**DEEP LEARNING**

**Biological inspiration of Neural Networks**

A neuron (nerve cell) is the basic building block of the nervous system. A human brain consists of billions of neurons that are interconnected to each other. They are responsible for receiving and sending signals from the brain. As seen in the below diagram, a typical neuron consists of the three main parts – dendrites, an axon, and cell body or soma. Dendrites are tree-like branches originating from the cell body. They receive information from the other neurons. Soma is the core of a neuron. It is responsible for processing the information received from dendrites. Axon is like a cable through which the neurons send the information. Towards its end, the axon splits up into many branches that make connections with the other neurons through their dendrites. The connection between the axon and other neuron dendrites is called synapses.

Diagram

Description automatically generated

As ANN is inspired by the functioning of the brain, let us see how the brain works. The brain consists of a network of billions of neurons. They communicate by means of electrical and chemical signals through a synapse, in which, the information from one neuron is transmitted to other neurons. The transmission process involves an electrical impulse called ‘action potential’. For the information to be transmitted, the input signals (impulse) should be strong enough to cross a certain threshold barrier, then only a neuron activates and transmits the signal further (output).

Inspired by the biological functioning of a neuron, an American scientist Franck Rosenblatt came up with the concept of perceptron at Cornell Aeronautical Laboratory in 1957.

* A neuron receives information from other neurons in form of electrical impulses of varying strength.
* Neuron integrates all the impulses it receives from the other neurons.
* If the resulting summation is larger than a certain threshold value, the neuron ‘fires’, triggering an action potential that is transmitted to the other connected neurons.

**A Perceptron is an Artificial Neuron. It is the simplest possible Neural Network.** Neural Networks are the building blocks of Machine Learning.

Main Components of Perceptron:

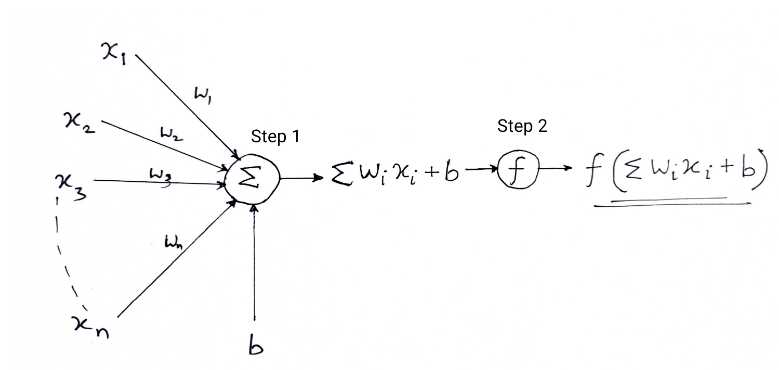
Rosenblatt’s perceptron is basically a binary classifier. The perceptron consists of 3 main parts:

* Input nodes or input layer: The input layer takes the initial data into the system for further processing. Each input node is associated with a numerical value. It can take any real value.
* Weights and bias: Weights control the signal (or the strength of the connection) between two neurons. In other words, a weight decides how much influence the input will have on the output. Biases, which are constant, are an additional input into the next layer that will always have the value of 1. Bias plays the same as the intercept in a linear equation.
* Activation function: The activation function determines whether the neuron will fire or not. At its simplest, the activation function is a step function, but based on the scenario, different activation functions can be used.

**Working of a Perceptron**

In the first step, all the input values are multiplied with their respective weights and added together. The result obtained is called weighted sum ∑wi\*xi, or stated differently,  x1\*w1 + x2\*w2 +…wn\*xn. This sum gives an appropriate representation of the inputs based on their importance. Additionally, a bias term *b* is added to this sum ∑wi\*xi + *b*. Bias serves as another model parameter (in addition to weights) that can be tuned to improve the model’s performance.

In the second step, an activation function *f* is applied over the above sum ∑wi\*xi + b to obtain output Y = *f*(∑wi\*xi + b). Depending upon the scenario and the activation function used, the Output is either binary {1, 0} or a continuous value.



(Often both these steps are represented as a single step in multi-layer perceptrons, here I have shown them as two different steps for better understanding)

Activation Functions:

A biological neuron only fires when a certain threshold is exceeded. Similarly, the artificial neuron will also only fire when the sum of the inputs (weighted sum) exceeds a certain threshold value, let’s say 0. Intuitively, we can think of a rule-based approach like this –

If  ∑wi\*xi + b > 0:

    output = 1

else:

    output = 0

Its graph will be something like this:

Chart, diagram, box and whisker chart

Description automatically generated

This is in fact the Unit Step (Threshold) activation function which was originally used by Rosenblatt. But as you can see, this function is discontinuous at 0, so it causes problems in mathematical computations. A smoother version of the above function is the sigmoid function. It outputs between 0 and 1. Another one is the Hyperbolic tangent(tan*h*) function, which produces the output between -1 and 1. Both sigmoid and tanh functions suffer from vanishing gradients problems. Nowadays, ReLU and Leaky ReLU are the most popularly used activation functions. They are comparatively stable over deep networks.

Importing some libraries –

from sklearn.datasets import make\_blobs

import matplotlib.pyplot as plt

import numpy as np

%matplotlib inline

Generating a dummy dataset using[*make\_blobs*](https://scikit-learn.org/stable/modules/generated/sklearn.datasets.make_blobs.html)functionality provided by scikit learn –

# Generate dataset

X, Y = make\_blobs(n\_features = 2, centers = 2, n\_samples = 1000, random\_state = 12)

# Visualize dataset

plt.figure(figsize = (6, 6))

plt.scatter(X[:, 0], X[:, 1], c = Y)

plt.title('Ground truth', fontsize = 18)

plt.show()

Chart, scatter chart

Description automatically generated

Let’s say the blue dots are 1s and the green dots are 0s. Using perceptron logic, we can create a decision boundary(hyperplane) for classification which separates different data points on the graph.

Before we proceed further, let’s add a bias term (ones) to the input vector –

# Add a bias to the input vector

X\_bias = np.ones([X.shape[0], 3])

X\_bias[:, 1:3] = X

The dataset will look something like this –

Text

Description automatically generated with medium confidence

Here each row of the above dataset represents the input vector (a datapoint). In order to create a decision boundary, we need to find out the appropriate weights. The weights are  ‘learned’ from the training using the below rule –

w = w + (expected — predicted) \* x

Perceptron w

It means that subtracting the estimated outcome from the ground truth and then multiplying this by the current input vector and adding old weights to it in order to obtain the new value of the weights. If our output is the same as the actual class, then the weights do not change. But if our estimation is different from the ground truth, then the weights increase or decrease accordingly. This is how the weights are progressively adjusted in each iteration.

We start by assigning arbitrary values to the weight vector, then we progressively adjust them in each iteration using the error and data at hand –

# initialize weights with random values

w = np.random.rand(3, 1)

print(w)

Output:

[[0.37547448]

 [0.00239401]

[0.18640939]]

Define the activation function of perceptron –

def activation\_func(z):

if z >= 1:

return 1

else:

return 0

Next, we apply the perceptron learning rule –

for \_ in range(100):

for i in range(X\_bias.shape[0]):

y = activation\_func(w.transpose().dot(X\_bias[i, :]))

# Update weights

w = w + ((Y[i] - y) \* X\_bias[i, :]).reshape(w.shape[0], 1)

It is not guaranteed that the weights will converge in one pass, so we feed all the training data into the perceptron algorithm 100 times while constantly applying the learning rule so that eventually we manage to obtain the optimal weights.

Now, that we have obtained the optimal weights, we predict the class for each datapoint using Y = *f(*∑wi\*xi + b) or Y = wT.x in vector form.

# predicting the class of the datapoints

result\_class = [activation\_func(w.transpose().dot(x)) for x in X\_bias]

Visualize the decision boundary and the predicted class labels –

# convert to unit vector

w = w/np.sqrt(w.transpose().dot(w))

# Visualize results

plt.figure(figsize = (6, 6))

plt.scatter(X[:, 0], X[:, 1], c = result\_class)

plt.plot([-10, -1], hyperplane([-10, -1], w), lw = 3, c = 'red')

plt.title('Perceptron classification with decision boundary')

plt.show()

Chart, scatter chart

Description automatically generated

You can compare the ground truth image with the predicted outcome image and see some points that are misclassified. If we calculate the accuracy, it comes to about 98%.

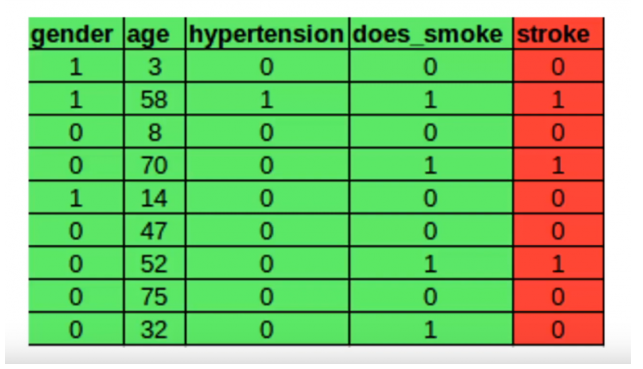
If you see, here our original data was fairly separated, so we are able to get such good accuracy. But this is not the case with real-world data. Using a single perceptron, we can only construct a linear decision boundary, so if the data is intermixed the perceptron algorithm will perform poorly. This is one of the limitations of the single perceptron model.

**Forward Propagation and Errors in a Neural Network:**

Introduction

If there is one area in data science that has led to the growth of Machine Learning and Artificial Intelligence in the last few years, it is Deep Learning. From research labs in universities with low success in the industry to powering every smart device on the planet – Deep Learning and Neural Networks have started a revolution. And the first step of training a neural network is Forward Propagation.

Before we start let’s have a look at a sample data set which will be used to train the neural network-



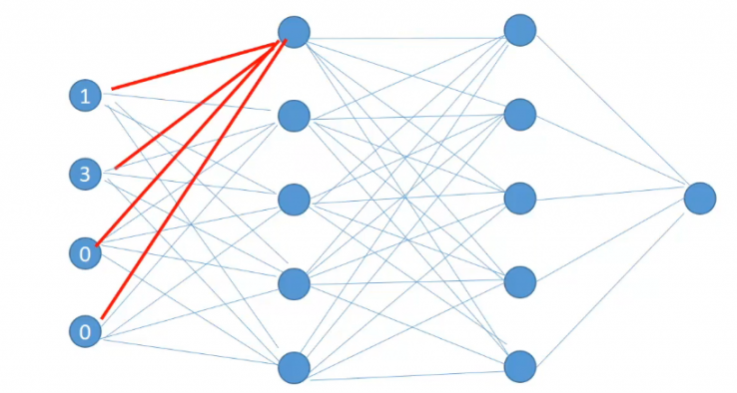
This data set has four independent features represented by green and one dependent feature represented by red. For the sake of simplicity let’s focus on the first row and try to understand forward propagation with this single row input. Each of the independent features in the first row will be passed to the input layer of the neural network to act as input.

Diagram

Description automatically generated

Now we start off the forward propagation by randomly initializing the weights of all neurons. These weights are depicted by the edges connecting two neurons. Hence the weights of a neuron can be more appropriately thought of as weights between two layers since edges connect two layers.

Now let’s talk about this first neuron in the first hidden layer.



This neuron receives input from all the neurons in the previous layer and combines it with its own set of weights represented by the red edges. And after applying some activation function, calculates the output known as Hidden Activation for this neuron. The output of the hidden activation will serve as input for the next layer neurons. Similarly, the hidden activations for all of the neurons in the layer are calculated.

So all the activations for the first hidden layer have been calculated. These activations will serve as inputs to the layer after them. Once the hidden activations for the last hidden layer are calculated, they are combined by a final set of weights between the last hidden layer and the output layer to produce an output for a single row observation.

A picture containing chart

Description automatically generated

These calculations of the first row features are 0.5 and the predicted value becomes 0.5 against the actual value of 0.

Table

Description automatically generated

Now if you notice that carefully we calculated the hidden activations for each of the neurons sequentially, that is one after the other. In fact, one optimization that we can make to the calculation is to calculate them in parallel. Since the calculation of neurons is independent of each other they can be computed in parallel easily. We can calculate the first hidden activation simultaneously-

Diagram

Description automatically generated

then calculate the activation of the second layer simultaneously,

A picture containing diagram

Description automatically generated

and then finally calculate the output.

Diagram

Description automatically generated with medium confidence

**This series of calculations which takes us from the input to output is called Forward Propagation.**

We will now understand the error generated during the predictions and what can be the possible reasons behind those errors.

**Errors in Neural Network**

So far we have seen how forward propagation helps us in calculating outputs. Let’s say for a particular row the actual target is 0 and the predicted target is 0.5. We can use this predicted value to calculate the error for a particular row. The kind of error that we have chosen here is Squared Error.

Schematic

Description automatically generated

There can be other error calculation formulas but for the sake of simplicity, we have chosen this one.

## What can be changed to reduce error?

## Now that we have an error what can be possible reasons for this and what can be changed to reduce errors? We know that the error is directly dependent upon two quantities- the actual values and the predicted values.

## Diagram Description automatically generated

## We cannot change the actual values so this is out of the possible suspects. Next, we know that the predicted values which are also the hidden activations are dependent upon 3 things-

## Inputs to the output layer i.e the hidden activation of the second layer

## Weights between the second hidden layer and the output

## And the activation function present at the output layer

## We cannot directly modify the second hidden layer activations since they’re calculated at each step. So inputs are also out. We can change both the activation functions and weights but we cannot iteratively change activation functions as it would change distributions during the training time. So activation function is also out of the question.

## Now, what remains are the weights between the second hidden layer and output. In fact not only can we change the weights between the second hidden layer and the output but also between the first and the second hidden layer and between the input layer and the first hidden layer. But we cannot change the weights in arbitrary amounts to check out what weights produce the lowest error as this would amount to an infinite amount of time to check what sort of weights produce the lowest error.

## How to reduce Errors?

## Interestingly what we can do is we can use a smart strategy to adjust the weights and reduce the errors. Specifically what we are interested is in two things-

## First is the amount of change in error on changing the weight by a small amount

## And second is the direction of that change.

## A picture containing graphical user interface Description automatically generated

## How does Backward Propagation Work in Neural Networks?

## Introduction

## Forward Propagation is the way to move from the Input layer (left) to the Output layer (right) in the neural network. The process of moving from the right to left i.e backward from the Output to the Input layer is called the Backward Propagation.

## Backward Propagation is the preferable method of adjusting or correcting the weights to reach the minimized loss function.

## 

## Table of Contents

## Setting up the Base

## Contribution of each Weight and Bias on the Error

## Matrix Form of the Backward Propagation

## Setting up the Base

## Let’s say we want to use the neural network to predict house prices. For our understanding purpose here, we will take a subset dummy dataset having four input variables and six observations here with the input having a dimension of 4\*5:

Table

Description automatically generated

The neural network for this subset data looks like below:

Diagram, schematic

Description automatically generated

## The architecture of the neural network is [4, 5, 1] with:

## 4 independent variables, Xs in the input layer

## 5 nodes in the hidden layer, and

## Since we have a regression problem at hand, we will have one node in the output layer.

## A Neural Network operates by:

## Initializing the weights with some random values, which are mostly between 0 and 1.

## Compute the output to calculate the loss or the error term.

## Then, adjust the weights so that to minimize the loss.

## We repeat these three steps until have reached the optimum solution of the minimum loss function or exhausted the pre-defined epochs (i.e. the number of iterations).

## Now, the computation graph after applying the sigmoid activation function is:

## Diagram Description automatically generated with medium confidence

## Building on this, the first step in Backward Propagation to calculate the error. In our regression problem, we shall take the loss function = (Y-Y^)2/2 where Y is actual values and Y^ is predicted values. For simplicity, replacing Y^ with O, so the error E becomes = (Y-O)2/2.

## Our goal is to minimize the error that is clearly dependent on Y, which is the actual observed values, and on the output, which is further is dependent on the:

## input values

## coefficients or betas of the input variables

## biases, the [activation function](https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6?gi=2074f5569281), and

## [Optimizers](https://www.analyticsvidhya.com/blog/2021/04/is-gradient-descent-sufficient-for-neural-network/)

## Now, we can neither change the input variables nor the actual Y values however, we can change the other factors. The activation function and the optimizers are the tuning parameters – and we can change these based on our requirement.

## The other two factors: the coefficients or betas of the input variables (Wis) and the biases (bho, bih) are updated using the Gradient descent algorithm with the following equation:

## Wnew = Wold – (α \* dE/dW)

## where,

## Wnew = the new weight of Xi

## Wold = the old weight of the Xi

## α = learning rate

## dE/dW is the partial derivative of the error for each of the Xs. It is the rate of change of the error to the change in weight.

## In the backward propagation, we adjust these weights or the betas in the output. The weights and biases between the respective input, hidden and output layers we have here are Wih, bih, Who, and bho:

## Wih: weight between the input and the hidden layer

## bih: bias between the input and the hidden layer

## Who: weight between the hidden and the output layer

## bho: bias between the hidden and the output layer

## In the first iteration, we randomly initialize the weights. In the second iteration, we change the weights of the hidden layer that is closest to the output layer. In this case, we go from the output layer, hidden layer, and then to the input layer.

## Contribution of each Weight and Bias on the Error

## Now, we have to calculate how much each of these weights (Wis) and biases (bis) contribute to the error term. For this, we need to calculate the rate of change of error to the respective weights and bias parameters.

## In other words, we need to compute the terms: dE/dWih, dE/dbih, dE/dWho, and dE/dbho. This is not a direct task. It is a series of steps involving the [Chain Rule](https://ml-cheatsheet.readthedocs.io/en/latest/backpropagation.html).

## The weight, Who, between the hidden and the output layer:

## The error E is not directly dependent on the Who:

## The error term is dependent on the Output O

## Output O is further dependent on Z2, and

## Z2 is dependent on Who

## Therefore we employ the chain rule to compute the rate of change in error to the change in weight Who and it becomes:

## dE/dWho = dE/dO \* dO/dZ2 \* dZ2/dWho

## Now, we take the partial derivatives of each of these individual terms:

## E = (Y-O)2/2.

## The partial derivative of error with respect to Output is: dE/dO = 2\*(Y-O)\*(-1)/2 = (O-Y)

## The partial derivative of Output with respect to Z2, as output O = Sigmoid of Z2 and the derivative of sigmoid is:

## dO/dZ2 = sigmoid(Z2) \*(1-sigmoid(Z2)) = O\*(1-O)

## The partial derivative of Z2 with respect to Who is:

## dZ2/dWho = d(WhoT \* h1 + bh0)/dWho

## dZ2/dWho = d(WhoT \* h1)/dWho + d(bho/Who) = h1 + 0 = h1

## Therefore, dE/dWho = dE/dO \* dO/dZ2 \* dZ2/dWho becomes:

## dE/dWho = (O-Y) \* O\*(1-O) \* h1

## Similarly, we will calculate the contribution for each of the other parameters in this manner.

## For the bias, bho, between the hidden and the output layer:

## dE/dbho = dE/dO \* dO/dZ2 \* dZ2/dbho

## dE/dbho = (O-Y) \* O\*(1-O) \* 1

## The weight, Wih, between the input and the hidden layer:

## From the above graph we can see that the terms are dependent as below:

## Error term is dependent on the Output O

## Output O is dependent on Z2

## Z2 this time is dependent on h1

## h1 is dependent on Z1, and

## Z1 is dependent on Wih

## dE/dWih = dE/dO \* dO/dZ2 \* dZ2/dh1 \* dh1/dZ1 \* dZ1/dWih

## So, this time, apart from the initial above dE/dO, dO/dZ2, we have the partial derivatives as follow:

## The partial derivative of Z2 with respect to h1 is:

## dZ2/dh1 = d(WhoT \* h1 + bho)/dh1

## dZ2/dh1 = d(WhoT \* h1)/dh1 + d(bho/h1) = Who + 0 = Who

## The partial derivative of h1 with respect to Z1, as h1 = Sigmoid of Z1 and the derivative of sigmoid is:

## dh1/dZ1 = sigmoid(Z1) \*(1-sigmoid(Z1)) = h1\* (1 – h1)

## The partial derivative of Z1 with respect to Wih is: X

## dZ1/dWih = d(WihT \* X + bih)/dWih

## dZ1/dWih = d(WihT \* X)/dWih + d(bih/Wih) = X + 0 = X

## Hence, the equation after plugging the partial derivative value is:

## dE/dWih = dE/dO \* dO/dZ2 \* dZ2/dh1 \* dh1/dZ1 \* dZ1/dWih

## dE/dWih = (O-Y) \* O\*(1-O) \* Who \* h1(1-h1) \* X

## The bias, bih, between the input and the hidden layer:

## dE/dbih = dE/dO \* dO/dZ2 \* dZ2/dh1 \* dh1/dZ1 \* dZ1/dbih

## dE/dbih = (O-Y) \* O\*(1-O) \* Who \* h1(1-h1) \* 1

## Now, that we have computed these terms we can update the parameters using the following respective update equations:

## Wih = Wih – (α \* dE/dWih)

## bih = bih – (α \* dE/dbih)

## Who = Who – (α \* dE/dWho)

## bho = bho – (α \* dE/dbho)

## Now, moving to another method to perform backward propagation …

## Matrix Form of the Backward Propagation

## The backward propagation can also be solved in the matrix form. The computation graph for the structure along with the matrix dimensions is:

## A close-up of a whiteboard Description automatically generated with low confidence

## Z1 = WihT \* X + bih

## where,

## Wih is the weight matrix between the input and the hidden layer with the dimension of 4\*5

## WihT, is the transpose of Wih, having shape 5\*4

## X is the input variables having dimension 4\*5, and

## bih is a bias term, has a single value here as considering the same for all the neurons.

## Z2 = WhoT \* h1 + bho

## where,

## Who is the weight matrix between the hidden and the output layer with shape 5\*1

## WhoT, is the transpose of Who having a dimension of 1\*5

## h1 is the result after the applying activation function on the outcome from the hidden layer with a shape of 5\*5, and

## bho is the bias term,  has a single value here as considering the same for all the neurons.

## To summarize, the four equations of the rate of change of error with the different parameters are:

## dE/dWho = dE/dO \* dO/dZ2 \* dZ2/dWho= (O-Y) \* O\*(1-O) \* h1

## dE/dbho = dE/dO \* dO/dZ2 \* dZ2/dbho= (O-Y) \* O\*(1-O) \* 1

## dE/dWih = dE/dO \* dO/dZ2 \* dZ2/dh1 \* dh1/dZ1 \* dZ1/dWih = (O-Y) \* O\*(1-O) \* Who \* h1(1-h1) \* X

## dE/dbih = dE/dO \* dO/dZ2 \* dZ2/dh1 \* dh1/dZ1 \* dZ1/dbih = (O-Y) \* O\*(1-O) \* Who \* h1(1-h1) \* 1

## Now, lets’ see how we can perform matrix multiplication on each of these equations. For the weight matrix between the hidden and the output layer, Who.

## Let us understand how the shape of this Who must be similar to that of the shape of dE/dWho, which is to used to update the weight in the following equation:

## Who = Who – (α \* dE/dWho)

## We saw above that dE/dWho is computed using the chain rule and is of the result:

## dE/dWho = dE/dO \* dO/dZ2 \* dZ2/dWho

## dE/dWho = (O-Y) \* O\*(1-O) \* h1

## Breaking the individual components of this above equation we see each part’s dimension:

## dE/dO = (O-Y) as both O and Y have the same shape of 1\*5. Hence, dE/dO is of dimension 1\*5.

## dO/dZ2 = O\*(1-O) having a shape of 1\*5, and

## dZ2/dWho = h1, which is of the shape 5\*5

## Now, performing matrix multiplication on this equation. As we know, matrix multiplication can be done when the number of columns of the first matrix must be equal to the number of rows of the second matrix. Where this matrix multiplication rule defies, we will take the transpose of one of the matrices to conduct the multiplication.

## On applying this our equation takes the form of:

## dE/dWho = dZ2/dWho . [dE/dO \* dO/dZ2] T

## dE/dWho = (5X5) . [(1X5) \*(1X5)]T

## dE/dWho = (5X5) . (5X1) = 5X1

## Therefore, the shape of dE/dWho 5\*1 is the same as that of Who 5\*1 which will be updated using the Gradient Descent update equation.

## In the same manner, we can find perform the backward propagation for the other parameters using matrix multiplication and the respective equations will be:

## dE/dWho = dZ2/dWho . [dE/dO \* dO/dZ2] T

## dE/dbho = dZ2/dbho . [dE/dO \* dO/dZ2 ]T

## dE/dWih = dZ1/dWih . [dh1/dZ1 \* dZ2/dh1. (dE/dO \* dO/dZ2)] T

## dE/dbih = dZ1/dbih . [dh1/dZ1 \* dZ2/dh1. (dE/dO \* dO/dZ2)] T

## Where, (.) dot is the dot product and \* is the element wise product.

## Endnotes

## To summarize, as promised, below is a very cool gif that shows how backward propagation operates in reaching to the solution by minimizing the loss function or error:

## 

## Backward Propagation is the preferred method for adjusting the weights and biases since it is faster to converge as we move from output to the hidden layer. Here, we change the weights of the hidden layer that is closest to the output layer, re-calculate the loss and if further need to reduce the error then repeat the entire process and in that order move towards the input layer.

## Whereas in the forward propagation, the pecking order is from the input layer, hidden, and then to the output layer which takes more time to converge to the optimum solution of the minimum loss function.

## Convolutional Neural Networks (CNN):

## Introduction

## In the past few decades, Deep Learning has proved to be a very powerful tool because of its ability to handle large amounts of data. The interest to use hidden layers has surpassed traditional techniques, especially in pattern recognition. One of the most popular deep neural networks is Convolutional Neural Networks.

## Diagram Description automatically generated

## Since the 1950s, the early days of AI, researchers have struggled to make a system that can understand visual data. In the following years, this field came to be known as Computer Vision. In 2012, computer vision took a quantum leap when a group of researchers from the University of Toronto developed an AI model that surpassed the best image recognition algorithms and that too by a large margin.

## The AI system, which became known as AlexNet (named after its main creator, Alex Krizhevsky), won the 2012 ImageNet computer vision contest with an amazing 85 percent accuracy. The runner-up scored a modest 74 percent on the test.

## At the heart of AlexNet was Convolutional Neural Networks a special type of neural network that roughly imitates human vision. Over the years CNNs have become a very important part of many Computer Vision applications. So let’s take a look at the workings of CNNs.

## Background of CNNs

## CNN’s were first developed and used around the 1980s. The most that a CNN could do at that time was recognize handwritten digits. It was mostly used in the postal sectors to read zip codes, pin codes, etc. The important thing to remember about any deep learning model is that it requires a large amount of data to train and also requires a lot of computing resources. This was a major drawback for CNNs at that period and hence CNNs were only limited to the postal sectors and it failed to enter the world of machine learning.

## Diagram Description automatically generated

## In 2012 Alex Krizhevsky realized that it was time to bring back the branch of deep learning that uses multi-layered neural networks. The availability of large sets of data, to be more specific ImageNet datasets with millions of labeled images and an abundance of computing resources enabled researchers to revive CNNs.

## What exactly is a CNN?

## In [deep learning](https://en.wikipedia.org/wiki/Deep_learning), a **convolutional neural network** (**CNN/ConvNet**) is a class of [deep neural networks](https://en.wikipedia.org/wiki/Deep_neural_network), most commonly applied to analyze visual imagery. Now when we think of a neural network we think about matrix multiplications but that is not the case with ConvNet. It uses a special technique called Convolution. Now in mathematics **convolution** is a mathematical operation on two functions that produces a third function that expresses how the shape of one is modified by the other.

## Diagram Description automatically generated

## But we don’t really need to go behind the mathematics part to understand what a CNN is or how it works.

## Bottom line is that the role of the ConvNet is to reduce the images into a form that is easier to process, without losing features that are critical for getting a good prediction.

## How does it work?

## Before we go to the working of CNN’s let’s cover the basics such as what is an image and how is it represented. An RGB image is nothing but a matrix of pixel values having three planes whereas a grayscale image is the same but it has a single plane. Take a look at this image to understand more.

## Diagram, table Description automatically generated

## For simplicity, let’s stick with grayscale images as we try to understand how CNNs work.

## Diagram, engineering drawing Description automatically generated

## The above image shows what a convolution is. We take a filter/kernel(3×3 matrix) and apply it to the input image to get the convolved feature. This convolved feature is passed on to the next layer.

## A picture containing diagram Description automatically generated

## In the case of RGB color, channel take a look at this animation to understand its working

## Calendar Description automatically generated

## Convolutional neural networks are composed of multiple layers of artificial neurons. Artificial neurons, a rough imitation of their biological counterparts, are mathematical functions that calculate the weighted sum of multiple inputs and outputs an activation value. When you input an image in a ConvNet, each layer generates several activation functions that are passed on to the next layer.

## The first layer usually extracts basic features such as horizontal or diagonal edges. This output is passed on to the next layer which detects more complex features such as corners or combinational edges. As we move deeper into the network it can identify even more complex features such as objects, faces, etc.

## A picture containing text, building Description automatically generated

## Based on the activation map of the final convolution layer, the classification layer outputs a set of confidence scores (values between 0 and 1) that specify how likely the image is to belong to a “class.” For instance, if you have a ConvNet that detects cats, dogs, and horses, the output of the final layer is the possibility that the input image contains any of those animals.

## Diagram Description automatically generated

## What’s a pooling layer?

## Similar to the Convolutional Layer, the Pooling layer is responsible for reducing the spatial size of the Convolved Feature. This is to **decrease the computational power required to process the data** by reducing the dimensions. There are two types of pooling average pooling and max pooling. I’ve only had experience with Max Pooling so far I haven’t faced any difficulties.

## A screenshot of a game Description automatically generated with low confidence

## So what we do in Max Pooling is we find the maximum value of a pixel from a portion of the image covered by the kernel. Max Pooling also performs as a**Noise Suppressant**. It discards the noisy activations altogether and also performs de-noising along with dimensionality reduction.

## Diagram Description automatically generated

## On the other hand, **Average Pooling**returns the **average of all the values**from the portion of the image covered by the Kernel. Average Pooling simply performs dimensionality reduction as a noise suppressing mechanism. Hence, we can say that **Max Pooling performs a lot better than Average Pooling**.