to shuffle the data).

Note: In the above line train_size is automatically given as 0.8 since test_size =0.2 therefore, test_size + train_size = 0.2+0.8 = 1

5.Train and Predict

```
model.fit(x_train,y_train)

result = model.predict(x_test)

print(result[5] , y_test )
```

In the above code, *fit (aka TRAINING)*function takes two parameters *x_train, y_train and train the model with the given datasets.*

Next, we will *predict* the *y_test* from the *x_test* dataset using the selected *model* (i.e. LinearRegression()) and put the pedeicted array of values in the *result* variable. and then we are just printing the fifth predicted value (i.e. result[5]) and the full *y_test* dataset. You should the **5** from something else just remember the counting starts from 0 not 1 in the list.

The Result

```
[ 25.43573299]          0
8     16.5
289   24.8
68    17.4
211   19.3
226   37.6
70    24.2
55    35.4
470   19.9
409   27.5
154   17.0
344   31.2
272   24.4
```

Convolutional neural networks

ConvNets are good at finding patterns, which is to capture local “spatial”(things that next to one another) patterns of data, so it’s best to apply on image because the patterns(Pattern in the images mean that the position of the data matters to us)in the image are the most common one, and if the data fails to be made to look like an image, CNN will be less useful

Convolutional neural networks are separated into 2 parts

- Feature learning
- classification

Images are a matrix of pixel values.
Channel is a conventional term used to refer to a certain component of an image. An image from a standard digital camera will have three channels — red, green and blue(RGB) — you can imagine those as three 2d-matrices stacked over each other (one for each color), each having pixel values in the range 0 to 255.While a grayscale image, has just one channel. The value of each pixel in the matrix will range from 0 to 255 — zero indicating black and 255 indicating white.

Feature learning

1.convolutional part

1	1	1	0	0
0	1	1	1	0
0	0	1	1	1
0	0	1	1	0
0	1	1	0	0

1	0	1
0	1	0
1	0	1

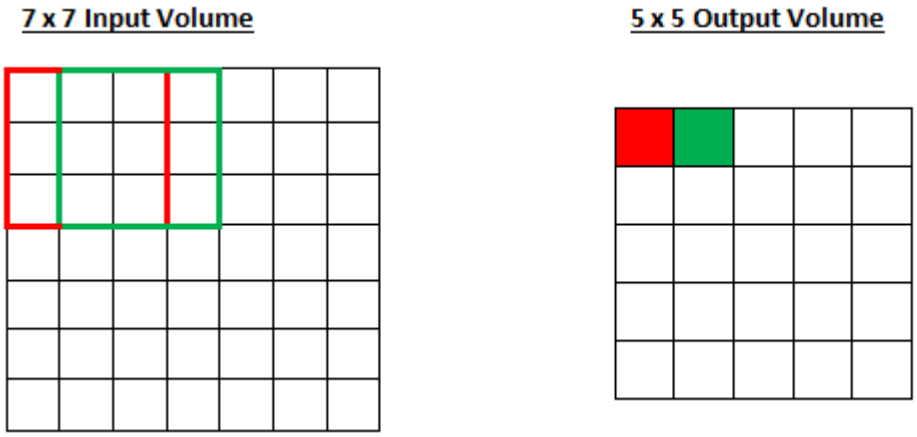
Think of this 2 image as a matrix with number, 0 act as white and 1 act as black.

Convolved feature also called feature maps/activation map. This is the process of convolving, which take the **filter** to do element-wise multiplication with the matrix pixel of the image. After convolving, the convolved feature which is the output of the previous layer, will be the input for the next layer. If the pixel in the image match the pixel of the filter, the pixel in convolved feature will be very high, if there's nothing related to the **filter** in the image, then the pixel will be very low in the convolved feature, which is the next layer.

Time takes to run the filter = ((height*width)of image)/stride
Parameters that control the behavior of each convolved layer.

- Stride
- Padding (Zero-padding)
- Number of filter (depth of next layer)
- Size of the filter

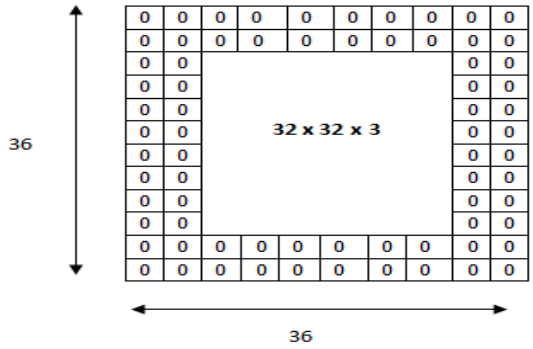
Stride
The amount of filter shift in the image. The bigger the stride, the smaller the feature map output.



Value of stride is 1, with filter 3x3 on 7x7 image.

Padding(Zero padding)

İn the early stage,we want to preserve as much information of an image as possible,so we can extract low level features.Padding can preserve the dimentions of an images well,or in another name---"Same convolutional"



The input volume is 32 x 32 x 3. If we imagine two borders of zeros around the volume, this gives us a 36 x 36 x 3 volume. Then, when we apply our conv layer with our three 5 x 5 x 3 filters and a stride of 1, then we will also get a 32 x 32 x3 output volume.

Adding zero-padding is also called wide convolution, and not using zero-padding would be a narrow convolution.

Same convolutional

Zero padding=(Filter size-1)/2

(Get the output (n+2p-f+1) by setting zero padding to be p=(f-1)/2 when the stride is 1 ensures that the input volume and output volume will have the same size spationally)

Output size of convolutional features

Output(height/width)=((input(height/width)-filterSize+2*(Zero padding))/stride+1

Constraints on strides. *Note again that the spatial arrangement hyperparameters have mutual constraints. For example, when the input has size W=10, no zero-padding is used P=0, and the filter size is F=3, then it would be impossible to use stride S=2, since (W−F+2P)/S+1=(10−3+0)/2+1=4.5, i.e.* ***not an integer, indicating that the neurons don't "fit" neatly and symmetrically across the input.*** *Therefore, this setting of the hyperparameters is considered to be* ***invalid****, and a ConvNet library could throw an exception or zero pad the rest to make it fit, or crop the input to make it fit, or something. As we will see in the ConvNet architectures section, sizing the ConvNets appropriately so that all the dimensions "work out" can be a real headache, which the use of zero-padding and some design guidelines will significantly alleviate.*

Number of filter (depth of next layer)

Example: 6x6x3 image with four 3x3 filter.
After convolving, will get 4x4xn, n is depends on the number of filter you use, in another words, means that depends on the number of feature detector you use. In this case, n will be 4

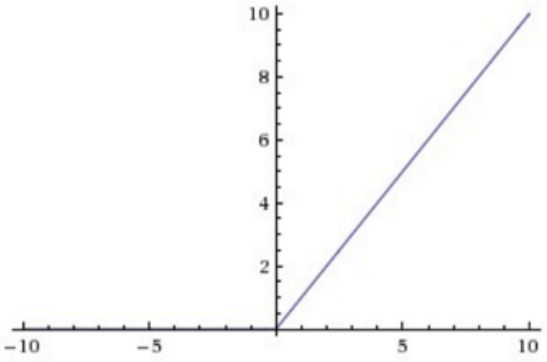
Size of the filter

Size of the filter usually is odd number so that the filter has the "central pixel"/"central vision" so to know the position of the filter. As if **size of filter** is even, then you need some asymmetric padding, or it's only **size of filter** is not that this type of same convolution gives a natural padding. We can pad the same dimension all around them, pad more on the left and pad less on the right or something asymmetric**.**

Introducing Non-Linearity, ReLU (Rectified Linear Units) to the Layers

It could be other activation function, but by far ReLU works the best.
The ReLU layer applies the function f(x) = max(0, x) to all of the values in the input volume. In basic terms, this layer just changes all the negative activations to 0**.**

Output = Max(zero, Input)



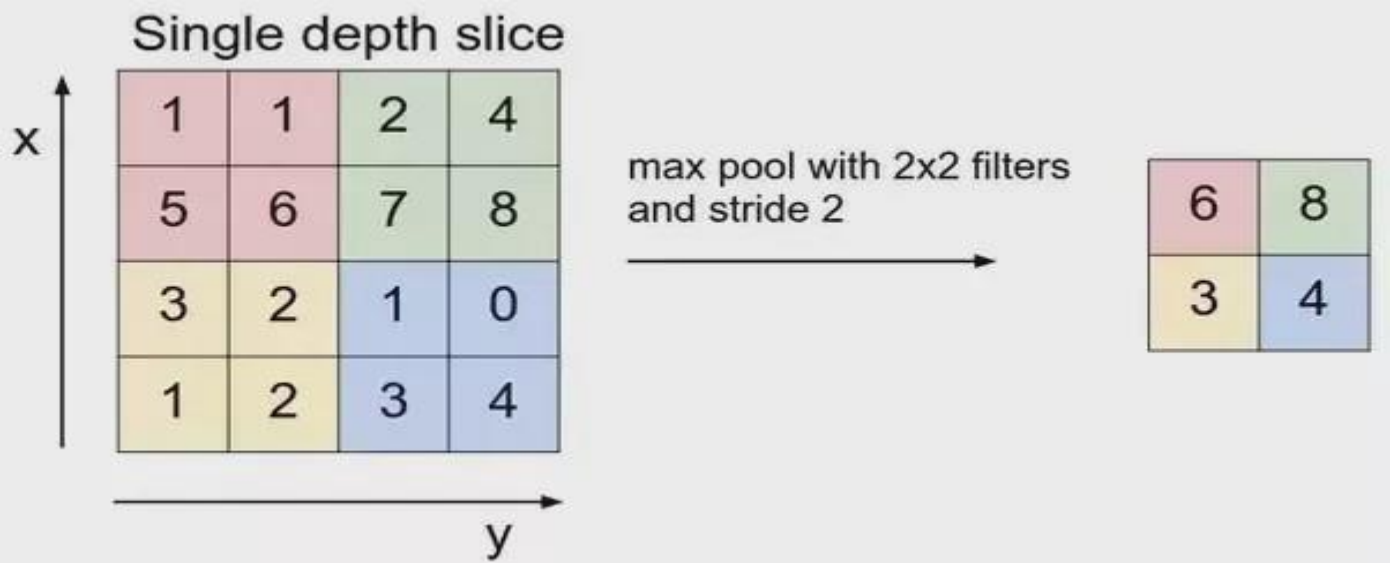
The reason that we apply non-linearities to the function is that Convolution is a linear operation — element wise matrix multiplication and addition, so we account for non-linearity by introducing a non-linear function like ReLU, prevent from computing the linear function, which will be a bad model.
ReLU helps in solving the vanishing gradient problem, which is a problem when we train the neural network using gradient-based algorithm, like sigmoid, it will squish all the gradient value into 0-1, then when performing the gradient descent, the gradient will be updated very small each time, and the time will take longer to complete. (Learning becomes super slow.)
Other non linear functions such as tanh or sigmoid can also be used instead of ReLU, but ReLU has been found to perform better in most situations**.**

Polling layer

Spatial Pooling (also called subsampling or downsampling) reduces the dimensionality of each feature map and retains the most important information of an image. Spatial Pooling can be of different types: Max, Average, Sum etc.

Instead of taking the largest element we could also take the average (Average Pooling) of sum of all elements in that window. In practice, Max Pooling has been shown to work better.

Max pooling being the most popular. This basically takes a filter (example: size 2x2) and a stride of the same length(which is 2). It then applies it to the input volume and outputs the maximum number in every sub-region that the filter convolves around. The output size formula for the max pooling is same as the convolution one.



Max polling usually doesn't use any padding

The function of Pooling is to progressively reduce the spatial size(the length and the width change but not the depth) of the input representation.
In particular, pooling:

- Makes the input representations (feature dimension) smaller and more manageable.
- Reduces the number of parameters and computations in the network, therefore, controlling overfitting.
- Makes the network invariant to small transformations, distortions and translations in the input image (a small distortion in input will not change the output of Pooling — since we take the maximum/average value in a local neighborhood).
- Helps us arrive at an almost scale invariant representation of our image(the exact term is "equivariant"). This is very powerful since we can detect objects in an image no matter where they are located(or no matter how they rotate in the images).

4.Dropout layer

(Not in the traditional architecture of CNN but very useful, because helps a lot in fighting overfitting)

The idea of dropout is simplistic in nature. This layer "drops out" a random set of activations in that layer by setting them to zero. Simple as that. Now, what are the benefits of such a simple and seemingly unnecessary and counterintuitive process? Well, in a way, it forces the network to be redundant. By that I mean the network should be able to provide the right classification or output for a specific example even if some of the activations are dropped out. It makes sure that the network isn't getting too "fitted" to the training data and thus helps alleviating the overfitting problem.
An important note is that this layer is only used during training, and not during test time.

Network in Network Layer (1x1 convolution)

A network in network layer refers to a conv layer where a 1x1 size filter is used.

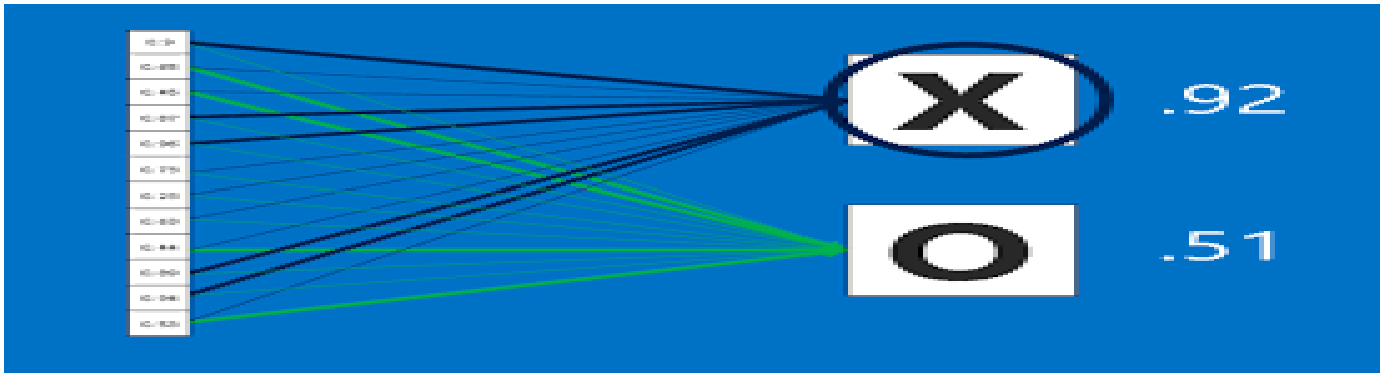
Classification

The output from the convolutional layers represents high-level features in the data. While that output could be flattened and connected to the output layer, adding a fully-connected layer is a (usually) cheap way of

learning non-linear combinations of these features.Essentially the convolutional layers are providing a meaningful, low-dimensional, and somewhat invariant feature space, and the fully-connected layer is learning a (possibly non-linear) function in that space.The whole classification part is a fully connected layer, which starts with **flattening step**, then **fully connected layer**(which is layer full of connections to do classification, also called dense layer) and end with **softmax function**.

Flattening step

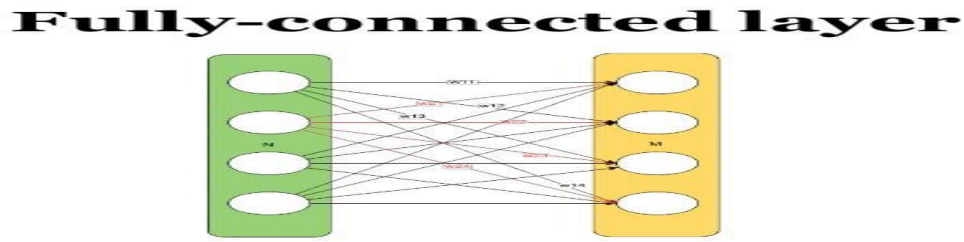
Therefore, you need to convert the output of the convolutional part of the CNN into a 1D feature vector, to be used by the ANN part of it. This operation is called flattening. It gets the output of the convolutional layers, flattens all its structure to create a single long feature vector to be used by the dense layer for the final classification.



A list of features vector which is also a list of **weights**, depending on the threshold that set early, and classify the object by using FC and softmax function.

Fully connected layer(FC)

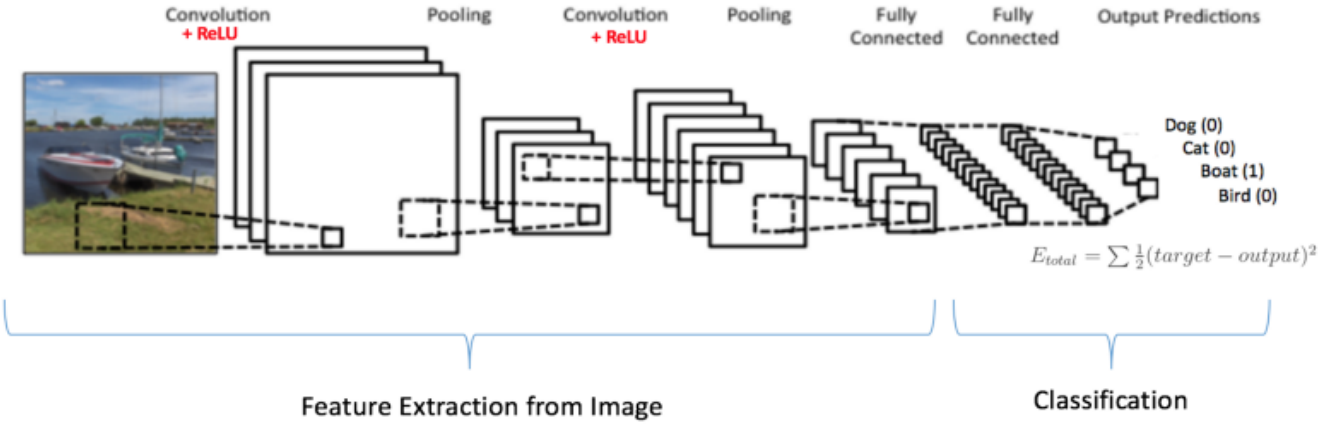
Neurons in a fully connected layer have full connections to all activations in the previous layer, as seen in regular Neural Networks. (When activations of all nodes in one layer goes to each and every node in the next layer. When all the nodes in the Lth layer connect to all the nodes in the (L+1)th layer we call these layers fully connected layers.)



Their activations can hence be computed with a matrix multiplication followed by a bias offset.
Adding a fully-connected layer is a (usually) cheap way of learning non-linear combinations of these features. Most of the features from convolutional and pooling layers may be good for the classification task, but combinations of those features might be even better.
In this layer, where the weight and bias are same as the normal neural network, use cost to compute the loss function, gradient descent to optimize parameters and reduce cost function**.**

Softmax function

The output from the Fully Connected Layer is then passed through the softmax function.
The Softmax function takes a vector of arbitrary real-valued scores and squashes it to a vector of values between zero and one that sum to one.



Overfitting and underfitting

A common danger in machine learning in **overfitting**-producing a model that performs too well on the data you train it on but that generelizes poorly to any new data. This could learning noise in data. Or it could involve learning to identify inputs rather than whatever factors are actually predictive for the desired output.

The other side of it is **underfitting**, producing a model that doesn't perform well even on the trained data,althought when this happens you decide your model isn't good enaught and keep looking for better model.

Machine learning

Machine learning algorithms use computational methods to "learn" information directly from data without relying on a predetermined equation as a model.

The algorithms adaptively improve their performance as the number of samples available for learning increases.

There are three types of ML Techniques:

1. Supervised Learning
2. Unsupervised Learning
3. Reinforced learning

Supervised learning

Finds patterns (and develops predictive models) using both, input data and output data.All Supervised Learning techniques area form of either ***Classification*** or ***Regression***

Classification

Classification is used for predicting discrete responses.

For example:

Whether India will WIN or LOSE a Cricket match?Whether an email is SPAM or GENUINE?

WIN, LOSE, SPAM, GENUINE are the predefined classes. And output has to fall among these depending on the input.

Regression

Regression is used for predicting continuous responses.

For example:

Trend in stock market prices, Weather forecast, etc.

Unsupervised Learning

Finds patterns based only on input data. This technique is useful when you're not quite sure what to look for. Often used for exploratory Analysis of raw data.Most Unsupervised Learning techniques are a form of Cluster Analysis.

Cluster Analysis

In Cluster Analysis, you group data items that have some measure of similarity based on characteristic values.

At the end what you will have is a set of different groups (Let's assume A — Z such groups). A Data Item(d1) in one group(A) is very much similar to other Data Items(d2 — dx) in the same group(A), but d1 is significantly different from Data Items belonging to different groups (B — Z).

Reinforced learning

In **reinforcement learning (RL)** there's no answer key, but your reinforcement learning **agent** still has to decide how to act to perform its task. In the absence of existing training data, the agent learns from experience. It collects the training examples ("this action was good, that action was bad") through **trial-and-error** as it attempts its task, with the goal of maximizing long-term **reward**.

MNIST dataset

The **MNIST database** (Modified National instute of standart and technology database) is a large database of handwritten digits that is commonly used for training various image prosessing systems.The database is also widely used for training and testing in the field of machine learning. It was created by "re-mixing" the samples from NIST's original database's. The creators felt that since NIST's training dataset was taken from American Census Nureau employees, while the testing dataset was taken from American high school students, it was not well-suited for machine learning experiments. Furthermore, the black and white images from NIST were normalized to fit into a 28x28 pixel bounding box and anti-aliased, which introduced grayscale levels.

Sample images from MNIST test dataset.

The MNIST database contains 60,000 training images and 10,000 testing images. Half of the training set and half of the test set were taken from NIST's training dataset, while the other half of the training set and the other half of the test set were taken from NIST's testing dataset. There have been a number of scientific papers on attempts to achieve the lowest error rate; one paper, using a hierarchical system of convolutional neural networks , manages to get an error rate on the MNIST database of 0.23%. The original creators of the database keep a list of some of the methods tested on it. In their original paper, they use a support vector machine to get an error rate of 0.8%. An extended dataset similar to MNIST called EMNIST has been published in 2017, which contains 240,000 training images, and 40,000 testing images of handwritten digits and characters.

CIFAR-10

he **CIFAR-10 dataset** (canadian institute for advanced research) is a collection of images that are commonly used to train machine learning and computer vision algorithms. It is one of the most widely used datasets for machine learning research.The CIFAR-10 dataset contains 60,000 32x32 color images in 10 different classes.The 10 different classes represent airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks. There are 6,000 images of each class.

Computer algorithms for recognizing objects in photos often learn by example. CIFAR-10 is a set of images that can be used to teach a computer how to recognize objects. Since the images in CIFAR-10 are low-resolution (32x32), this dataset can allow researchers to quickly try different algorithms to see what works. Various kinds of convolutional neural networks tend to be the best at recognizing the images in CIFAR-10.

CIFAR-10 is a labeled subset of the 80 million tiny images dataset. When the dataset was created, students were paid to label all of the images.

CIFAR-100

CIFAR-100 is really similar to CIFAR-10. The difference is the number of classified label is 100. chainer.datasets.get_cifar100 method is prepared in Chainer to get CIFAR-100 dataset.The dataset structure is quite same with MNIST dataset, it is TupleDataset(A **TupleDataset** combines multiple equal-

length datasets into a single dataset of tuples. The **i**-th tuple contains the **i**-th example from each of the argument datasets, in the same order that they were supplied.)

train[i] represents i-th data, there are 50000 training data. Total train data is same size while the number of class label increased. So the training data for each class label is fewer than CIFAR-10 dataset.test data structure is same, with 10000 test data.