Experiment No-01

Title: Design of McCulloch-Pitts Neuron Model in python

Objective:

i)To understand designing of logical operations using simple McCulloch- Pitts model.

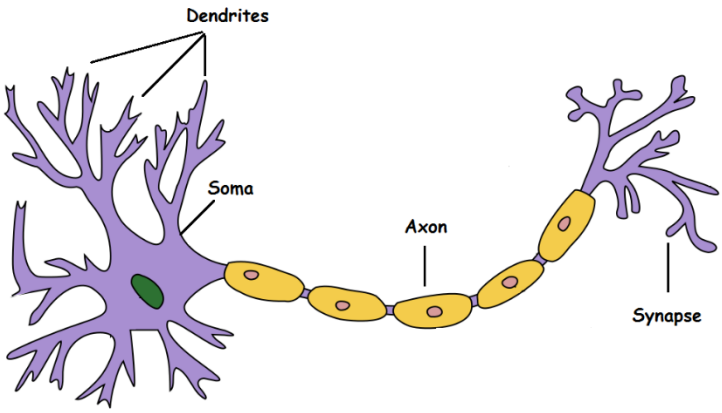
ii)To understand the concept of activation function

Tools: Spyder IDE/Google Colab/Jupyter Notebook

Theory:

It is very well known that the most fundamental unit of deep neural networks is called an artificial neuron/perceptron. But the very first step towards the *perceptron* we use today was taken in 1943 by McCulloch and Pitts, by mimicking the functionality of a biological neuron**.**

- **Biological Neurons: An Overly Simplified Illustration**

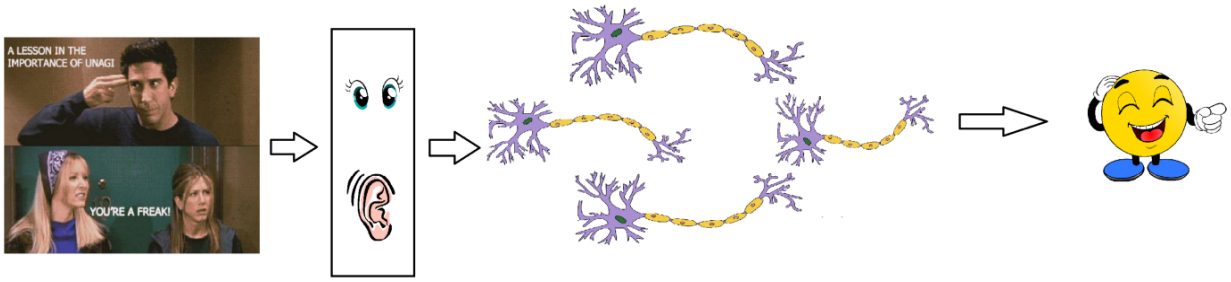


Biological Neuron

Dendrite: Receives signals from other neurons
Soma: Processes the information
Axon: Transmits the output of this neuron
Synapse: Point of connection to other neurons
Basically, a neuron takes an input signal (dendrite), processes it like the CPU (soma), passes the output through a cable like structure to other connected neurons

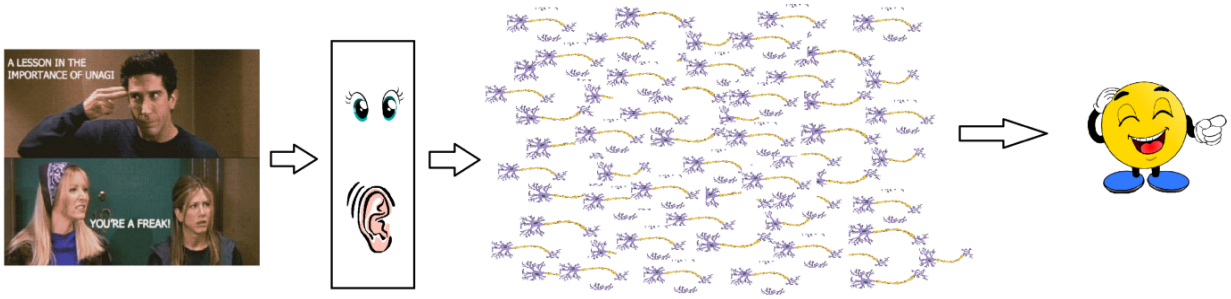
(axon to synapse to other neuron's dendrite). Now, this might be biologically inaccurate as there is a lot more going on out there but on a higher level, this is what is going on with a neuron in our brain — takes an input, processes it, throws out an output.

Our sense organs interact with the outer world and send the visual and sound information to the neurons. Let's say you are watching Friends. Now the information your brain receives is taken in by the "laugh or not" set of neurons that will help you make a decision on whether to laugh or not. Each neuron gets fired/activated only when its respective criteria (more on this later) is met like shown below.



Not real

Of course, this is not entirely true. In reality, it is not just a couple of neurons which would do the decision making. There is a massively parallel interconnected network of 10^{11} neurons (100 billion) in our brain and their connections are not as simple as I showed you above. It might look something like this:

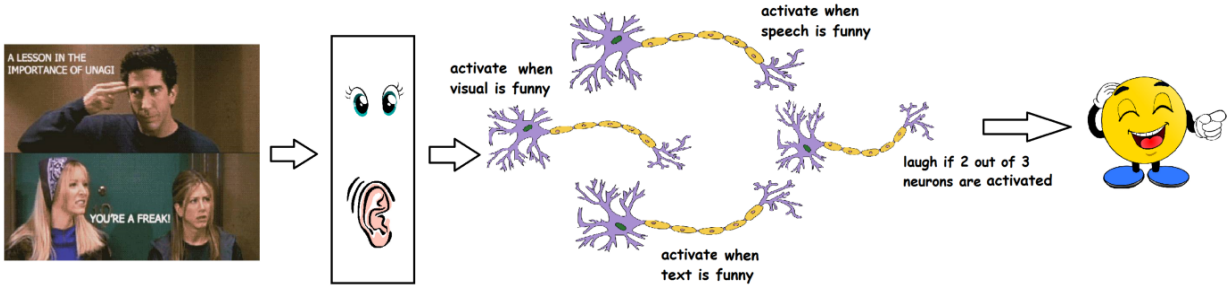


Still not real but closer

Now the sense organs pass the information to the first/lowest layer of neurons to process it. And the output of the processes is passed on to the next layers in a

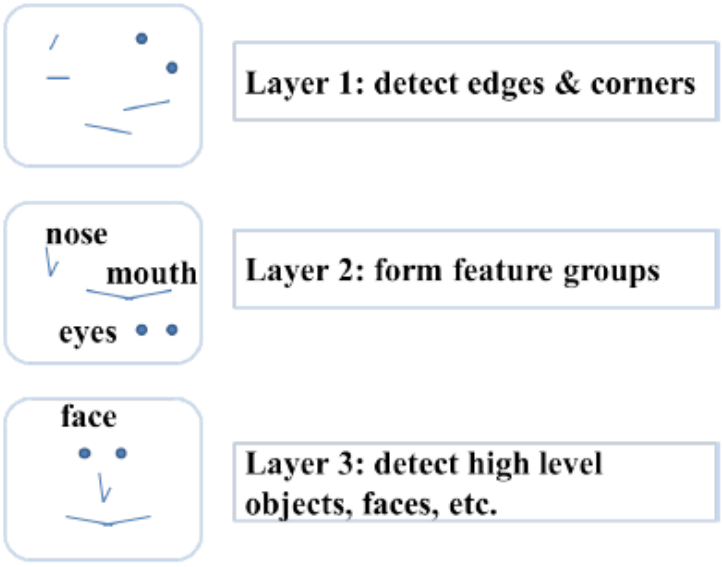
hierarchical manner, some of the neurons will fire and some won't and this process goes on until it results in a final response — in this case, laughter.

This massively parallel network also ensures that there is a division of work. Each neuron only fires when its intended criteria is met i.e., a neuron may perform a certain role to a certain stimulus, as shown below.



Division of work

It is believed that neurons are arranged in a hierarchical fashion (however, many credible alternatives with experimental support are proposed by the scientists) and each layer has its own role and responsibility. To detect a face, the brain could be relying on the entire network and not on a single layer.

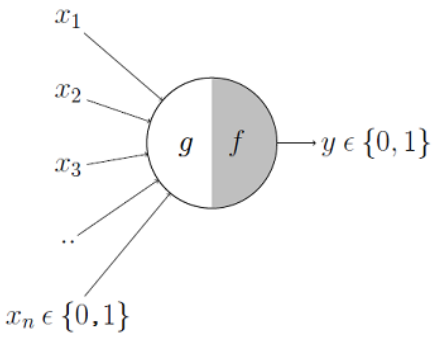


Sample illustration of hierarchical processing. Credits: Mitesh M. Khapra's lecture slides

Now that we have established how a biological neuron works, lets look at what McCulloch and Pitts had to offer.

- **McCulloch-Pitts Neuron**

The first computational model of a neuron was proposed by Warren MuCulloch (neuroscientist) and Walter Pitts (logician) in 1943.



This is where it all began..

It may be divided into 2 parts. The first part, ***g*** takes an input (ahem dendrite ahem), performs an aggregation and based on the aggregated value the second part, ***f*** makes a decision.

Lets suppose that I want to predict my own decision, whether to watch a random football game or not on TV. The inputs are all boolean i.e., {0,1} and my output variable is also boolean {0: Will watch it, 1: Won't watch it}.

- So, ***x_1*** could be *isPremierLeagueOn* (I like Premier League more)
- ***x_2*** could be *isItAFriendlyGame* (I tend to care less about the friendlies)
- ***x_3*** could be *isNotHome* (Can't watch it when I'm running errands. Can I?)
- ***x_4*** could be *isManUnitedPlaying* (I am a big Man United fan. GGMU!) and so on.

These inputs can either be *excitatory* or *inhibitory*. Inhibitory inputs are those that have maximum effect on the decision making irrespective of other inputs i.e., if ***x_3*** is 1 (not home) then my output will always be 0 i.e., the neuron will never fire, so ***x_3*** is an inhibitory input. Excitatory inputs are NOT the ones that will make the neuron fire on their own but they might fire it when combined together. Formally, this is what is going on:

$$g(x_1, x_2, x_3, ..., x_n) = g(\mathbf{x}) = \sum_{i=1}^{n} x_i$$

$$\begin{aligned} y = f(g(\mathbf{x})) &= 1 \quad if \quad g(\mathbf{x}) \geq \theta \\ &= 0 \quad if \quad g(\mathbf{x}) < \theta \end{aligned}$$

We can see that ***g*(x)** is just doing a sum of the inputs — a simple aggregation. And ***theta*** here is called thresholding parameter. For example, if I always watch the game when the sum turns out to be 2 or more, the ***theta*** is 2 here. This is called the Thresholding Logic.

Algorithm:

Step 1: generate a vector of inputs and a vector of weights. import numpy as np

Step 2: compute the dot product between the vector of inputs and weights. ...

Step 3: define the threshold activation function. ...

Step 4: compute the output based on the threshold value.

Outcome: Simple McCulloch-Pitts neurons can be used to design logical operations. For that purpose, the connection weights need to be correctly decided along with the threshold function (rather than the threshold value of the activation function).

Conclusion

In this experiment, we briefly looked at biological neurons. We then established the concept of MuCulloch-Pitts neuron, the first ever mathematical model of a biological neuron. We represented a bunch of Boolean functions using the M-P neuron. We also tried to get a geometric intuition of what is going on with the model, using 3D plots. In the end, we also established a motivation for a more generalized model, the one and only artificial neuron/perceptron model.

Experiment No-02

Title: Implementation of XOR gates using McCulloch-Pitts Artificial Neuron Model in python

Objective:

i) To understand the design of XOR gate McCulloch-Pitts Artificial Neuron Model in python

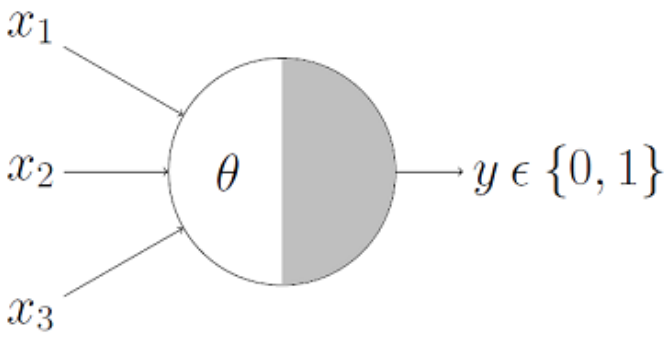
Tools: Spyder IDE/Google Colab/Jupyter Notebook

Theory:

Boolean Functions Using M-P Neuron

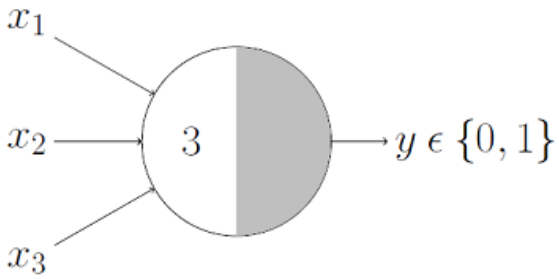
So far, we have seen how the M-P neuron works. Now let's look at how this very neuron can be used to represent a few Boolean functions. Mind you that our inputs are all boolean and the output is also boolean so essentially, the neuron is just trying to learn a boolean function. A lot of boolean decision problems can be cast into this, based on appropriate input variables— like whether to continue reading this post, whether to watch Friends after reading this post etc. can be represented by the M-P neuron.

M-P Neuron: A Concise Representation



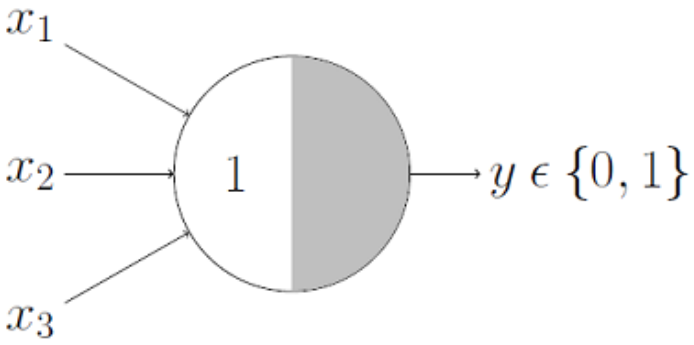
This representation just denotes that, for the boolean inputs ***x_1***, ***x_2*** and ***x_3*** if the ***g*(x)** i.e., **sum ≥ theta**, the neuron will fire otherwise, it won't.

AND Function



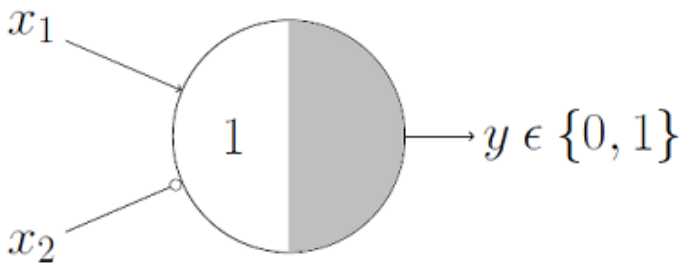
An AND function neuron would only fire when ALL the inputs are ON i.e., $\boldsymbol{g(x)} \geq 3$ here.

OR Function



I believe this is self explanatory as we know that an OR function neuron would fire if ANY of the inputs is ON i.e., $\boldsymbol{g(x)} \geq 1$ here.

A Function With An Inhibitory Input



$$x_1 \; AND \; !x_2^*$$

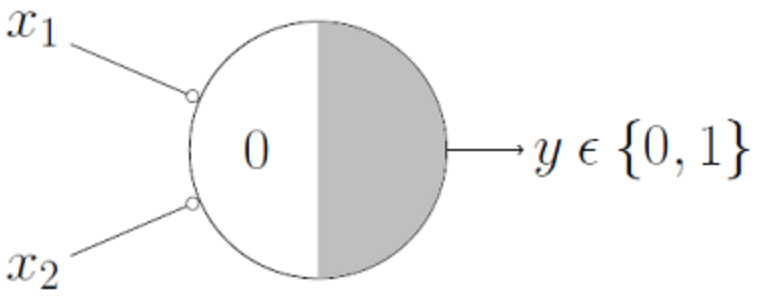
Now this might look like a tricky one but it's really not. Here, we have an inhibitory input i.e., ***x_2*** so whenever ***x_2*** is 1, the output will be 0. Keeping that in mind, we know that ***x_1 AND !x_2*** would output 1 only when ***x_1*** is 1 and ***x_2*** is 0 so it is obvious that the threshold parameter should be 1.

Lets verify that, the ***g*(x)** i.e., ***x_1*** + ***x_2*** would be ≥ 1 in only 3 cases:

Case 1: when ***x_1*** is 1 and ***x_2*** is 0
Case 2: when ***x_1*** is 1 and ***x_2*** is 1
Case 3: when ***x_1*** is 0 and ***x_2*** is 1

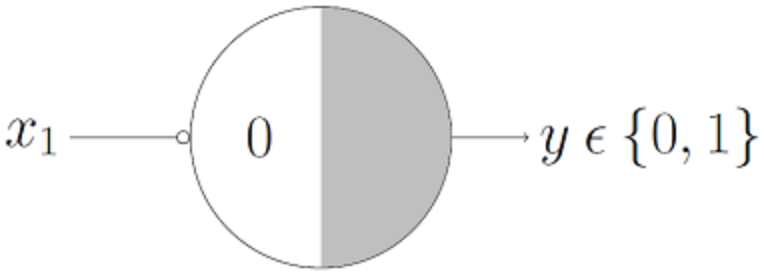
But in both Case 2 and Case 3, we know that the output will be 0 because ***x_2*** is 1 in both of them, thanks to the inhibition. And we also know that ***x_1 AND !x_2*** would output 1 for Case 1 (above) so our thresholding parameter holds good for the given function.

NOR Function



For a NOR neuron to fire, we want ALL the inputs to be 0 so the thresholding parameter should also be 0 and we take them all as inhibitory input.

NOT Function



For a NOT neuron, 1 outputs 0 and 0 outputs 1. So we take the input as an inhibitory input and set the thresholding parameter to 0. It works!

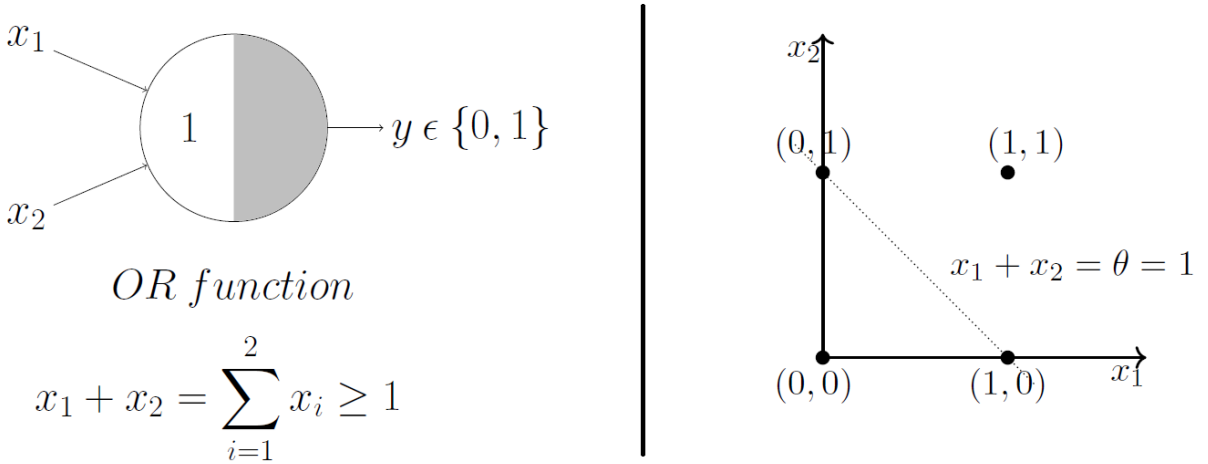
Can any boolean function be represented using the M-P neuron? Before you answer that, lets understand what M-P neuron is doing geometrically.

- **Geometric Interpretation Of M-P Neuron**

Lets start with the OR function.

OR Function

We already discussed that the OR function's thresholding parameter ***theta*** is 1, for obvious reasons. The inputs are obviously boolean, so only 4 combinations are possible — (0,0), (0,1), (1,0) and (1,1). Now plotting them on a 2D graph and making use of the OR function's aggregation equation
i.e., ***x_1 + x_2*** ≥ **1** using which we can draw the decision boundary as shown in the graph below. Mind you again, this is not a real number graph.

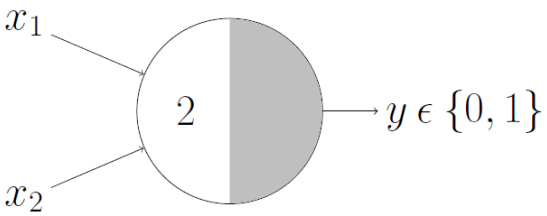


We just used the aggregation equation i.e., ***x_1 + x_2*** =**1** to graphically show that all those inputs whose output when passed through the OR function M-P neuron lie ON or ABOVE that line and all the input points that lie BELOW that line are going to output 0.

Voila!! The M-P neuron just learnt a linear decision boundary! The M-P neuron is splitting the input sets into two classes — positive and negative. Positive ones (which output 1) are those that lie ON or ABOVE the decision boundary and negative ones (which output 0) are those that lie BELOW the decision boundary.

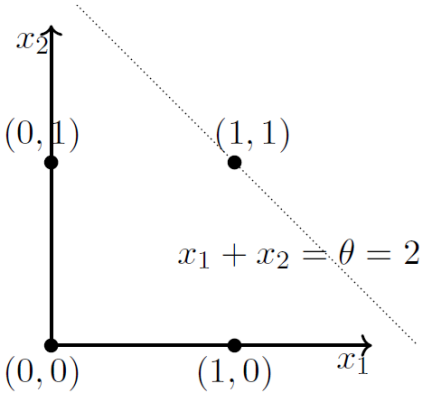
Lets convince ourselves that the M-P unit is doing the same for all the boolean functions by looking at more examples (if it is not already clear from the math).

AND Function



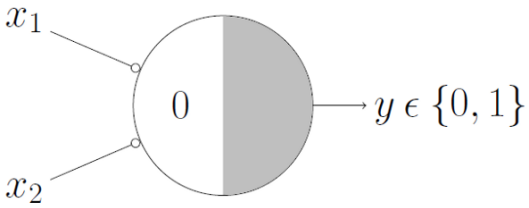
AND function

$$x_1 + x_2 = \sum_{i=1}^{2} x_i \geq 2$$

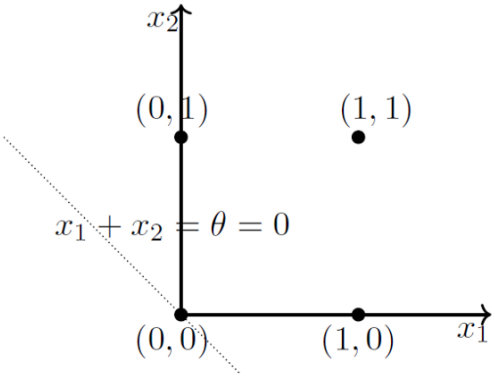


In this case, the decision boundary equation is ***x_1 + x_2* =2**. Here, all the input points that lie ON or ABOVE, just (1,1), output 1 when passed through the AND function M-P neuron. It fits! The decision boundary works!

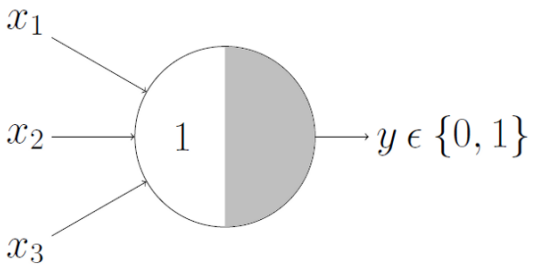
Tautology



Tautology (always ON)

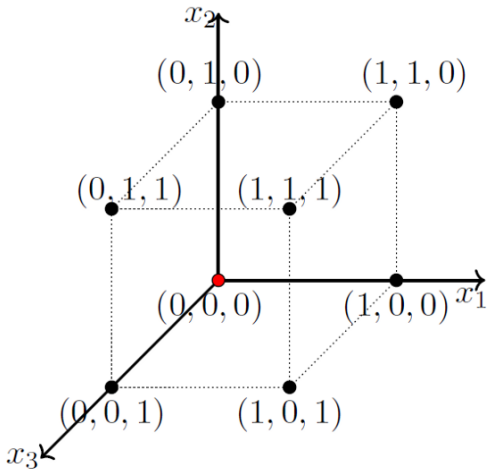


OR Function With 3 Inputs

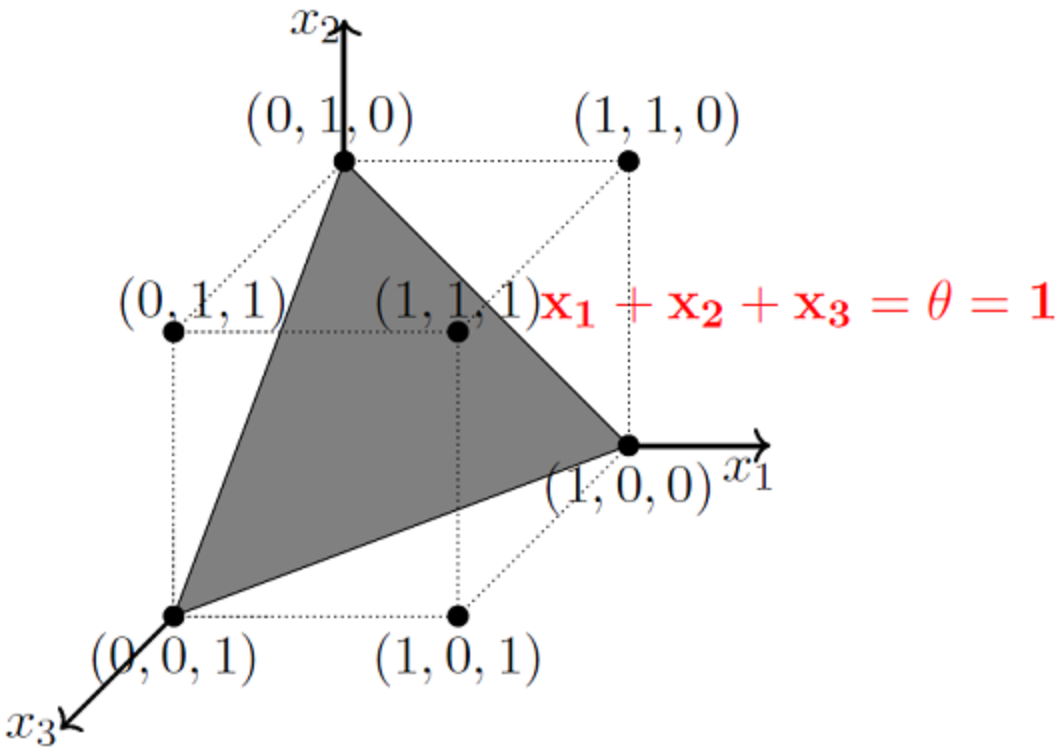


OR function

$$x_1 + x_2 + x_3 = \sum_{i=1}^{3} x_i \geq 1$$



Lets just generalize this by looking at a 3 input OR function M-P unit. In this case, the possible inputs are 8 points — (0,0,0), (0,0,1), (0,1,0), (1,0,0), (1,0,1),… you got the point(s). We can map these on a 3D graph and this time we draw a decision boundary in 3 dimensions. The plane that satisfies the decision boundary equation ***x_1 + x_2 + x_3 = 1*** is shown below:



Take your time and convince yourself by looking at the above plot that all the points that lie ON or ABOVE that plane (positive half space) will result in output 1 when passed through the OR function M-P unit and all the points that lie BELOW that plane (negative half space) will result in output 0.

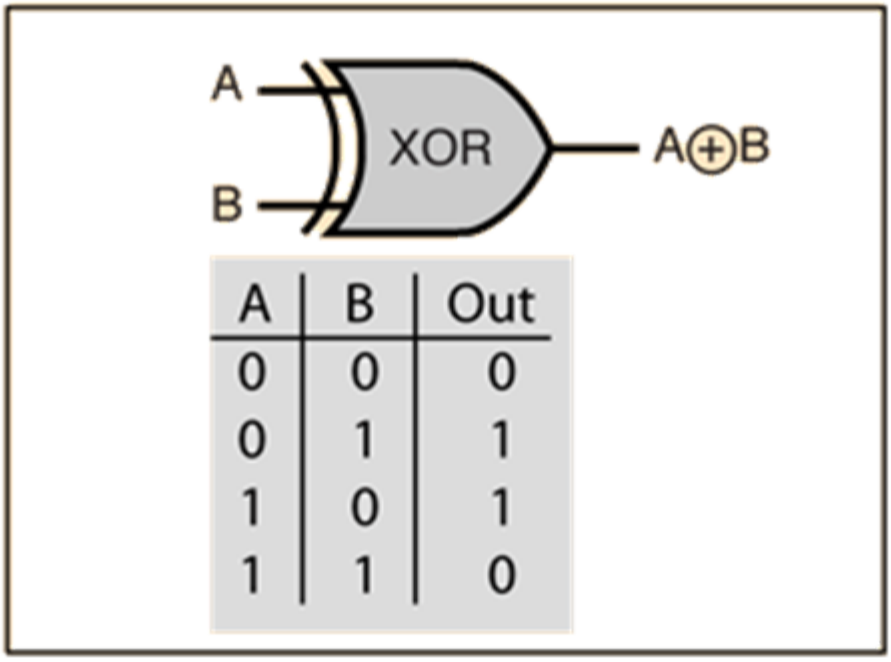
Just by hand coding a thresholding parameter, M-P neuron is able to conveniently represent the boolean functions which are linearly separable.

Limitations Of M-P Neuron

- What about non-boolean (say, real) inputs?
- Do we always need to hand code the threshold?
- Are all inputs equal? What if we want to assign more importance to some inputs?
- What about functions which are not linearly separable? Say XOR function.

I hope it is now clear why we are not using the M-P neuron today. Overcoming the limitations of the M-P neuron, Frank Rosenblatt, an American psychologist, proposed the classical perception model, the mighty *artificial neuron*, in 1958. It is more generalized computational model than the McCulloch-Pitts neuron where weights and thresholds can be learnt over time.

XOR Gate



A	B	Out
0	0	0
0	1	1
1	0	1
1	1	0

XOR Gate

The Boolean representation of an XOR gate is;

x1x`2 + x`1x2

We first simplify the Boolean expression

x`1x2 + x1x`2 + x`1x1 + x`2x2

x1(x`1 + x`2) + x2(x`1 + x`2)

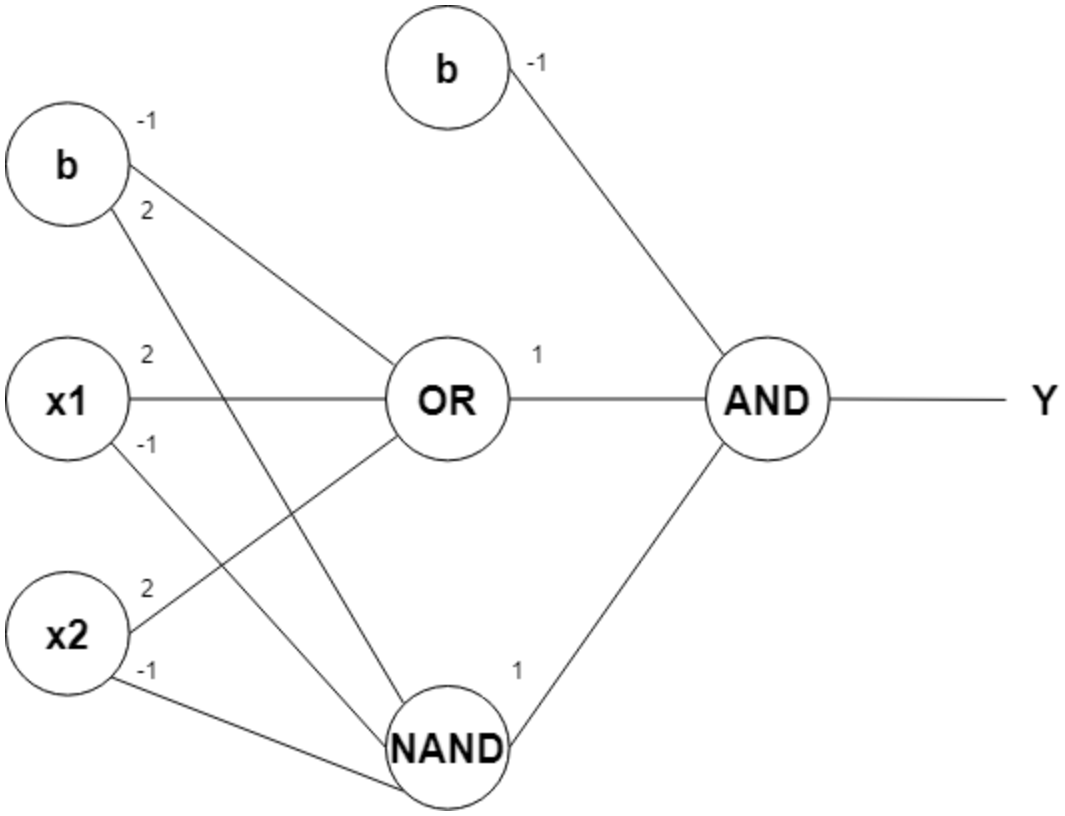
(x1 + x2)(x`1 + x`2)

(x1 + x2)(x1x2)`

From the simplified expression, we can say that the XOR gate consists of an OR gate (x1 + x2), a NAND gate (-x1-x2+1) and an AND gate (x1+x2–1.5).

This means we will have to combine 2 perceptron's:

- OR (2x1+2x2–1)
- NAND (-x1-x2+2)
- AND (x1+x2–1)



Algorithm:

Step1: Import the required Python libraries
Step2: Define Activation Function: Sigmoid Function
Step3: Initialize neural network parameters (weights, bias)
and define model hyperparameters (number of iterations, learning rate)
Step4: Forward Propagation
Step5: Backward Propagation
Step6: Update weight and bias parameters
Step7: Train the learning model
Step8: Plot Loss value vs Epoch
Step9: Test the model performance

Outcome: we got the basic idea about why and how implement the XOR operation in deep learning

Conclusion:

In conclusion, this is just a custom method of achieving this, there are many other ways and values you could use in order to achieve Logic gates using perceptron's.

Experiment No-03

Title: Write a python program to implement gradient descent algorithm

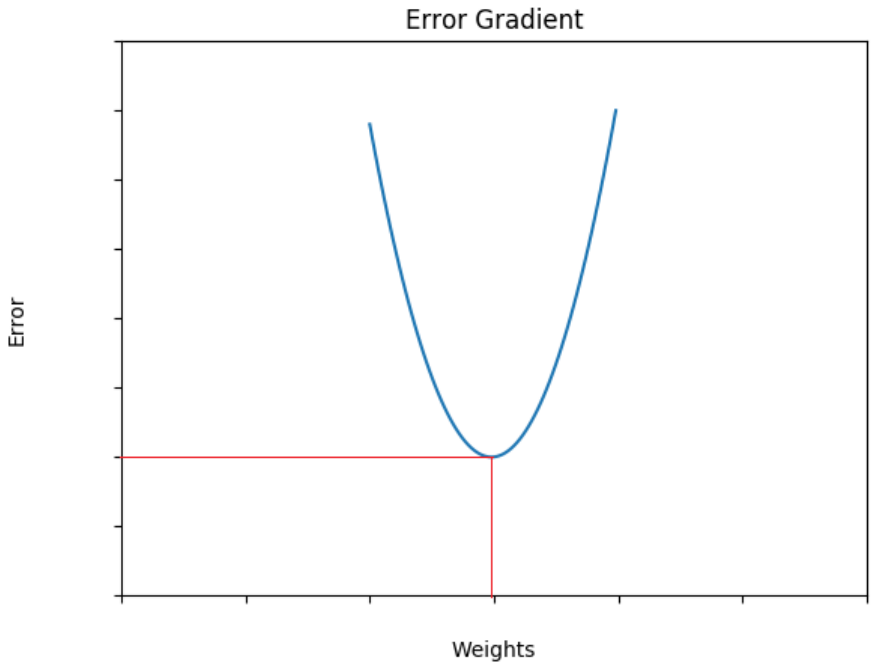
Objective: To Learn about gradient descent using artificial neural network

Tools: Spyder IDE/Google Colab/Jupyter Notebook

Theory:

- **GRADIENT DESCENT**

The loss function of the sigmoid neuron is the squared error loss. If we plot the loss/error against the weights we get something like this:



Error/Loss vs Weights Graph

Our goal is to find the weight vector corresponding to the point where the error is minimum i.e. the minima of the error gradient. And here is where calculus comes into play.

THE MATH BEHIND GRADIENT DESCENT

Error can be simply written as the difference between the predicted outcome and the actual outcome. Mathematically:

$$Error = (t - y)$$

where *t* is the targeted/expected output & *y* is the predicted output

However, is it fair to assign different error values for the same amount of error? For example, the absolute difference between -1 and 0 & 1 and 0 is the same, however the above formula would sway things negatively for the outcome that predicted -1. To solve this problem, we use square error loss.(Note modulus is not used, as it makes it harder to differentiate). Further, this error is divided by 2, to make it easier to differentiate, as we'll see in the following steps.

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

Squared Error Loss

Since, there may be many weights contributing to this error, we take the partial derivative, to find the minimum error, with respect to each weight at a time. The change in weights are different for the output layer weights (W31 & W32) and different for the hidden layer weights (W11, W12, W21, W22).
Let the outer layer weights be wo while the hidden layer weights be wh.

$$\frac{\partial Error}{\partial w}$$

We'll first find ΔW for the outer layer weights. Since the outcome is a function of activation and further activation is a function of weights, by chain rule:

$$\frac{\partial Error}{\partial w_o} = \frac{\partial Error}{\partial y_o} \times \frac{\partial y_o}{\partial a_o} \times \frac{\partial a_o}{\partial w_o}$$

On solving,

$$\frac{\partial Error}{\partial w_o} = (t - y_o) \times y_o \times (1 - y_o) \times x_o$$

Change in the outer layer weights

Note that for Xo is nothing but the output from the hidden layer nodes.
This output from the hidden layer node is again a function of the activation and correspondingly a function of weights. Hence, the chain rule expands for the hidden layer weights:

$$\frac{\partial Error}{\partial w_h} = \frac{\partial Error}{\partial y_o} \times \frac{\partial y_o}{\partial a_o} \times \frac{\partial a_o}{\partial x_o} \times \frac{\partial x_o}{\partial a_h} \times \frac{\partial a_h}{\partial w_h}$$

Which comes to,

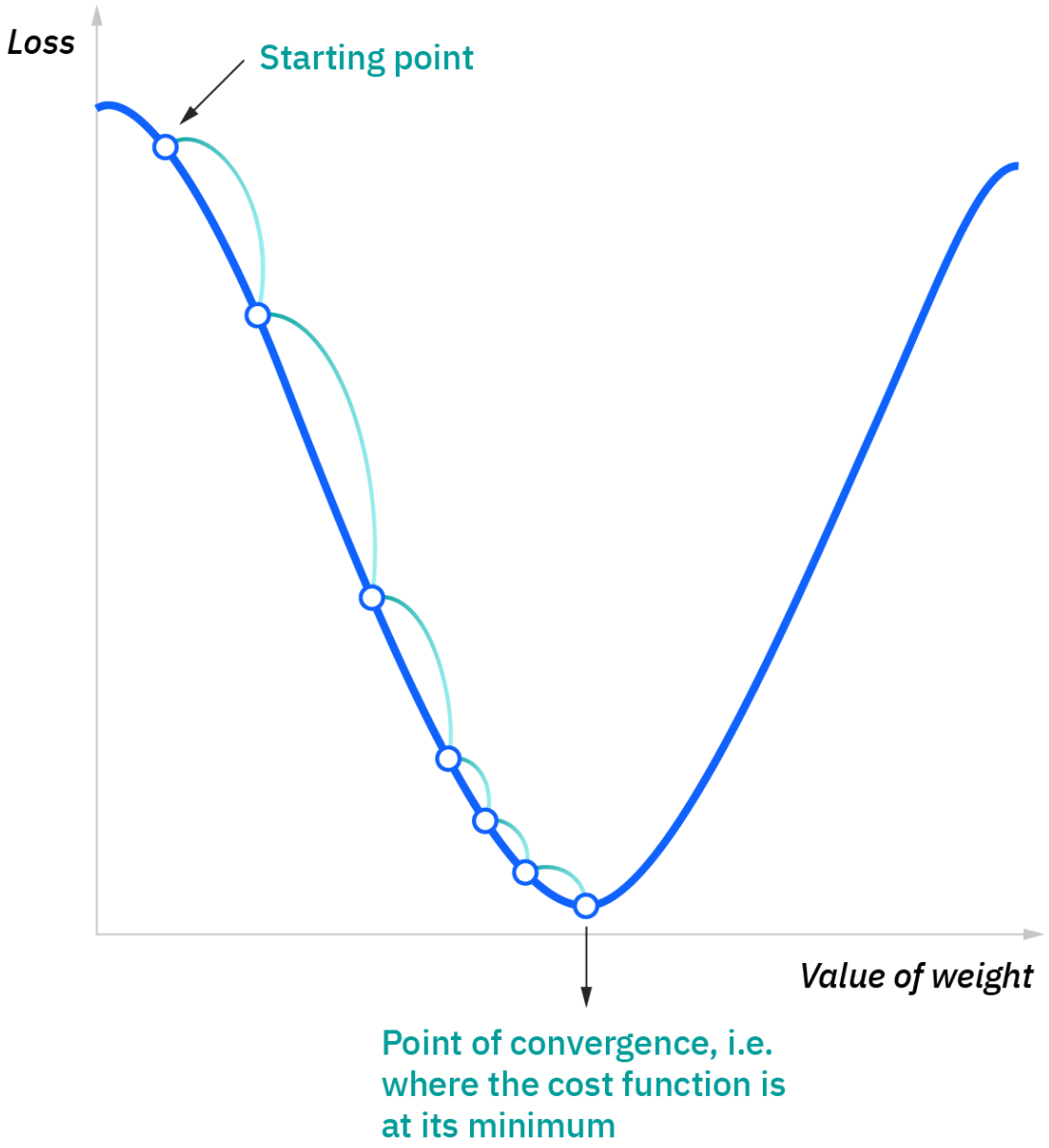
$$\frac{\partial Error}{\partial w_h} = (t - y_o) \times y_o \times (1 - y_o) \times w_o \times x_o \times (1 - x_o) \times x_h$$

Change in the hidden layer weights

How does gradient descent work?

Before we dive into gradient descent, it may help to review some concepts from linear regression. You may recall the following formula for the slope of a line, which is y = mx + b, where *m* represents the slope and *b* is the intercept on the y-axis.

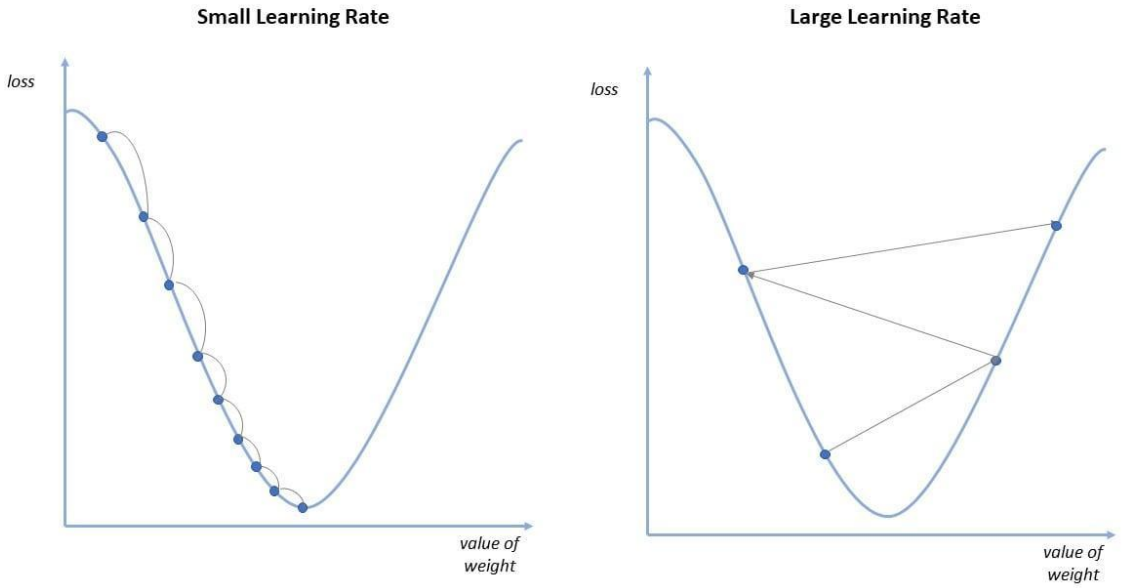
You may also recall plotting a scatterplot in statistics and finding the line of best fit, which required calculating the error between the actual output and the predicted output (y-hat) using the mean squared error formula. The gradient descent algorithm behaves similarly, but it is based on a convex function, such as the one below:



The starting point is just an arbitrary point for us to evaluate the performance. From that starting point, we will find the derivative (or slope), and from there, we can use a tangent line to observe the steepness of the slope. The slope will inform the updates to the parameters—i.e. the weights and bias. The slope at the starting point will be steeper, but as new parameters are generated, the steepness should gradually reduce until it reaches the lowest point on the curve, known as the point of convergence.

Similar to finding the line of best fit in linear regression, the goal of gradient descent is to minimize the cost function, or the error between predicted and actual y. In order to do this, it requires two data points—a direction and a learning rate. These factors determine the partial derivative calculations of future iterations, allowing it to gradually arrive at the local or global minimum (i.e. point of convergence). More detail on these components can be found below:

- **Learning rate** (also referred to as step size or the alpha) is the size of the steps that are taken to reach the minimum. This is typically a small value, and it is evaluated and updated based on the behavior of the cost function. High learning rates result in larger steps but risks overshooting the minimum. Conversely, a low learning rate has small step sizes. While it has the advantage of more precision, the number of iterations compromises overall efficiency as this takes more time and computations to reach the minimum.



- **The cost (or loss) function** measures the difference, or error, between actual y and predicted y at its current position. This improves the machine learning model's efficacy by providing feedback to the model so that it can adjust the parameters to minimize the error and find the local or global minimum. It continuously iterates, moving along the direction of steepest descent (or the negative gradient) until the cost function is close to or at

zero. At this point, the model will stop learning. Additionally, while the terms, cost function and loss function, are considered synonymous, there is a slight difference between them. It's worth noting that a loss function refers to the error of one training example, while a cost function calculates the average error across an entire training set.

Types of Gradient Descent

There are three types of gradient descent learning algorithms: batch gradient descent, stochastic gradient descent and mini-batch gradient descent.

Batch gradient descent

Batch gradient descent sums the error for each point in a training set, updating the model only after all training examples have been evaluated. This process referred to as a training epoch.

While this batching provides computation efficiency, it can still have a long processing time for large training datasets as it still needs to store all of the data into memory. Batch gradient descent also usually produces a stable error gradient and convergence, but sometimes that convergence point isn't the most ideal, finding the local minimum versus the global one.

Stochastic gradient descent

Stochastic gradient descent (SGD) runs a training epoch for each example within the dataset and it updates each training example's parameters one at a time. Since you only need to hold one training example, they are easier to store in memory. While these frequent updates can offer more detail and speed, it can result in losses in computational efficiency when compared to batch gradient descent. Its frequent updates can result in noisy gradients, but this can also be helpful in escaping the local minimum and finding the global one.

Mini-batch gradient descent

Mini-batch gradient descent combines concepts from both batch gradient descent and stochastic gradient descent. It splits the training dataset into small batch sizes and performs updates on each of those batches. This approach strikes a balance between the computational efficiency of batch gradient descent and the speed of stochastic gradient descent.

Challenges with gradient descent

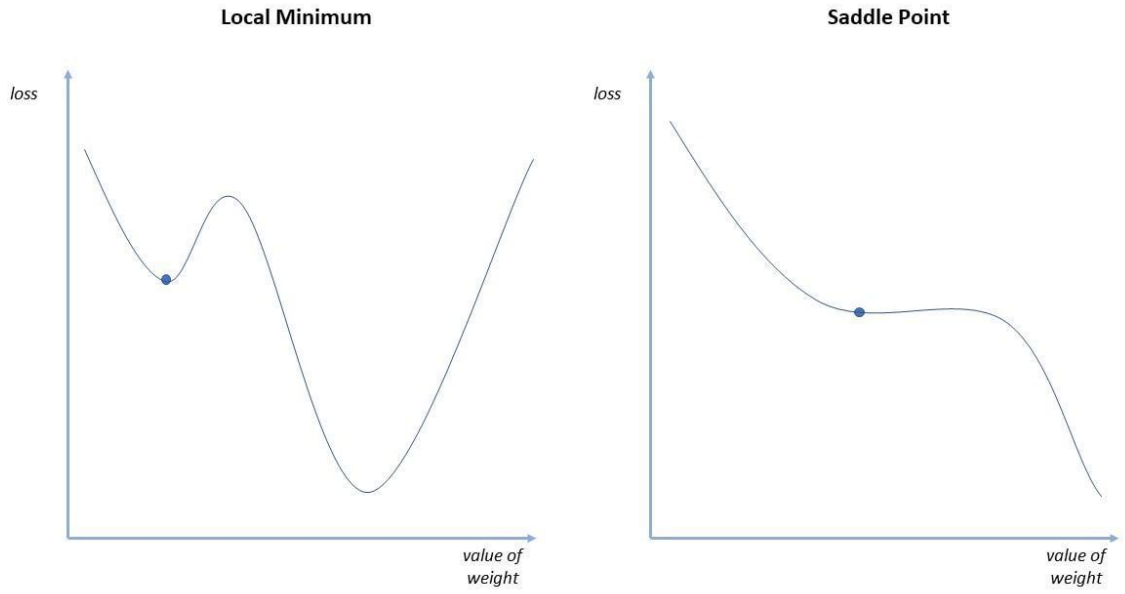
While gradient descent is the most common approach for optimization problems, it does come with its own set of challenges. Some of them include:

Local minima and saddle points

For convex problems, gradient descent can find the global minimum with ease, but as nonconvex problems emerge, gradient descent can struggle to find the global minimum, where the model achieves the best results.

Recall that when the slope of the cost function is at or close to zero, the model stops learning. A few scenarios beyond the global minimum can also yield this slope, which are local minima and saddle points. Local minima mimic the shape of a global minimum, where the slope of the cost function increases on either side of the current point. However, with saddle points, the negative gradient only exists on one side of the point, reaching a local maximum on one side and a local minimum on the other. Its name inspired by that of a horse's saddle.

Noisy gradients can help the gradient escape local minimums and saddle points.



Algorithm:

1. Initialize the weights and biases randomly.

2. Iterate over the data
   i. Compute the predicted output using the sigmoid function
   ii. Compute the loss using the square error loss function
   iii. W(new) = W(old) — α ΔW
   iv. B(new) = B(old) — α ΔB

3. Repeat until the error is minimal

Outcome:

Gradient Descent is an optimizing algorithm used in Machine/ Deep Learning algorithms. The goal of Gradient Descent is to minimize the objective convex function f(x) using iteration. Gradient Descent on Cost function.

Conclusion:

Gradient descent is an optimization algorithm which is commonly-used to train machine learning models and neural networks. Training data helps these models learn over time, and the cost function within gradient descent specifically acts as a barometer, gauging its accuracy with each iteration of parameter updates.

Experiment No-04

Title: Build a single neuron to predict the output value for given input data

Objective: TO Create model that capable to predict the value with respect to input array

Tools: Spyder IDE/Google Colab/Jupyter Notebook

Theory:

A neural network is made up of many neurons which help in computation. A single neuron has something called a weight attached to it, also called synaptic weight. These weights changes in the direction of our prediction when we train our neural network.

So, the focus of this post is, creating a neural network with a single neuron, training it for 10000 runs, predicting the output in every run, obtaining the error by comparing it with our expected output, adjusting weight based on the error and then finally try to predict the actual output.

The best part about this is, we will not be using any libraries for making our neural net (except numpy obviously). This would help us in understanding the basic structure of a neural network and how it actually works.

Without making it anymore overwhelming for you, let's start with the code. We will start by installing the library required, in this case NumPy .
Then we will import the libraries required. We will need exp for exponential (you will discover why, later), array (for array obviously), random for generating random numbers and dot for doing the dot product (right the physics one!) all from numpy.

Okay, so we will be using the following function for our program training and testing purposes as data.

Input 1	Input 2	Input 3	Output
0	0	1	0
1	1	1	1
1	0	1	0
0	1	1	1

Data

You must be wondering how the heck did he got that output? Don't you worry, no rocket science here, I just copy-pasted the entire Input 2 row of the table!

Input 1	Input 2	Input 3	Output
0	0	1	0
1	1	1	1
1	0	1	0
0	1	1	1

Input 2 = Output

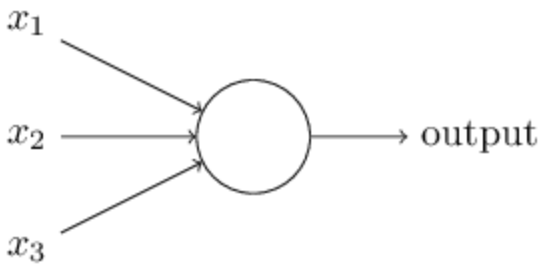
So we will be using this data set. Now, moving on, let's declare the input and output variables for our program, which would both be arrays that represent this table.

The .T used in train_outputs is used to transform the array, thereby making it a vector so that it can be multiplied easily and so that we can . tally it easily with the table . Also, train_inputs is an array of arrays, which will be used in our model. Notice, that I have exactly replicated the table above in our input and output variables.

Our next step in generation of random synaptic weights for our neuron at the starting of training. These generated weights will be adjusted as our program progresses. We will do so by using random function.

We have used seed() function here to keep the generated random weights the same, each time the program is executed, until it is closed.

Now here is a concept worth understanding. Well it's more of an application than a concept. We need to generate random numbers for a neuron with 3 input connection and 1 output connection, so our function becomes random(3,1).



A Single Neuron

Also we need to generate random numbers in the range -1 to 1 (since our output is either 0 or 1). While using the random function for range a to b , the random function is defined as : (b — a) * random_sample() + a , so here a = -1 and b = 1 and the function becomes something like this, where synaptic_weights = 2 * random.random((3, 1)) — 1.

Next, we will create the main part of our neuron, i.e the function we will use to train it. Because a neuron without training is as good as a piece of log.

Don't be intimidated by this simple function (yeah simple!), let's break it into small pieces. the train function contains 4 main blocks : the head, output predictor, error calculator and weight adjuster. It's that simple!

- **The Head**

The first part of train() function is the head. It accepts 3 arguments, namely train_inputs (the inputs for training), train_outputs (the outputs for training) and iterations (the number of time the loop will run to train the neuron, so that it's weights could be adjusted.).

There is also a for loop which runs for the number of iterations provided. Here xrange is used to specify the number of time the loop should run. The remaining 3 parts of train() function are inside this loop.

- **Output Predictor**

This part of train() is used to predict output for the given train_inputs using the current synaptic_weights. This is done by using getoutput() function.

Here we will be using sigmoid value of inputs and synaptic_weights. For combining these, we have used dot() function to perform the dot product of inputs and synaptic_weights and sending them into a sigmoid() function.

A sigmoid function is a common function used in neural networks to predict the output. What it does, is it normalizes the value of the dot product between 0 and 1(exactly the thing with need!).

It looks something like this:

$$\frac{1}{1+e^{-x}}$$

Sigmoid Function

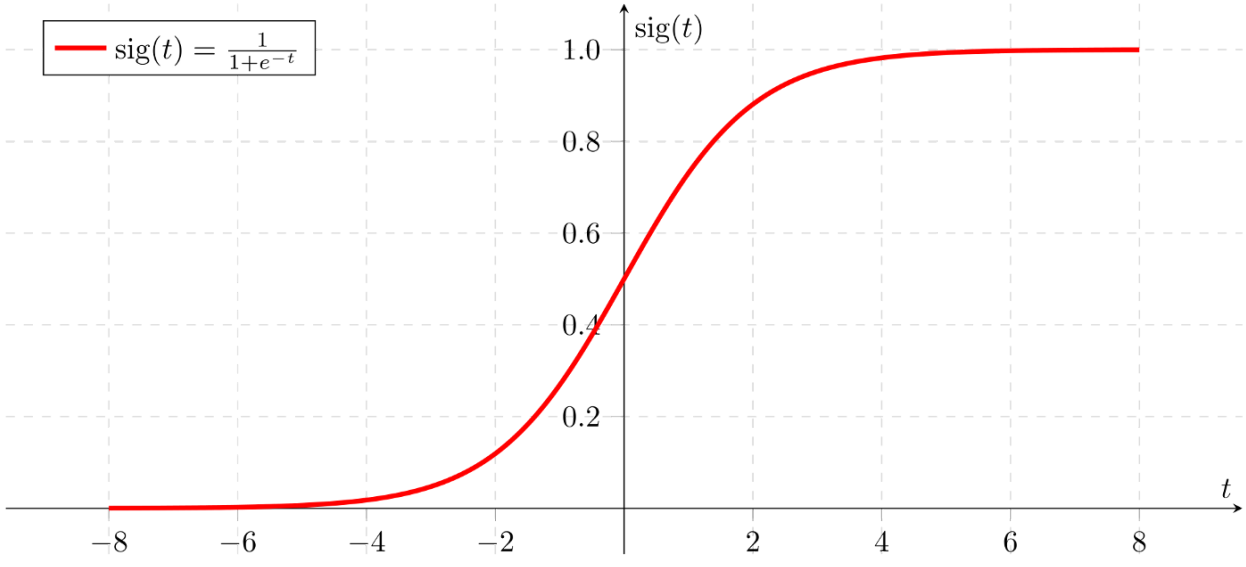
Error Calculator

The next part is calculating the error. It is simple step that subtracts the output we got in **Output Predictor** part from the actual expected output stored in the train_outputs variable.

Weight Adjuster

The last and the most important part of train() function. Here synaptic_weights are adjusted using the error obtained in **Error Calculator.**

The sigmoid curve looks something like this -



Sigmoid Curve

Notice, that the line gets straighter and constant as we move towards the edges. That means the gradient increases and the surety of getting the right prediction increases. We will also use this concept of sigmoid gradient in our program to get more accurate values.

Sigmoid Gradient = output * (1-output)

Here, first we have given our output from **Output Predictor** to sig_grad() function. What it does is, it calculates the Sigmoid Gradient of the output using the formula above so that we can move towards the edges of the sigmoid curve and get more accurate results.

Next, we will multiply our error value to sigmoid gradient of output to modify our outputs accordingly and finally, we take the dot product of these and the transpose (.T) of train_inputs to get our adjustment.

The final step of each run and train() function is to update the synaptic_weights by adding adjustment to it. We declare the variable as global synaptic_weights to access the global variable called synaptic_weights (because in python, definition and declaration are together and it would give an error if we update synaptic_weights before declaring them, also they would be present in local scope).

The rest of the program is just passing the values into train() function and then printing the results.

printed the Random Starting Synaptic Weights, and the called train() function by passing train_inputs , train_outputs and 10000 as our number of runs.

Next we printed the synaptic weights after training and the tested the neural net with a custom input of [1,1,0] , ideally we should get 1 as it is in the 2nd row and we just copied the 2nd row in the output (see the top of this post). Let's see what happens!

Observe that output is [0.9999225] , which is very close to 1 , exactly the output we wanted. Also the synaptic weights have changed and look much more consistent. We trained our model 10000 times and it can predict output so closely, and this is just one neuron, imagine the power of neural nets with 1000s of neurons!

Outcome: in this we learned the execution and mathematics behind the single neuron's calculation and prediction

Conclusion:

Hence, we understood the implementation of single neuron model using python

Algorithm:

Step 1: Define input values i.e., array with some value

Step 2: Define the class & function

Step 3: Calculate the errors & update the weights

Step 4: Repeat the [process until you didn't get good accuracy

Step 5: Observe the output

Experiment No-05

Title: Design and develop a Neural Network for classifying Breast Cancer Dataset

Objective: TO Create multilayer neurons model for classifying the cancer dataset

Tools: Spyder IDE/Google Colab/Jupyter Notebook

Theory:

- **Multi-Layer Perceptrons**

The field of artificial neural networks is often just called neural networks or multi-layer perceptrons after perhaps the most useful type of neural network. A perceptron is a single neuron model that was a precursor to larger neural networks.

It is a field that investigates how simple models of biological brains can be used to solve difficult computational tasks like the predictive modeling tasks we see in machine learning. The goal is not to create realistic models of the brain, but instead to develop robust algorithms and data structures that we can use to model difficult problems.

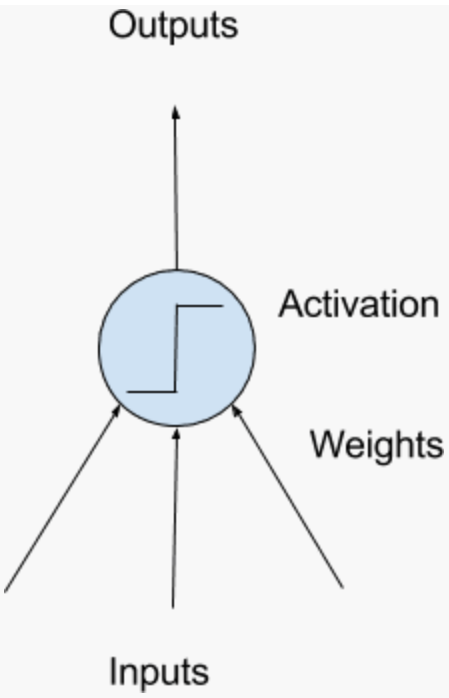
The power of neural networks comes from their ability to learn the representation in your training data and how to best relate it to the output variable that you want to predict. In this sense neural networks learn a mapping. Mathematically, they are capable of learning any mapping function and have been proven to be a universal approximation algorithm.

The predictive capability of neural networks comes from the hierarchical or multi-layered structure of the networks. The data structure can pick out (learn to represent) features at different scales or resolutions and combine them into higher-order features. For example from lines, to collections of lines to shapes.

- Neurons

The building block for neural networks are artificial neurons.

These are simple computational units that have weighted input signals and produce an output signal using an activation function.



Model of a Simple Neuron

- **Neuron Weights**

You may be familiar with linear regression, in which case the weights on the inputs are very much like the coefficients used in a regression equation.

Like linear regression, each neuron also has a bias which can be thought of as an input that always has the value 1.0 and it too must be weighted.

For example, a neuron may have two inputs in which case it requires three weights. One for each input and one for the bias.

Weights are often initialized to small random values, such as values in the range 0 to 0.3, although more complex initialization schemes can be used.

Like linear regression, larger weights indicate increased complexity and fragility. It is desirable to keep weights in the network small and regularization techniques can be used.

- **Activation**

The weighted inputs are summed and passed through an activation function, sometimes called a transfer function.

An activation function is a simple mapping of summed weighted input to the output of the neuron. It is called an activation function because it governs the threshold at which the neuron is activated and strength of the output signal.

Historically simple step activation functions were used where if the summed input was above a threshold, for example 0.5, then the neuron would output a value of 1.0, otherwise it would output a 0.0.

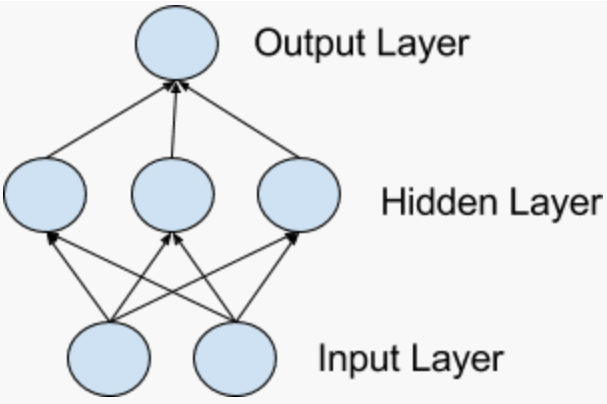
Traditionally non-linear activation functions are used. This allows the network to combine the inputs in more complex ways and in turn provide a richer capability in the functions they can model. Non-linear functions like the logistic also called the sigmoid function were used that output a value between 0 and 1 with an s-shaped distribution, and the hyperbolic tangent function also called tanh that outputs the same distribution over the range -1 to +1.

More recently the rectifier activation function has been shown to provide better results.

- **Networks of Neurons**

Neurons are arranged into networks of neurons.

A row of neurons is called a layer and one network can have multiple layers. The architecture of the neurons in the network is often called the network topology.



Model of a Simple Network

- **Input or Visible Layers**

The bottom layer that takes input from your dataset is called the visible layer, because it is the exposed part of the network. Often a neural network is drawn with

a visible layer with one neuron per input value or column in your dataset. These are not neurons as described above, but simply pass the input value though to the next layer.

- **Hidden Layers**

Layers after the input layer are called hidden layers because that are not directly exposed to the input. The simplest network structure is to have a single neuron in the hidden layer that directly outputs the value.

Given increases in computing power and efficient libraries, very deep neural networks can be constructed. Deep learning can refer to having many hidden layers in your neural network. They are deep because they would have been unimaginably slow to train historically, but may take seconds or minutes to train using modern techniques and hardware.

- **Output Layer**

The final hidden layer is called the output layer and it is responsible for outputting a value or vector of values that correspond to the format required for the problem.

The choice of activation function in he output layer is strongly constrained by the type of problem that you are modeling. For example:

- A regression problem may have a single output neuron and the neuron may have no activation function.
- A binary classification problem may have a single output neuron and use a sigmoid activation function to output a value between 0 and 1 to represent the probability of predicting a value for the class 1. This can be turned into a crisp class value by using a threshold of 0.5 and snap values less than the threshold to 0 otherwise to 1.
- A multi-class classification problem may have multiple neurons in the output layer, one for each class (e.g. three neurons for the three classes in the famous iris flowers classification problem). In this case a softmax activation function may be used to output a probability of the network predicting each of the class values. Selecting the output with the highest probability can be used to produce a crisp class classification value.

- **Training Networks**

Once configured, the neural network needs to be trained on your dataset.

- **Data Preparation**

You must first prepare your data for training on a neural network.

Data must be numerical, for example real values. If you have categorical data, such as a sex attribute with the values “male” and “female”, you can convert it to a real-valued representation called a one hot encoding. This is where one new column is added for each class value (two columns in the case of sex of male and female) and a 0 or 1 is added for each row depending on the class value for that row.
This same one hot encoding can be used on the output variable in classification problems with more than one class. This would create a binary vector from a single column that would be easy to directly compare to the output of the neuron in the network’s output layer, that as described above, would output one value for each class.

Neural networks require the input to be scaled in a consistent way. You can rescale it to the range between 0 and 1 called normalization. Another popular technique is to standardize it so that the distribution of each column has the mean of zero and the standard deviation of 1.

Scaling also applies to image pixel data. Data such as words can be converted to integers, such as the popularity rank of the word in the dataset and other encoding techniques.

- **Stochastic Gradient Descent**

The classical and still preferred training algorithm for neural networks is called stochastic gradient descent.

This is where one row of data is exposed to the network at a time as input. The network processes the input upward activating neurons as it goes to finally produce an output value. This is called a forward pass on the network. It is the type of pass that is also used after the network is trained in order to make predictions on new data.

The output of the network is compared to the expected output and an error is calculated. This error is then propagated back through the network, one layer at a time, and the weights are updated according to the amount that they contributed to the error. This clever bit of math is called the backpropagation algorithm.

The process is repeated for all of the examples in your training data. One round of updating the network for the entire training dataset is called an epoch. A network may be trained for tens, hundreds or many thousands of epochs.

- **Weight Updates**

The weights in the network can be updated from the errors calculated for each training example and this is called online learning. It can result in fast but also chaotic changes to the network.

Alternatively, the errors can be saved up across all of the training examples and the network can be updated at the end. This is called batch learning and is often more stable.

Typically, because datasets are so large and because of computational efficiencies, the size of the batch, the number of examples the network is shown before an update is often reduced to a small number, such as tens or hundreds of examples.

The amount that weights are updated is controlled by a configuration parameters called the learning rate. It is also called the step size and controls the step or change made to network weight for a given error. Often small weight sizes are used such as 0.1 or 0.01 or smaller.

The update equation can be complemented with additional configuration terms that you can set.

- Momentum is a term that incorporates the properties from the previous weight update to allow the weights to continue to change in the same direction even when there is less error being calculated.
- Learning Rate Decay is used to decrease the learning rate over epochs to allow the network to make large changes to the weights at the beginning and smaller fine tuning changes later in the training schedule.
- **Prediction**

Once a neural network has been trained it can be used to make predictions.

You can make predictions on test or validation data in order to estimate the skill of the model on unseen data. You can also deploy it operationally and use it to make predictions continuously.

The network topology and the final set of weights is all that you need to save from the model. Predictions are made by providing the input to the network and performing a forward-pass allowing it to generate an output that you can use as a prediction.

use ANN to diagnose breast cancer, and compares Multi-Layer Perceptron Neural Network and Convolutional Neural Network based on their accuracy of diagnosis and breast cancer classification.

Outcome: using MLP we learned the classification of given data successfully

Conclusion:

Hence, we learn implementation of Multilayer perceptron using python

Experiment No-06

Title: Design and develop a Deep Convolutional Neural Network for hand written digits from MNIST dataset

Objective: TO develop Convolutional network for hand written digit classification

Tools: Spyder IDE/Google Colab/Jupyter Notebook

Theory:

Convolutional neural networks are distinguished from other neural networks by their superior performance with image, speech, or audio signal inputs. They have three main types of layers, which are:

- Convolutional layer
- Pooling layer
- Fully-connected (FC) layer

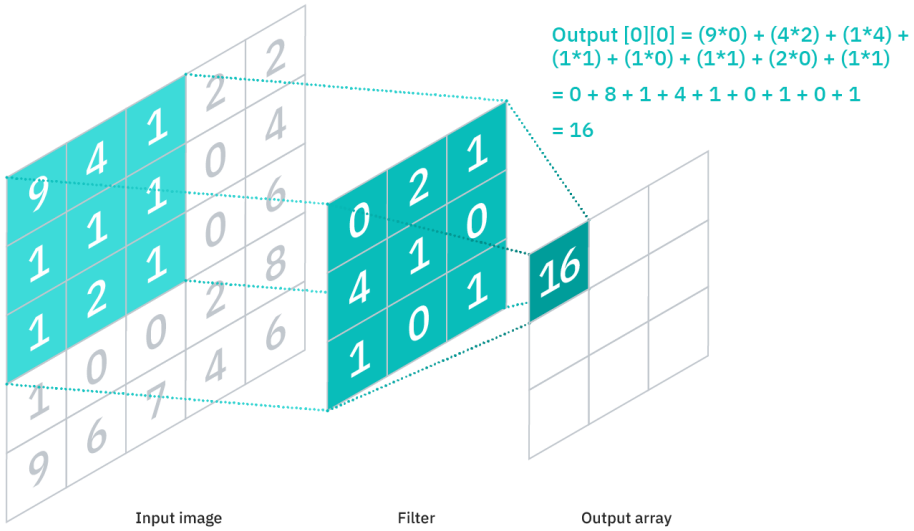
The convolutional layer is the first layer of a convolutional network. While convolutional layers can be followed by additional convolutional layers or pooling layers, the fully-connected layer is the final layer. With each layer, the CNN increases in its complexity, identifying greater portions of the image. Earlier layers focus on simple features, such as colors and edges. As the image data progresses through the layers of the CNN, it starts to recognize larger elements or shapes of the object until it finally identifies the intended object.

Convolutional Layer

The convolutional layer is the core building block of a CNN, and it is where the majority of computation occurs. It requires a few components, which are input data, a filter, and a feature map. Let's assume that the input will be a color image, which is made up of a matrix of pixels in 3D. This means that the input will have three dimensions—a height, width, and depth—which correspond to RGB in an image. We also have a feature detector, also known as a kernel or a filter, which will move across the receptive fields of the image, checking if the feature is present. This process is known as a convolution.

The feature detector is a two-dimensional (2-D) array of weights, which represents part of the image. While they can vary in size, the filter size is typically a 3x3

matrix; this also determines the size of the receptive field. The filter is then applied to an area of the image, and a dot product is calculated between the input pixels and the filter. This dot product is then fed into an output array. Afterwards, the filter shifts by a stride, repeating the process until the kernel has swept across the entire image. The final output from the series of dot products from the input and the filter is known as a feature map, activation map, or a convolved feature.



As you can see in the image above, each output value in the feature map does not have to connect to each pixel value in the input image. It only needs to connect to the receptive field, where the filter is being applied. Since the output array does not need to map directly to each input value, convolutional (and pooling) layers are commonly referred to as “partially connected” layers. However, this characteristic can also be described as local connectivity.

Note that the weights in the feature detector remain fixed as it moves across the image, which is also known as parameter sharing. Some parameters, like the weight values, adjust during training through the process of backpropagation and gradient descent. However, there are three hyperparameters which affect the volume size of the output that need to be set before the training of the neural network begins. These include:

1. The **number of filters** affects the depth of the output. For example, three distinct filters would yield three different feature maps, creating a depth of three.

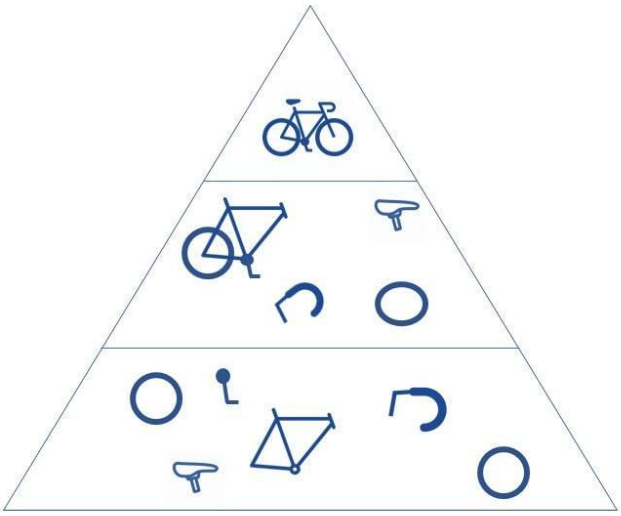
2. **Stride** is the distance, or number of pixels, that the kernel moves over the input matrix. While stride values of two or greater is rare, a larger stride yields a smaller output.

3. **Zero-padding** is usually used when the filters do not fit the input image. This sets all elements that fall outside of the input matrix to zero, producing a larger or equally sized output. There are three types of padding:

- **Valid padding:** This is also known as no padding. In this case, the last convolution is dropped if dimensions do not align.
- **Same padding:** This padding ensures that the output layer has the same size as the input layer
- **Full padding:** This type of padding increases the size of the output by adding zeros to the border of the input.

After each convolution operation, a CNN applies a Rectified Linear Unit (ReLU) transformation to the feature map, introducing nonlinearity to the model.

As we mentioned earlier, another convolution layer can follow the initial convolution layer. When this happens, the structure of the CNN can become hierarchical as the later layers can see the pixels within the receptive fields of prior layers. As an example, let's assume that we're trying to determine if an image contains a bicycle. You can think of the bicycle as a sum of parts. It is comprised of a frame, handlebars, wheels, pedals, et cetera. Each individual part of the bicycle makes up a lower-level pattern in the neural net, and the combination of its parts represents a higher-level pattern, creating a feature hierarchy within the CNN.



Ultimately, the convolutional layer converts the image into numerical values, allowing the neural network to interpret and extract relevant patterns.

Pooling Layer

Pooling layers, also known as downsampling, conducts dimensionality reduction, reducing the number of parameters in the input. Similar to the convolutional layer, the pooling operation sweeps a filter across the entire input, but the difference is that this filter does not have any weights. Instead, the kernel applies an aggregation function to the values within the receptive field, populating the output array. There are two main types of pooling:

- **Max pooling:** As the filter moves across the input, it selects the pixel with the maximum value to send to the output array. As an aside, this approach tends to be used more often compared to average pooling.
- **Average pooling:** As the filter moves across the input, it calculates the average value within the receptive field to send to the output array.

While a lot of information is lost in the pooling layer, it also has a number of benefits to the CNN. They help to reduce complexity, improve efficiency, and limit risk of overfitting.

Fully-Connected Layer

The name of the full-connected layer aptly describes itself. As mentioned earlier, the pixel values of the input image are not directly connected to the output layer in partially connected layers. However, in the fully-connected layer, each node in the output layer connects directly to a node in the previous layer.

This layer performs the task of classification based on the features extracted through the previous layers and their different filters. While convolutional and pooling layers tend to use ReLu functions, FC layers usually leverage a softmax activation function to classify inputs appropriately, producing a probability from 0 to 1.

Algorithm:

Step 1: Import the necessary libraries

Step 2: Import the MNIST dataset from TensorFlow datasets

Step 3: Split the dataset & add the convolution layers and the max pooling layer

Step 4: Compile the network and start the training the model

Step 5: Observe the accuracy and plot the epoch vs accuracy & epoch vs loss graph

Outcome: In this experiment we learned the use of convolution neural network for MNIST Dataset for digit recognition, We got idea about convolution layer and there various parameters & tunning of network

Conclusion:

Convolutional Neural Net is a popular deep learning technique for current visual recognition tasks. Like all deep learning techniques, CNN is very dependent on the size and quality of the training data. Given a well-prepared dataset, CNNs are capable of surpassing humans at visual recognition tasks.

Experiment No-07

Title: Design and develop a model for object detection using VGG16 model

Objective: To Develop and test the VGG16 architecture for detecting the object given

Tools: Spyder IDE/Google Colab/Jupyter Notebook

Theory:

- **Transfer Learning**

As humans, we find it easy to transfer knowledge we have learned from one domain or task to another. When we encounter a new task, we don't have to start from scratch. Instead, we use our previous experience to learn and adapt to that new task faster and more accurately,

For example, if you have any experience programming with Java, C/C# or any other programming languages, you're already familiar with concepts like loops, recursion, objects and etc (Figure 1, a). If you then try to pick up a new programming language like python, you don't need to learn these concepts again, you just need to learn the corresponding syntax. Or to take another example, if you have played table tennis a lot it will help you learn tennis faster as the strategies in these games are similar

a) Transferring Learned knowledge from Java to Python.

b) Transferring Learned knowledge Table Tennis to Tennis.

Transferring the learned knowledge in humans.

In recent years, fuelled by the advances in supervised and unsupervised machine learning, we have seen astonishing leaps in the application of artificial intelligence. We have reached a stage that we can build **autonomous vehicles**, **intelligent robots** and **cancer detection systems** with human-level or even super-human performance

Despite the remarkable results, these models are data hungry and their performance relies heavily on the quality and size of training data. However, in real-world scenarios, large amounts of labeled data are usually expensive to obtain or not available which means performance is low or projects are abandoned entirely. What's more, these models still lack the ability to generalize to any situation beyond those they encountered during training[2], so they are limited in what they can achieve.

Inspired by the human capability to transfer knowledge, the machine learning community has turned their focus to transfer learning to overcome these issues. Unlike the traditional machine learning paradigm where the learning process happens in isolation, without considering knowledge from any other domain (Figure 3 left side), transfer learning uses knowledge from other existing domains (source) during the learning process for a new domain (target) (Figure 3 right side).

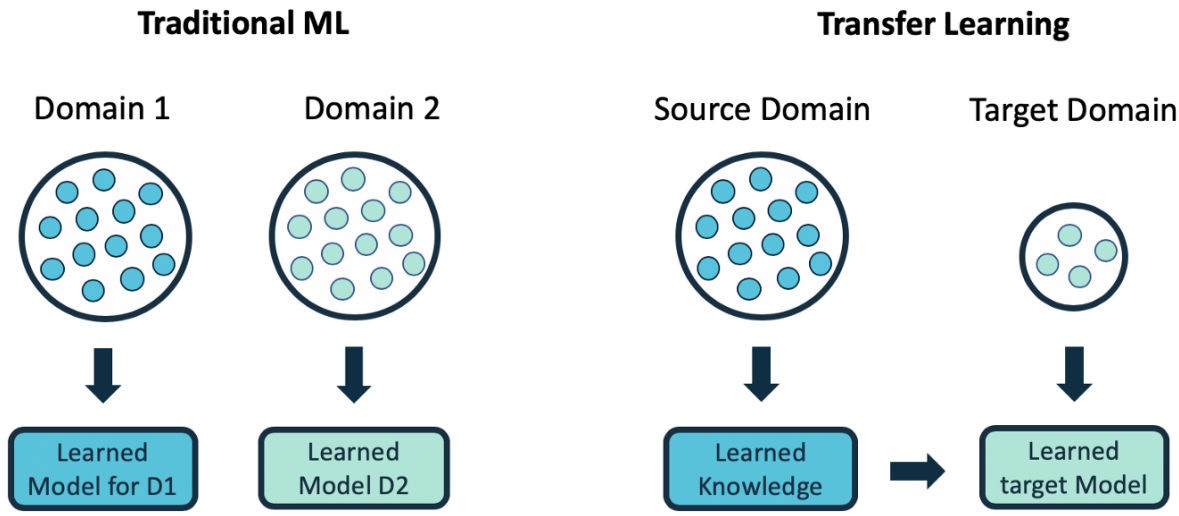


Figure 3. Let's assume that the task is to build a recommendation system. As you see on the left side, in traditional machine learning, each model is built based on a

single domain in isolation, however, in transfer learning (right side) the target model is built using the learned knowledge from the source domain.

Transfer learning addresses these three questions:

1. What information in the source is useful and transferable to target?
2. What is the best way of transferring this information?
3. How to avoid transferring information that is detrimental to the desired outcome?

The answer to these questions depends on the similarities between the feature spaces, models and tasks of the target and source domains[1]. We will provide examples below to describe these concepts.

Mathematical Notations and Definitions in Transfer Learning

In this section, we will briefly discuss the standard notations and definitions used for transfer learning in the research community[1, 3]. In the remainder of the post, we use these notations and definitions to dive deeper into more technical topics, so it's worth going a little slower through this section.

Notation

Domain: A domain $\mathfrak{D} = \{X, P(X)\}$ is defined by two components:

- A feature space X
- and a marginal probability distribution P(X) where $X=\{x_1, x_2, x_3, \ldots, x_n\} \in X$

If two domains are different, then they either have different feature spaces ($Xt \neq Xs$) or different marginal distributions ($P(Xt) \neq P(Xs)$).

Task: Given a specific domain $\mathfrak{D}$, a task $\mathcal{T}=\{Y, \boldsymbol{f}(.)\}$ consists of two parts:

- A label space Y
- and a predictive function $\boldsymbol{f}(.)$, which is not observed but can be learned from training data $\{(x_i, y_i)| \ i \in \{1, 2, 3, \ldots, N\}$, where $x_i \in X$ and $y_i \in Y \}$. From a probabilistic viewpoint $\boldsymbol{f}(x_i)$, can also be written as $p(y_i|x_i)$, so we can rewrite task $\mathcal{T}$as $\mathcal{T}=\{Y, P(Y|X)\}$.

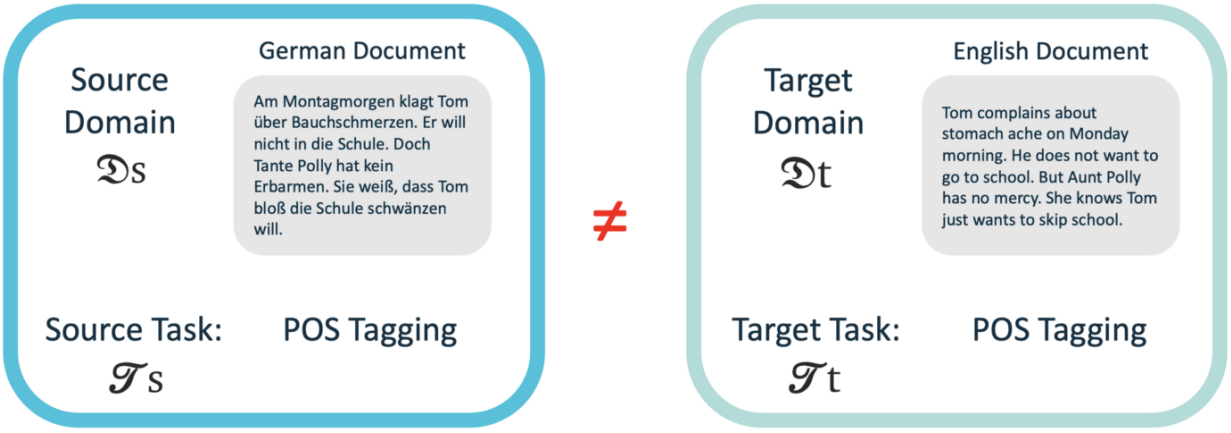
In general, if two tasks are different, then they may have different label spaces($Yt \neq Ys$) or different conditional probability distributions ($P(Yt|Xt) \neq P(Ys|Xs)$).

Definition

Given a source domain $\mathfrak{D}s$ and corresponding learning task $\mathcal{T}s$, a target domain $\mathfrak{D}t$ and learning task $\mathcal{T}t$, transfer learning aims to improve the learning of the conditional probability distribution $P(Yt|Xt)$ in $\mathfrak{D}t$ with the information gained from $\mathfrak{D}s$ and $\mathcal{T}s$, where $\mathfrak{D}t \neq \mathfrak{D}s$ or $\mathcal{T}t \neq \mathcal{T}s$. For simplicity, we only used a single source domain in the definition above, but the idea can be extended to multiple source domains.

If we take this definition of domain and task, then we will have either $\mathfrak{D}t \neq \mathfrak{D}s$ or $\mathcal{T}t \neq \mathcal{T}s$, which results in four common transfer learning scenarios[3]. We will explain these scenarios below in the context of two popular machine learning tasks, part of speech (POS) tagging, and object classification.

POS tagging is the process of associating a word in a corpus with its corresponding part of speech tag, based on its context and definition. For example: In the sentence "My name is Azin." 'My' is a 'PRP', 'name' is 'NN', 'is' is 'VBZ', and 'Azin' is 'NNP'. Object classification is the process of classifying the objects seen in an image to the set of defined classes like apple, bus, forest etc. Let's take a look at the following four cases with these tasks in mind.



An example of a transfer learning scenario where tasks are the same ($\mathcal{T}t = \mathcal{T}s$), but domains have different feature spaces ($Xt \neq Xs$).

1. $Xt \neq Xs$ Let's say that we would like to do POS tagging in German documents ($\mathcal{T}t$). Assuming that the basics of Germany and English are similar in grammar and structure, we can leverage the knowledge learned from thousands of existing rich English datasets ($\mathcal{D}s$) for this task (Figure 4), even though our features spaces (English and German words) are completely different ($Xt \neq Xs$). Another example is using tags and descriptions ($\mathcal{D}s$) provided alongside images to improve the object classification task ($\mathcal{T}t$) where text and images are presented in completely different features spaces ($Xt \neq Xs$).

2. P(Xt) ≠ P(Xs) Let's say that we would like to do POS tagging in English documents ($\mathcal{T}t$) and we would like to use readily available English datasets for this. Although these documents are written in the same language ($Xt = Xs$), they focus on different topics, and so the frequency of the words used (features) is different. For example, in a cooking document, "tasty" or "delicious" might be common, but they would be rarely used in a technical document. Words that are generic and domain-independent occur at a similar rate in both domains. However, words that are domain-specific are used more frequently in one domain because of the strong relationship with that domain topic. This is referred to as frequency feature bias and will cause the marginal distributions between the source and target domains to be different. Another example is using cartoon images to improve object classification for photo images. They are both images so the features

spaces are the same (Xt =Xs), however, the colors and shapes in cartoons are very different to photos (P(Xt) ≠ P(Xs)). This scenario is generally referred to as domain adaptation.

3. Yt ≠ Ys Let's say we want to do POS tagging with a customized set of tags ($\mathcal{T}$t) which is different from tags in other existing datasets. In this case, the target and source domains have different label spaces (Yt ≠ Ys). Another example would be using a data set with different object classes (cat and dog) to improve object classification for a specific set of classes (chair, desk, and human).

4. P(Yt|Xt) ≠ P(Ys|Xs) In POS tagging the source and target can have the same language(Xt = Xs), the same number of classes (Yt =Ys), and same frequency of words (P(Xt)=P(Xs)), but individual words can have different meanings in the source and target (P(Yt|Xt) ≠ P(Ys|Xs)). A specific example is the word "monitor." In one domain (technical reports) it might be used more frequently as a noun and in another domain (patient monitoring reports), it might be used predominantly as a verb. This is another form of bias which is referred to as context feature bias and it causes the conditional distributions to be different between the source and target. Similarly, in an image, if a piece of bread is hung on a refrigerator it is most likely to be a magnet, however, if it is close to a pot of jam it is more likely to be a piece of bread.

One final case that does not sit well in any of the four above (based on our definitions) but that can cause a difference between the source and target is P(Yt) ≠ P(Ys). For example, the source dataset may have completely balanced binary samples, but the target domain may have 90% positive and only 10% negative samples. There are different techniques for addressing this issue, such as down-sampling and up-sampling, and SMOTE.

Categorization of Transfer Learning Problems

When you read the literature on transfer learning, you'll notice that the terminology and definitions are often inconsistent. For example, domain adaptation and transfer learning are sometimes used to refer to the same concept. Another common inconsistency is how transfer learning problems are grouped together. Traditionally transfer learning problems were categorized into three main groups

based on the similarity between domains and also the availability of labeled and unlabeled data[1]: Inductive transfer learning, transductive transfer learning, and unsupervised transfer learning.

However, thanks to developments in deep learning, the recent body of research in this field, has broadened the scope of transfer learning and a new and more flexible taxonomy[3] has emerged. This taxonomy generally categorizes transfer learning problems into two main classes based on the similarity of domains, regardless of the availability of labeled and unlabeled data[3]: Homogeneous transfer learning and heterogeneous transfer learning. We will explore this taxonomy below.

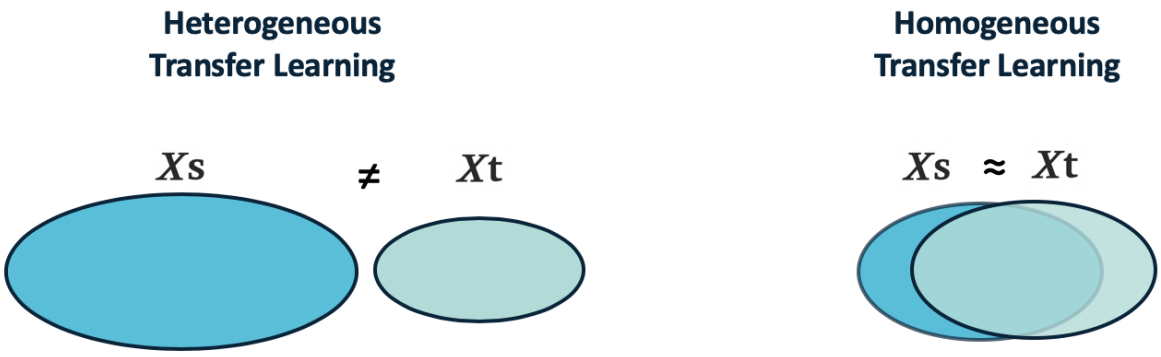
1. Homogeneous Transfer Learning

In homogeneous transfer learning (Figure 5 right side), we have the situation where Xt = Xs and Yt = Ys. Therefore, we want to bridge the gap in the data distributions between the source and target domains, i.e. address P(Xt) ≠ P(Xs) and/or P(Yt|Xt) ≠ P(Ys|Xs). The solutions to homogeneous transfer learning problems use one of the following general strategies:

1. Trying to correct for the marginal distribution differences in the source and target (P(Xt) ≠ P(Xs)).
2. Trying to correct for the conditional distribution difference in the source and target (P(Yt|Xt) ≠ P(Ys|Xs)).
3. Trying to correct both the marginal and conditional distribution differences in the source and target.

2. Heterogeneous Transfer Learning

In heterogeneous transfer learning, the source and target have different feature spaces Xt ≠ Xs (generally non-overlapping) and/or Yt ≠ Ys, as the source and target domains may share no features and/or labels (Figure 5 left side). Heterogeneous transfer learning solutions bridge the gap between feature spaces and reduce the problem to a homogeneous transfer learning problem where further distribution (marginal or conditional) differences will need to be corrected.



The two main classes of transfer learning problems: Homogeneous transfer learning and heterogeneous transfer learning.

Another important concept to discuss is negative transfer. If the source domain is not very similar to the target domain, the information learned from the source can have a detrimental effect on a target learner. This is referred to as negative transfer.

In the rest of the post, we will describe the existing methods for homogeneous and heterogeneous transfer learning. We'll also discuss some techniques to avoid negative transfer.

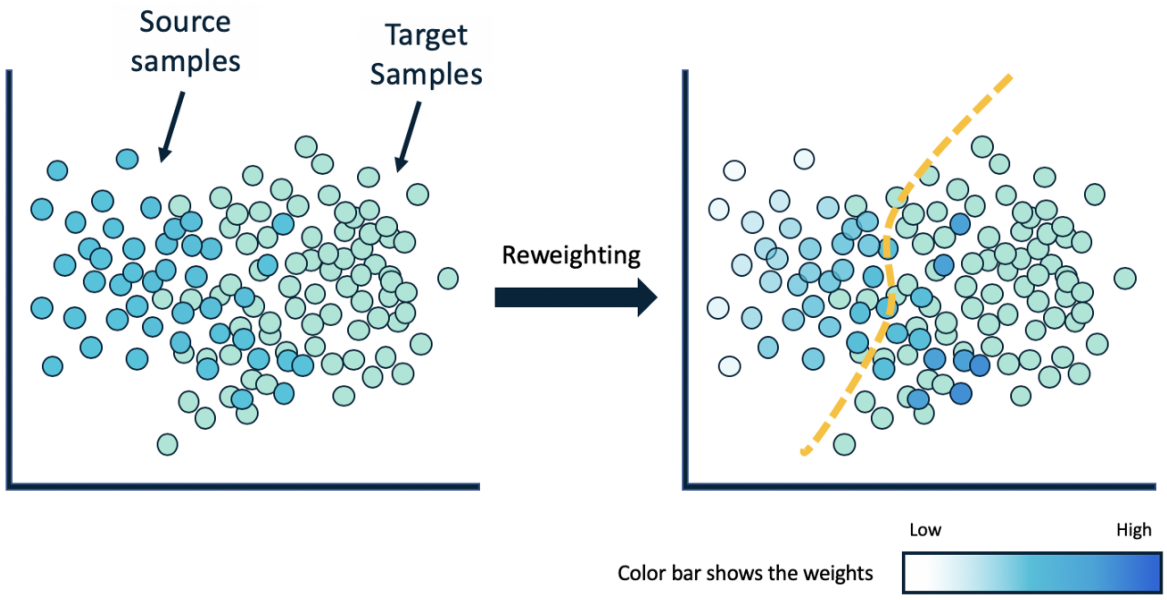
Categorization of Transfer Learning Solutions

Solutions for the two main categories of transfer learning problems above can be summarized into five different classes based on what is being transferred:

Homogeneous Transfer Learning

1.Instance-based Approaches: Instance-based transfer learning methods try to reweight the samples in the source domain in an attempt to correct for marginal distribution differences[4, 5, 6]. These reweighted instances are then directly used in the target domain for training. Using the reweighted source samples helps the target learner to use only the relevant information from the source domain. These methods work best when the conditional distribution is the same in both domains.

Instance-based approaches differ in their weighting strategies. For example, the method in Correcting Sample Selection Bias by Unlabeled Data (Huang et al.) [7] finds and applies the weight that matches up the mean of the target and source domains. Another common solution is to train a binary classifier that separates source samples from target samples and then use this classifier to estimate the source sample weights (Figure 6). This method gives a higher weight to the source samples that are more similar to target samples.

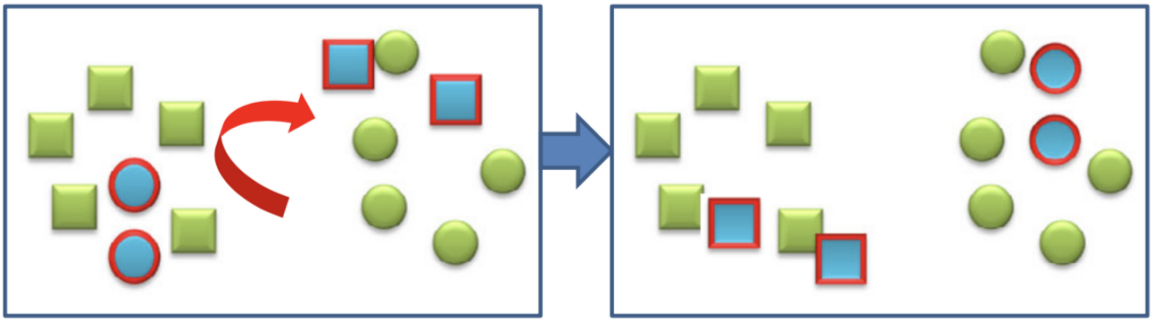


One simple yet effective instance-based approach to reweight the source samples is to train a binary classifier that distinguishes source samples from target samples and use this classifier to estimate source sample weights.

2. Feature-based Approaches: Feature-based approaches are applicable to both homogeneous and heterogeneous problems. With heterogeneous problems, the main goal of using these methods is to reduce the gap between feature spaces of source and target[10, 11, 14, 15]. With homogeneous problems, these methods aim to reduce the gap between the marginal and conditional distributions of the source and target domains[8, 9, 12, 13]. Feature-based transfer learning approaches fall into two groups:

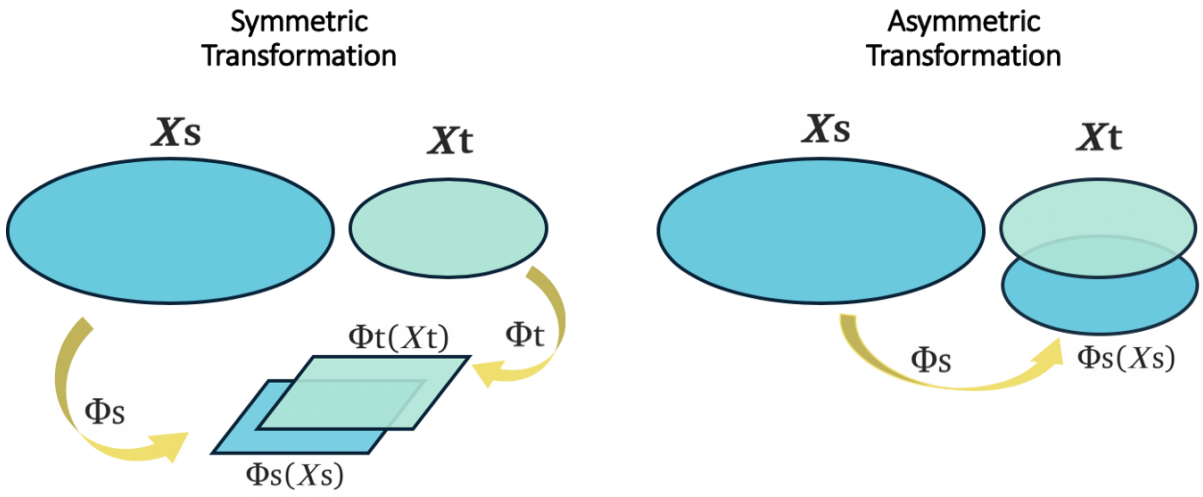
- Asymmetric Feature Transformation: This approach employs a transformation (Φs/Φt) to transform one of the domains (source/target) to the other one (target/source)[8, 9, 25]. This method works the best

when the source and target domains have the same label spaces and one can transform them without context feature bias (Figure 8 right side). Figure 7, shows the asymmetric feature-based method as presented in Asymmetric and Category Invariant Feature Transformations for Domain Adaptation (Hoffman et al.) [25]. This method tries to transform the source domain (blue samples) so that the distance between similar samples in source and target domains is minimized.



An asymmetric transformation — a rotation applied only to source domain (blue samples) — successfully compensates for domain shift.

- Symmetric Feature Transformation: This approach discovers underlying meaningful structures by transforming both of the domains to a common latent feature space — usually of a low dimension — that has predictive qualities while reducing the marginal distribution between the domains (Figure 8 left side)[12, 13]. Although the high-level goal behind these methods (improving the performance of a target learner) is very different from the objective in representation learning, the idea behind them is fairly close[23].



Symmetric feature-based methods are presented in the left side and asymmetric feature-based methods are shown on the right side. These methods are applicable to both classes of transfer learning problems. However, to illustrate the point clearly, we used heterogeneous feature spaces here.

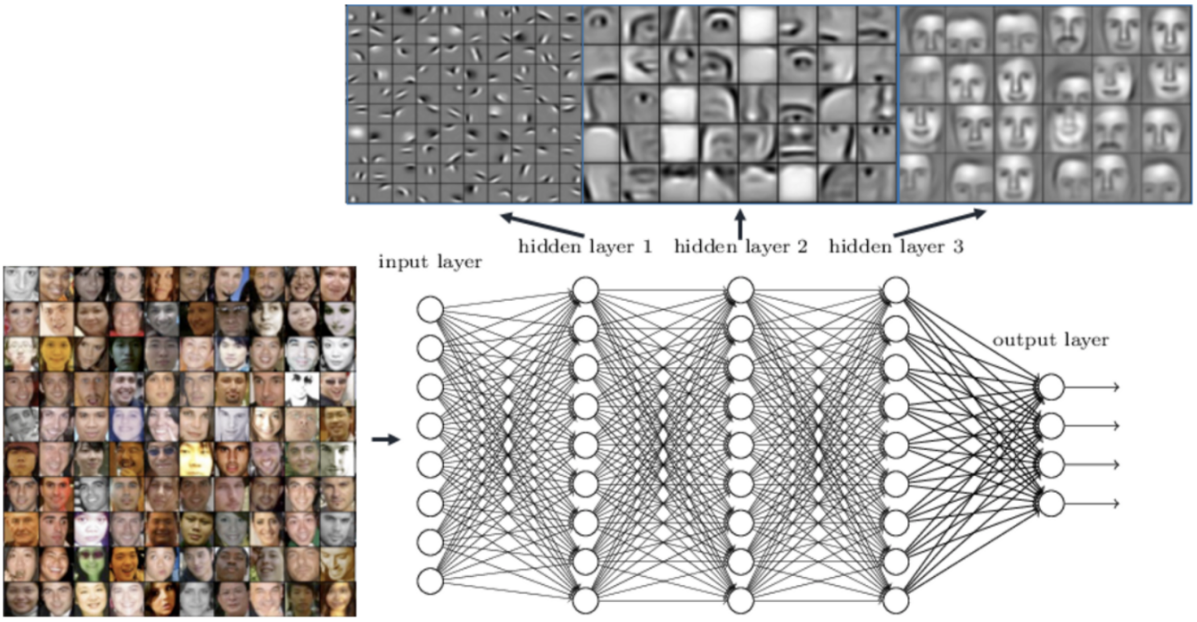
3.Parameter-based_Approaches:

This category of transfer learning tries to transfer knowledge through the shared parameters of the source and target domain learner models[16, 17]. Some of these methods also transfer the learned knowledge by creating multiple source learner models and optimally combining the reweighted learners (ensemble learners) to form an improved target learner. The idea behind parameter-based methods is that a well-trained model on the source domain has learned a well-defined structure, and if two tasks are related, this structure can be transferred to the target model.

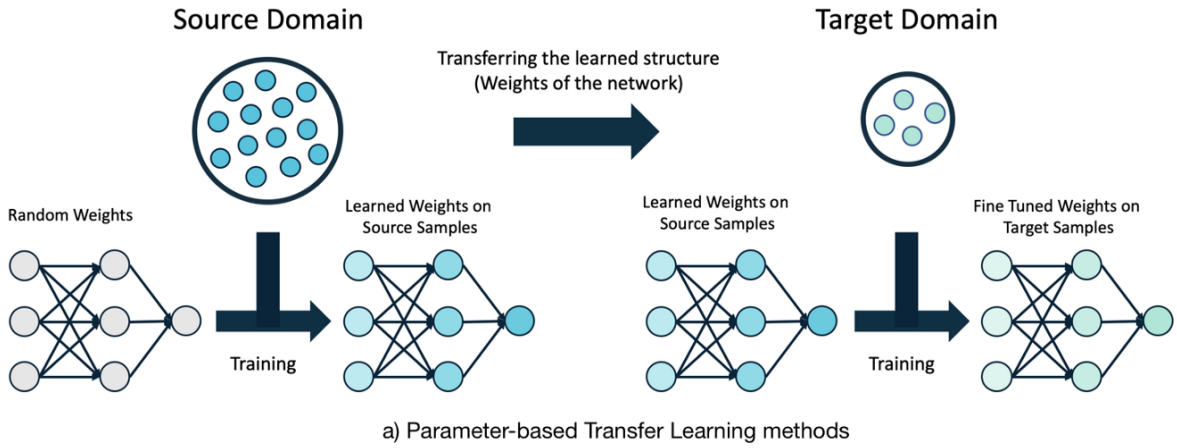
The concept of sharing parameters (weights) has been widely used in deep learning models. In general, there are two ways to share the weights in deep learning models, soft weight sharing, and hard weight sharing. In soft weight sharing, the model is usually penalized if its weights deviate significantly from a given set of weights[18]. In hard weight sharing, the exact weights are shared among different models[19]. Very commonly hard weight sharing uses previously trained weights as the starting weights of a deep learning model.

Usually, when training a deep neural network, the model starts with randomly initialized weights close to zero and adapts its weights as it sees more and more training samples. However, training a deep model in this way requires a lot of time and effort to collect and label data. That's why it's so advantageous to start with the previously trained weights from another similar domain (source) and then fine-tune

the weights specifically for a new domain (target). This can potentially save time and reduce the costs since fine-tuning requires much less labeled data. It's also been shown that this approach can help with robustness. Figure 9, demonstrates this method.



b) Deep neural networks learn hierarchical representations [26]



a) Parameter-based Transfer Learning methods

(a) Parameter-based transfer learning methods are used widely in the context of deep learning. A very popular example is using the weights of a pre-trained network and fine-tuning them for a new domain. (b) An illustration of weights in neural networks.

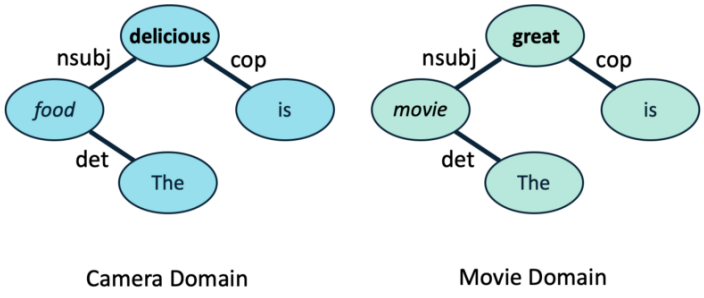
4. Hybrid-based Approaches (Instance and Parameter): This category focuses on transferring knowledge through both instances and shared parameters. This is a relatively new approach and a lot of interesting research is emerging.

5. Relational-based Approaches: The last transfer learning category (and also the newest) is to transfer the knowledge through learning the common relationships between the source and target domains[21, 22]. In the following, we provide an example here in the context of sentiment analysis.

As you see in Figure 10, the frequency of used words in different review domains (here Movie and Food) is very different, but the structure of the sentences is fairly similar. Therefore, if we learn the relationship between different parts of the sentence in the Food domain, it can help significantly with analyzing the sentiment in another domain.

Domain	Reviews
Food	The *food* is **delicious**. I highly **recommend** this *Patsa*. This is very **amazing** *meal*.
Movie	The *movie* is **great**. I **love** this *movie*. *Godfather* was the most **amazing** *movie*.

Reviews in movie and food domains. Boldfaces are topic words and Italics are sentiment words.



Dependency tree structure.

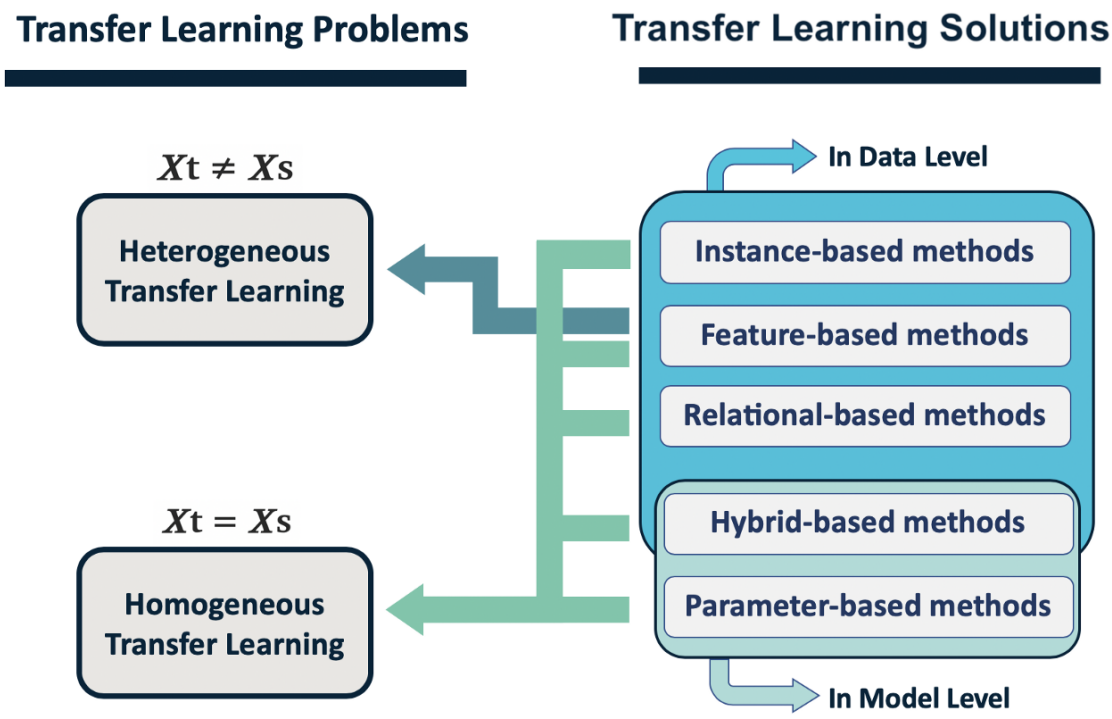
Figure 10. Relational-based approaches to transfer learning — learning sentence structure from Food domain can help with analyzing the sentiment in a different domain like Movie [21].

Heterogeneous Transfer Learning

Since in heterogeneous transfer learning problems the features spaces are not equivalent, the only class of transfer learning solutions that can be applied to this category is the feature-based methods[10, 11, 14, 15] described above (Figure 8). After reducing the gap between features spaces by using symmetric[10, 11] or asymmetric[14, 15] feature-based methods and converting a heterogeneous problem to a homogeneous problem, the other four classes of transfer learning solutions can be applied to further correct the distribution differences between target and source domains.

Summary

Figure shows the different categories of transfer learning problems and solutions. As you can see, some classes of transfer learning solutions (instance-based, feature-based, and relational-based methods) transfer the knowledge at the data level while some classes (parameter-based methods) transfer the knowledge at the model level.

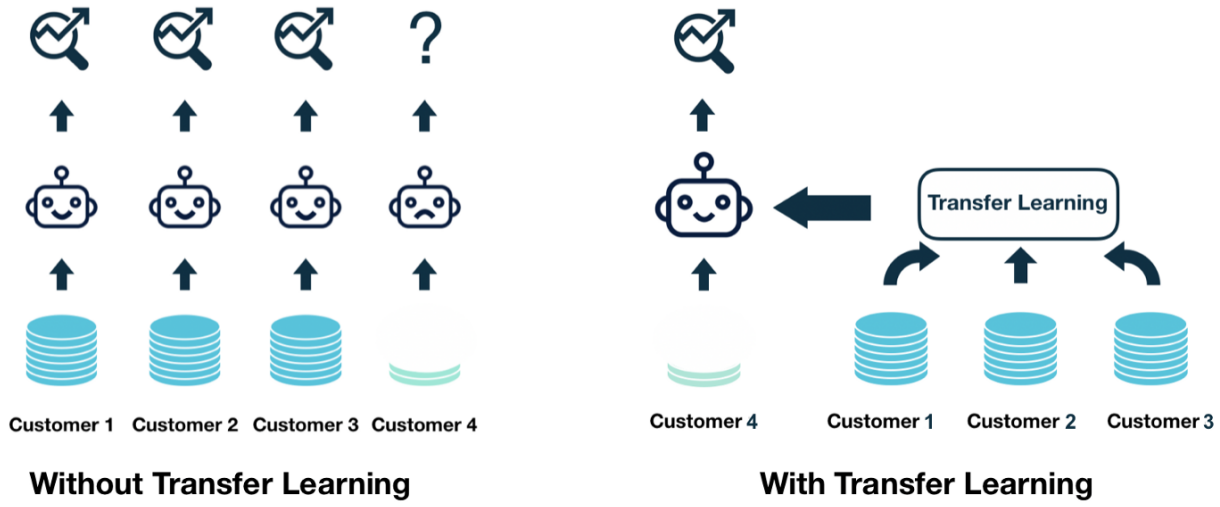


Summary of transfer learning problems and solutions to address them.

Real Applications of Transfer Learning

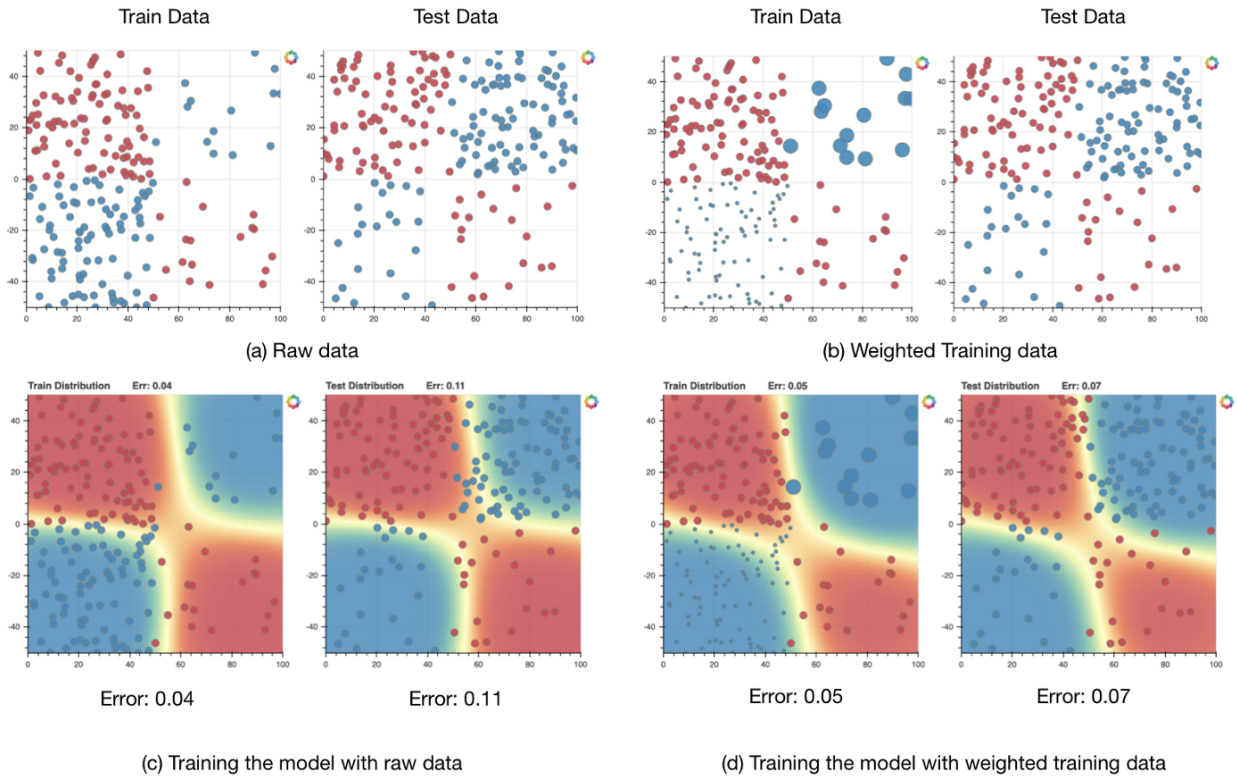
One of the biggest challenges that SaaS/AI companies often face when onboarding a new customer is the lack of labeled data to build a machine learning model for the new customer, which is referred to as cold-start. Even after collecting data, which can potentially take several months to years, there are other issues like sparsity and imbalance problems that make it hard to build a model with acceptable performance. When a model uses sparse and imbalanced data, it is often not sufficiently expressive and performs poorly especially on minority classes.

The aforementioned issues negatively affect customers acquisition and retention and make it extremely difficult for SaaS/AI companies to scale efficiently. Transfer learning can overcome these issues by leveraging existing information from other customers (source domain) when building the model for the new customer (target domain).



A schematic view of the application of machine learning in the industry with (right side) and without (left side) transfer learning.

Figure 13 illustrates another machine learning challenge that can be addressed by transfer learning — domain shift. Supervised machine learning models still lack the ability to generalize to conditions that are different from the ones encountered during training. In other words, when the statistical characteristics of the training data and the test data are different (domain shift), we often experience performance deterioration or collapse as the model doesn't know how to generalize to the new situation. Transfer learning (domain adaptation) methods can help to address this issue by reducing the gap between the two domains and consequently improving the performance of the model on the test data.



The raw data and the reweighted samples based on the method in Dataset Shift in Machine Learning (Quiñonero-Candela) are shown in Figures (a) and (b) respectively. Figures (c) and (d) show the SVM binary classifier (with RBF kernel) trained with raw data and reweighted samples correspondingly. comparing figures (c) and (d) we can see that the error on the test set has decreased from 0.11 to 0.07 (around 36%)

- **Pretrained Models-**

A pre-trained model is a model created by someone else to solve a similar problem. Instead of building a model from scratch to solve a similar problem, we can use the model trained on other problem as a starting point. A pre-trained model may not be 100% accurate in your application.

Keras Applications are deep learning models that are made available alongside pre-trained weights. These models can be used for prediction, feature extraction, and fine-tuning.

Weights are downloaded automatically when instantiating a model. They are stored at ~/.keras/models/.

The following image classification models (with weights trained on ImageNet) are available:

- Xception
- VGG16
- VGG19
- ResNet50
- InceptionV3
- InceptionResNetV2
- MobileNet
- MobileNetV2
- DenseNet
- NASNet

- **VGG16**

VGG16 is a simple and widely used Convolutional Neural Network (CNN) Architecture used for ImageNet, a large visual database project used in visual object recognition software research. The VGG16 Architecture was developed and introduced by Karen Simonyan and Andrew Zisserman from the University of Oxford, in the year 2014, through their article "Very Deep Convolutional Networks for Large-Scale Image Recognition." 'VGG' is the abbreviation for Visual Geometry Group, which is a group of researchers at the University of Oxford who developed this architecture, and '16' implies that this architecture has 16 layers (explained later).

The VGG16 model achieved 92.7% top-5 test accuracy in ImageNet, which is a dataset of over 14 million images belonging to 1000 classes. It was one of the famous models submitted to ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in the year 2014. It made improvements over AlexNet architecture by replacing large kernel-sized filters (11 and 5 in the first and second convolutional layer, respectively) with multiple three × three kernel-sized filters one after another. VGG16 was trained for weeks using NVIDIA Titan Black GPUs.

VGG16 is used in many deep learning image classification techniques and is popular due to its ease of implementation. VGG16 is extensively used in learning applications due to the advantage that it has.

VGG16 is a CNN Architecture, which was used to win the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) in 2014. It is still one of the best vision architecture to date.

VGG16 architecture

During training, the input to the convnets is a fixed-size 224 x 224 RGB image. Subtracting the mean RGB value computed on the training set from each pixel is the only pre-processing done here. The image is passed through a stack of convolutional (conv.) layers, where filters with a very small receptive field: 3 × 3 (which is the smallest size to capture the notion of left/right, up/down, center and has the same effective receptive field as one 7 x 7), is used. It is deeper, has more non-linearities, and has fewer parameters. In one of the configurations, 1 × 1 convolution filters, which can be seen as a linear transformation of the input channels (followed by non-linearity), are also utilized. The convolution stride and the spatial padding of conv. layer input is fixed to 1 pixel for 3 x 3 convolutional layers, which ensures that the spatial resolution is preserved after convolution. Five max-pooling layers, which follow some of the convolutional layers, helps in spatial pooling. Max-pooling is performed over a 2×2 pixel window, with stride 2.

There are three Fully-Connected (FC) layers that follow a stack of convolutional layers (these have different depths in different architectures): the first two have 4096 channels each, the third performs 1000-way ILSVRC classification and thus contains 1000 channels (one for each class). The final layer is the soft-max layer. The configuration of the fully connected layers is the same in all networks.

The 16 layer VGG architecture was the best performing, and it achieved a top-5 error rate of 7.3% (92.7% accuracy) in ILSVRC — 2014, as mentioned above. VGG16 had significantly outperformed the previous generation of models ILSVRC — 2012 and ILSVRC — 2013 competitions.

The VGG16 architecture is depicted in figure 1, shown below:

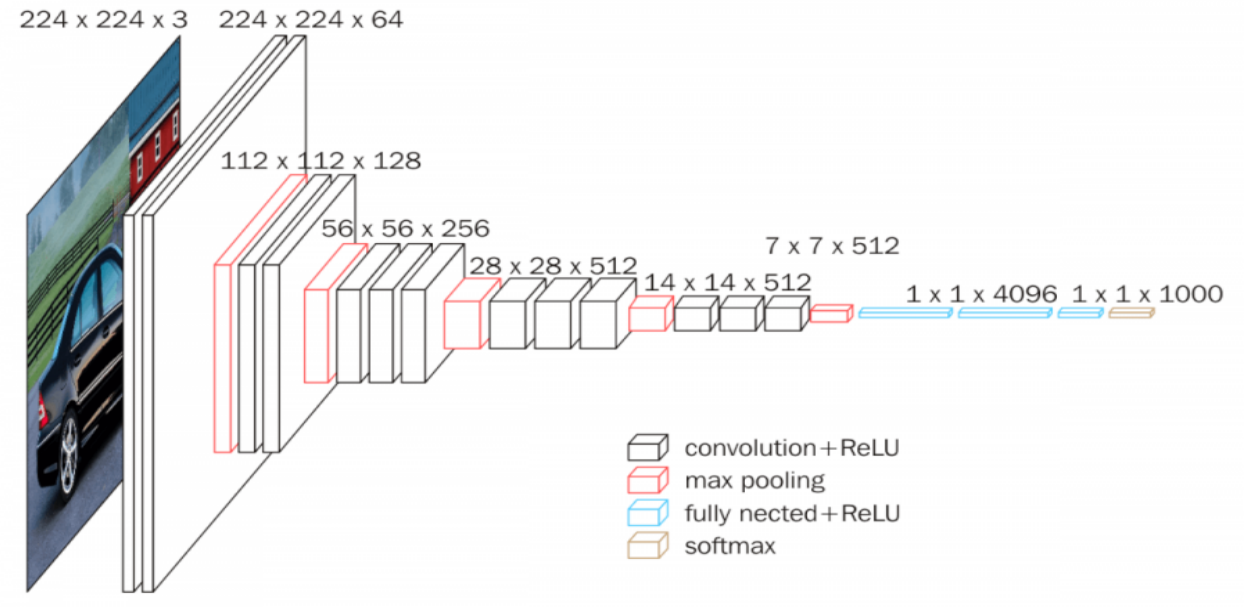


Figure 1. VGG16 architecture

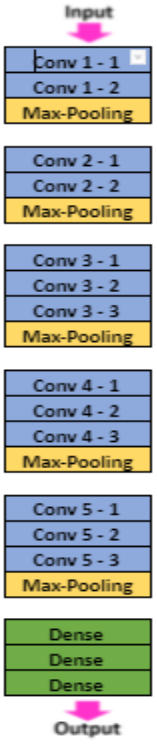


Figure 2. VGG16 layers

VGG16 architecture is an improvement over AlexNet (ILSVRC — 2012 winner) wherein it replaced large kernel-sized filters with multiple 3 x 3 kernel-sized filters one after the other.

How to train VGG16 from scratch?

The ConvNet training procedure is generally carried out by optimizing the multinomial logistic regression objective using mini-batch gradient descent (based on back-propagation) with momentum. The batch size was set to 256, and momentum was to 0.9. The training was regularised by weight decay (the L2 penalty multiplier set to $5 \cdot 10^{-4}$) and dropout regularisation for the first two fully-connected layers (dropout ratio set to 0.5). The learning rate was initially set to 10^{-2} and then decreased by a factor of 10 when the validation set accuracy stopped improving. In total, the learning rate was decreased 3 times, and the learning was stopped after 370,000 iterations (74 epochs). It is conjectured that in spite of the larger number of parameters and the greater depth of convolutional networks, the nets required fewer epochs to converge due to (a) implicit regularisation imposed by the greater depth and smaller conv. filter sizes; (b) pre-initialization of certain layers.

The initialization of the network weights is important since bad initialization can stall learning due to the instability of the gradient in deep nets. To circumvent this problem, the training of configuration A (Table 1), which is shallow enough to be trained with the random initialization, is begun. Then, when training deeper architectures, the first four convolutional layers and the last three fully-connected layers with the layers of net A (the intermediate layers were initialized randomly) are initialized. The learning rate for the pre-initialized layers is not decreased, allowing them to change during learning. For random initialization (where applicable), the weights from a normal distribution with the zero mean and 10^{-2} variance are sampled. The biases were initialized with zero.

It is worth noting that after the paper submission of the original work done by Karen Simonyan and Andrew Zisserman, it was found to be possible to initialize the weights without pre-training by using the random initialization procedure of Glorot & Bengio (2010).

Table 1: **ConvNet configurations** (shown in columns). The depth of the configurations increases from the left (A) to the right (E), as more layers are added (the added layers are shown in bold). The convolutional layer parameters are denoted as "conv⟨receptive field size⟩-⟨number of channels⟩". The ReLU activation function is not shown for brevity.

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

Table 1. ConvNet configurations (displayed in columns). It can be noted that the depth of the configurations increases from the left (A) to the right (E) as more layers are added (displayed in bold). The convolutional layer parameters are denoted as "conv (receptive field size) — (number of channels)." The ReLU activation function is not shown here.

The ConvNet input images were randomly cropped from rescaled training images (one crop per image per SGD iteration) to obtain the fixed-size 224 × 224 ConvNet images. To further augment the training set, the crops underwent random horizontal flipping and random RGB color shift (Krizhevsky et al., 2012). Training image rescaling is explained below.

Training image size:

Let S be the smallest side of an isotropically-rescaled training image, from which the ConvNet input is cropped (we also refer to S as the training scale). While the crop size is fixed to 224 × 224, in principle, S can take on any value not less than 224: for S = 224, the crop will capture whole-image statistics, completely spanning the smallest side of a training image; for S ≫ 224 the crop will correspond to a small part of the image, containing a small object or an object part.

Two approaches are considered for setting the training scale S. The first is to fix S, which corresponds to single-scale training (note that image content within the sampled crops can still represent multi-scale image statistics). For experimental evaluation, the evaluation of models is trained at two fixed scales: S = 256 (which has been widely used in the prior art (Krizhevsky et al., 2012; Zeiler & Fergus, 2013; Sermanet et al., 2014)) and S = 384. Given a ConvNet configuration, the network was first trained using S = 256. To speed up the training of the S = 384 network, it was initialized with the weights pre-trained with S = 256, and a lower initial learning rate of 10−3 was used. The second approach to setting S is multi-scale training, where each training image is individually rescaled by randomly sampling S from a certain range [Smin, Smax] (Smin = 256 and Smax = 512 was used). Since objects in images can be of different sizes, it is beneficial to take this into account during training. This can also be seen as training set augmentation by scale jittering, where a single model is trained to recognize objects over a wide range of scales. For speed reasons, the training of multi-scale models was done by fine-tuning all layers of a single-scale model with the same configuration, pre-trained with fixed S = 384.

Algorithm:

Step 1: Import library to need

Step 2: Import the dataset for model

Step 3: Add the neural layers as per preference

Step 4: Extract the VGG16 model into code using standard syntax

Step 4: fit the model and start the training

Step 6: Test the model with test image

Outcome:

In this learned about pre-trained model in deep neural network also use in real programming cases

Conclusion:

This experiment gives idea about new concept of deep learning i.e. Transfer learning, using this we can build our model with good accuracy one trained model can use as a validation data for model that we are building.

Experiment No-08

Title: Develop a linear regression model using deep neural network for predicting price of houses in Boston city

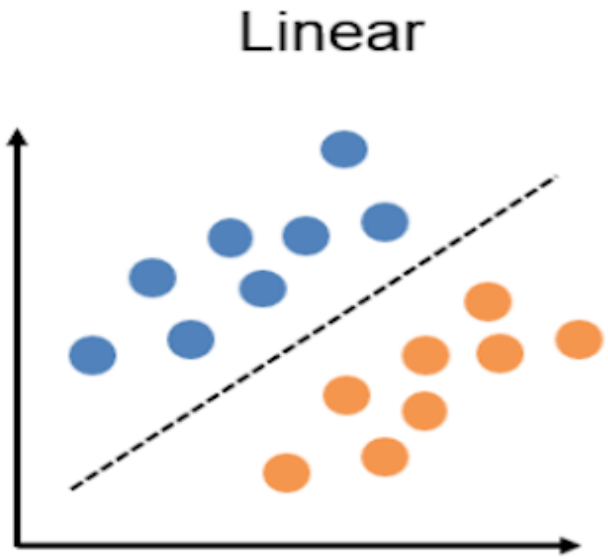
Objective: To Develop and test House Price Prediction using linear regression

Tools: Spyder IDE/Google Colab/Jupyter Notebook

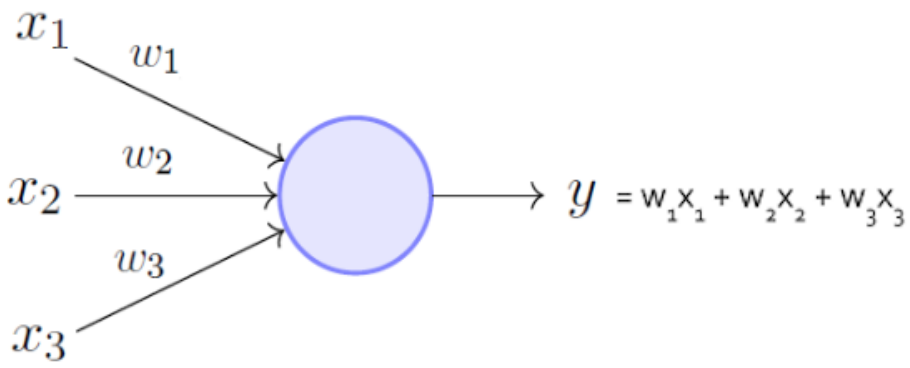
Theory:

A beginner in data science, after going through the concepts of Regression, Classification, Feature Engineering etc. and enters into the field of deep learning, it would be very beneficial if one can relate the functionality of algorithms in deep learning with above concepts.

Before understanding ANN, let us understand a perceptron, which is a basic building block of ANN. Perceptron is the name initially given to a binary classifier. However, we can view the perceptron as a function which takes certain inputs and produces a linear equation which is nothing but a straight line. This can be used to separate certain easily separable data as shown in the figure. However, remember that in real-world scenarios, classes will not be so easily separable.



The structure of a perceptron can be visualised as below:



This means generating multiple linear equations at multiple points. These perceptrons can also be called as neurons or nodes which are actually the basic building blocks in natural neural network within our body. In the above figure, the first vertical set of 3 neurons is the input layer. The next two vertical sets of neurons are part of the middle layer which are usually referred to as hidden layers, and the last single neuron is the output layer. The neural network in the above figure is a 3-layered network. This is because the input layer is generally not counted as part of network layers. Each neuron in the input layer represents an attribute (column) in the input data (i.e., x1, x2, x3 etc.). What is happening in the above network is that input data is fed to set of neurons, and each produces an output. Again, each of these outputs are fed to other neurons which in turn produces another output, which is again fed to the output layer. Error calculated at this output layer is again sent back in the network to further refine the outputs of each neuron which are again fed to the neuron in output layer to produce a refined output than before. As explained in the 5-step process above, this process is repeated until we get an output with minimal error.

The process of producing outputs, calculating errors, feeding them back again to produce a better output is generally a confusing process, especially for a beginner to visualise and understand. Hence, an effort is made here to explain this process with just one neuron and one layer. Once this basic concept is understood, expanding this to a larger neural network is not difficult.

Everyone agrees that simple linear regression is the simplest thing in machine learning or atleast the first thing that anyone learns in machine learning. So, we

will try to understand this concept of deep learning also with a simple linear regression, by solving a regression problem using ANN.

Implementing ANN for Linear Regression

We have understood from the above that each of the neuron in the ANN except the input layer produces an output. The output is based on what function that we use. This function is generally referred as 'Activation Function'. As ANN is mainly used for classification purposes, generally sigmoid function or other similar classification algorithms are used as activation functions. But, as we are now trying to solve a linear regression problem, our activation function here is nothing but a 'Simple Linear Equation' of the form –

$y=w_0 + w_1x_1 + w_2x_2 + w_3x_3 + \ldots. w_nx_n$

where x_1, x_2, x_3.. x_n are the independent attributes in the input data,

w_1, w_3… w_n are the weights (Co-efficients) to corresponding attributes, and

w_0 is the bias

Because our output should just be a single linear line, we should configure our ANN with just 1 neuron. As the output of this 1 neuron itself is the linear line, this neuron will be placed in the output layer. Hidden layers are required when we try to classify objects with using multiple lines (or curves). So, we don't need any hidden layers as well here.

Hence the ANN to solve a linear regression problem consists of an input layer with all the input attributes and an output layer with just 1 neuron as shown below:

Now, we have finalised the structure of our ANN. Our next task is to actually write code to implement it. We will be implementing this simple ANN from scratch as that will help to understand lot of underlying concepts in already available ANN libraries.

Recall the 5 steps that are mentioned at the beginning. As mentioned there, the process involves feeding input to a neuron in the next layer to produce an output using an activation function. This process is called as 'Feed Forward'. After producing the output, error (or loss) is calculated and a correction is sent back in

the network. This process is called as 'Back Propagation'. We will also use some standard terminologies for our ANN network such as 'Network', 'Topology' etc. which we will see in the code. With various terms and terminologies that we have learnt so far, let us implement the code –

1. Import the required libraries

2. Initialise the weights and other variables

In our approach, we will be providing input to the code as a list such as [2,3,1]. Here, the total no. of values present in the list (list size) indicate the number of layers that we want to configure, and each number in the list indicate the no. of neurons inside each layer. So, the list [2,3,1] indicates our network should consists of 3 layers in which first layer consists of 2 neurons, second layer consists of 3 neurons and output layer consists of 1 neuron. This structure can be called as 'network topology'. However, as we are solving regression problem, we just need 1 neuron at the output layer as discussed above. So, we just need to pass the input list as [1].

In our approach to build a Linear Regression Neural Network, we will be using Stochastic Gradient Descent (SGD) as an algorithm because this is the algorithm used mostly even for classification problems with a deep neural network (means multiple layers and multiple neurons). I will assume the reader is already aware of this algorithm and proceed with its implementation.

We will initialise all the weights to zeros. Let us create a class called 'Network' and initialise all required variable in the constructor as below –

'self.output' variable in the above code is to hold the outputs of each neuron. It will be initialised accordingly with a sufficient sized list based on our input. Remaining variables are pretty self-explanatory.

3. Coding 'fit' function

We know that the gradient descent algorithm requires 'learning rate' (eta) and no. of iterations (epoch) as inputs. We will be passing all these values in a list to the

program along with the training data. Let us build a 'fit' method to construct a predictive model with all the inputs given –

4. Produce the Output and Correct the Error

I have mentioned above what 'feed forward' and 'back propagation' are. Let us implement those methods –

function is just forming a simple linear equation of y = mx + c kind and nothing more.

In SGD algorithm, we continuously update the initialised weights in the negative direction of the slope to reach the minimal point.

Error function $E(w) = \sum[(w_0 + w_1x_1 - y_1)^2 + (w_0 + w_1x_2 - y_2)^2 + \ldots + (w_0 + w_1x_n - y_n)^2]$

Here, I have not taken ½ as scaling factor to the equation. One may take if desired so. Also, in SGD only one row is passed to the above error function every time to calculate the error. Hence, if we differentiate the above equation w.r.t. each of the weights w_0, w_1, w_2 .. etc., we get equations like

$\partial E0 = 1,$

$\partial E1 = 2*(w_0 + w_1x_1 - y_1)*x_1$

After calculating the slope w.r.t. each of the weights, we will be updating the weights with new values in the negative direction of the slope as below –

$W_n = w_n - \eta * \frac{\partial E}{\partial w_n}$ where η is the learning rate.

Let us implement all this logic in the back propagate function as below:

In order to visualise the error at each step, let us quickly write functions to calculate Mean Squared Error (for full dataset) and Squared Error (for each row) which will be called for each step in an epoch.Having the model built in the above

way, let us define a method which takes some input and predicts the output –That's it. We have built a simple neural network which builds a model for linear regression and also predicts values for unknowns.

Outcome: Got understanding about the linear regression using neural network

Conclusion:

Hence, we understood the Implementation of Regression using Deep Neural Network.

Algorithm:

Step 1: Importing important libraries

Step 2: Import the train and test data

Step 3: Defining the Labels & preprocessing the data

Step 4: Defining the Neural networks layers

Step 5: starting the training the model

Step 6: Calculating the MSE & Predicting the pricing from the test data