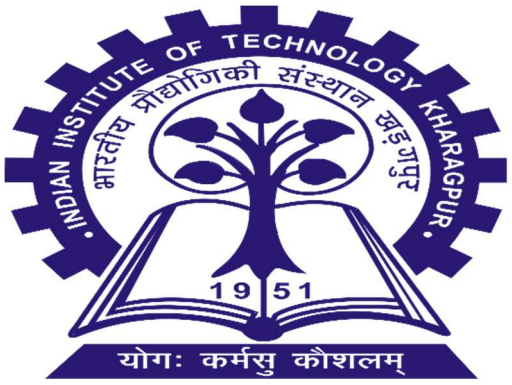
Conversion for Sign Language to Text



IIT KHARAGPUR
Department of Mathematics
Kharagpur, West Bengal India

Submitted by:
Swarnim Mishra
23MA60R26

Acknowledgements

I am profoundly grateful to IIT Kharagpur for granting me the opportunity to present this seminar. I extend my special thanks to Dr. Pratima Panigrahi, our seminar coordinator, and a Professor in the Department of Mathematics at IIT Kharagpur, West Bengal, India.

Additionally, I express my gratitude to our esteemed professors and mentors for their guidance and support throughout this endeavour.

Furthermore, I extend my heartfelt appreciation to online resources such as Wikipedia, YouTube, and Online articles for their substantial contributions, which significantly facilitated the work for aspiring individuals like me.

Sincerely,

Swarnim Mishra
Department of Mathematics
IIT KHARAGPUR,
Kharagpur-721302,
West Bengal India
Email: swarnimmishra006@gmail.com

Abstract of Report

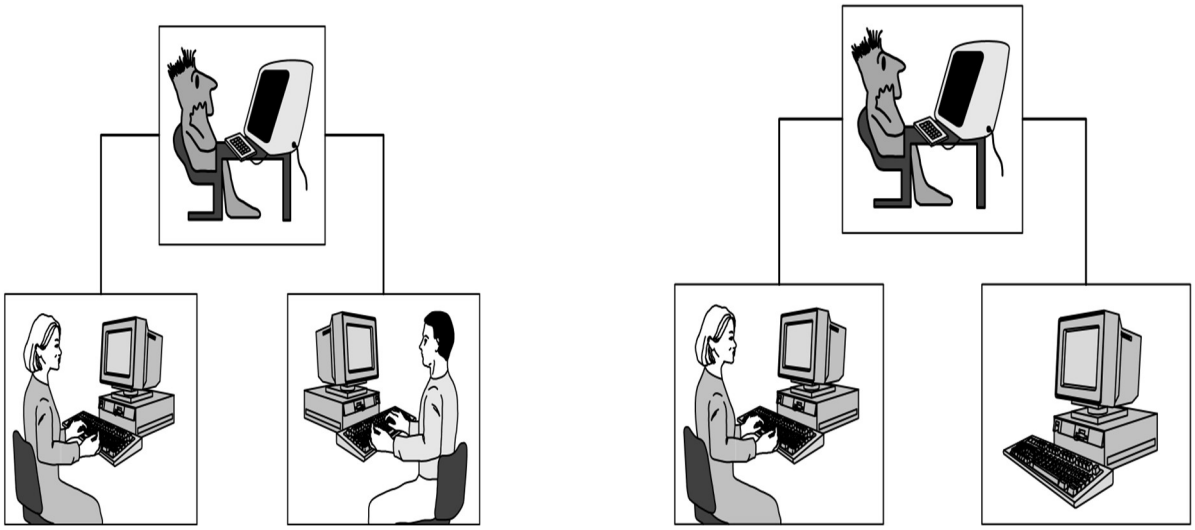
This report addresses the significant communication challenges faced by millions of individuals unable to speak or hear, who rely on sign language for interaction. However, the formidable hurdle of learning sign language necessitates the exploration of alternative solutions. The report focuses on the question of how to effectively convert sign language into text, and it highlights neural networks as the key solution. It outlines a structured approach: an initial exploration of mathematical numbers, followed by the intricate relationship between numbers and the sign alphabet, culminating in a flagship project dedicated to converting sign language into text with precision and efficiency. Sign language, one of humanity's oldest and most instinctive forms of communication, deserves to be universally accessible. However, the scarcity of sign language interpreters and the lack of sign language knowledge among the broader population necessitate innovative solutions. The report introduces a real-time method that leverages neural networks to facilitate fingerspelling-based American Sign Language.

Table of Contents

1. Introduction

1.1 The Imitation Game

The imitation game proposed by Turing originally included two phases. In the first phase, shown in Figure 1.1, the interrogator, a man, and a woman are each placed in separate rooms and can communicate only via a neutral medium such as a remote terminal. The interrogator's objective is to work out who is the man and who is the woman by questioning them. The rules of the game are that the man should attempt to deceive the interrogator that he is the woman, while the woman has to convince the interrogator that she is the woman. In the second phase of the game, shown in Figure, the man is replaced by a computer programmed to deceive the interrogator as the man did. It would even be programmed to make mistakes and provide fuzzy answers in the way a human would. If the computer can fool the interrogator as often as the man did, we may say this computer has passed the intelligent behaviour test.[1]



1.2 Intuitive Example

On the surface, a machine recognizing handwritten digits may not seem particularly impressive. After all, you know how to identify digits, and I bet you do not even find it very hard. For example, you can tell instantly that these are all images of the digit three:

Each three is drawn differently, so the light-sensitive cells in your eye that fire are different for each, but something in that crazy smart visual cortex of yours resolves all these as representing the same idea, while recognizing images of other numbers as their own distinct ideas.

But if I told you to sit down and write a program, that takes in a grid of 28x28 pixels, and outputs a single number between 0 and 9, the task goes from comically trivial to dauntingly difficult. Somehow identifying digits is incredibly easy for your brain to do, but almost impossible to describe *how* to do. The traditional methods of computer programming, with if statements and for loops and classes and objects and functions, just do not seem suitable to tackle this problem.

But what if we could write a program that mimics the structure of your brain? That is the idea behind neural networks.

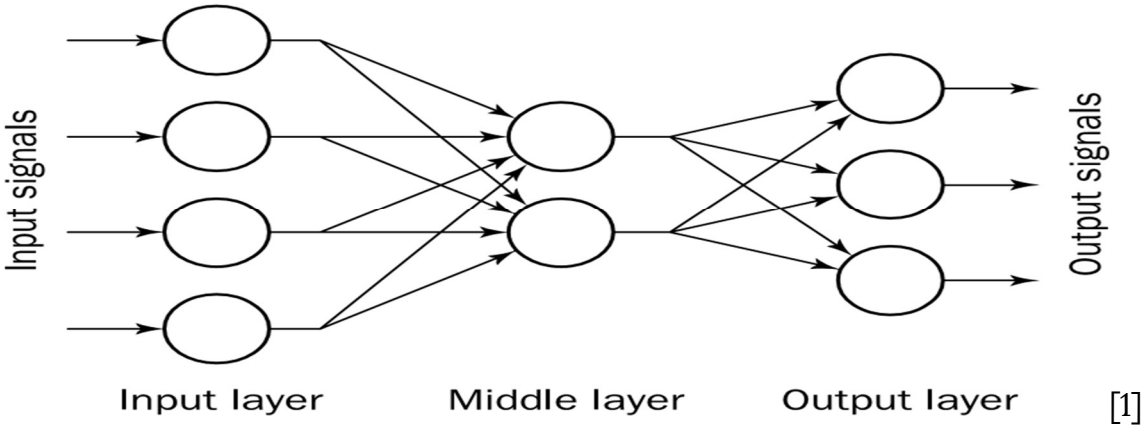
The hope is that by writing brain-inspired software, we might be able to create programs that tackle the kinds of fuzzy and difficult-to-reason-about problems that your mind is so good at solving.

Moreover, just as we learn by seeing many examples, the “learning” part of machine learning comes from the fact that we never give the program any specific instructions for how to identify digits. Instead, we’ll show it many examples of hand-drawn digits together with labels for what they should be and leave it up to the computer to adapt the network based on each new example.[4]

2. Key Words and Definitions

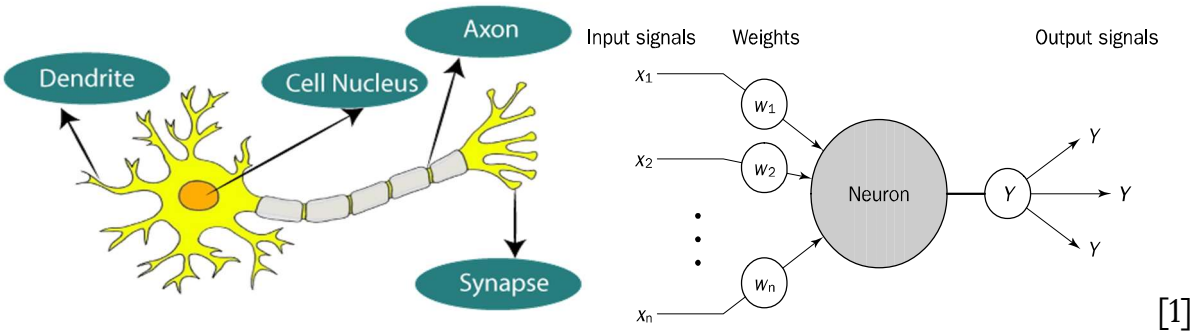
2.1 Artificial Neural Networks:

Artificial Neural Network is a connection of neurons, replicating the structure of human brain. Each connection of neuron transfers information to another neuron. Inputs are fed into first layer of neurons which processes it and transfers to another layer of neurons called as hidden layers. After processing of information through multiple layers of hidden layers, information is passed to final output layer.



[1]

The given figure illustrates the typical diagram of Biological Neural Network and Artificial Neural Network.



[1]

Dendrites from Biological Neural Network represent inputs in Artificial Neural Networks, cell nucleus represents Nodes, synapse represents Weights, and Axon represents Output.

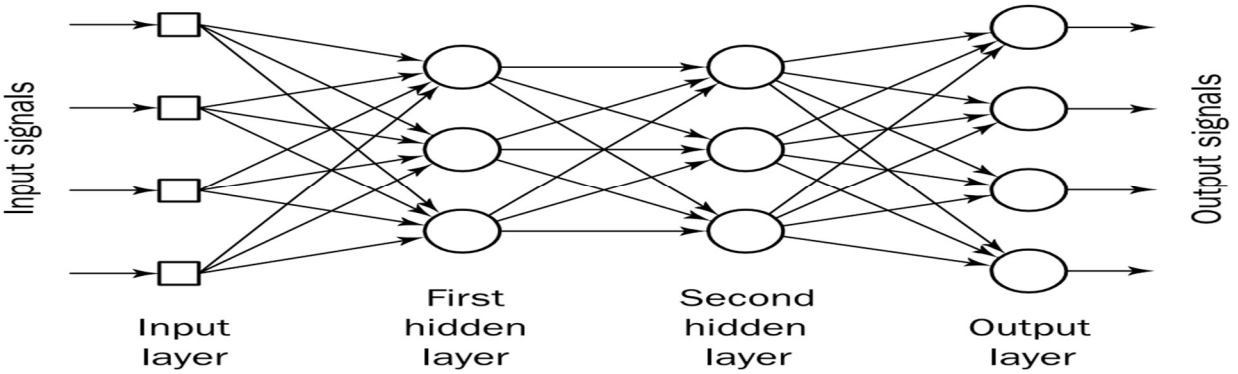
The neuron as a simple computing element

A neuron receives several signals from its input links, computes a new activation level and sends it as an output signal through the output links. The input signal can be raw data or outputs of other neurons. The output signal can be either a final solution to the problem or an input to other neurons. Above Figure shows a typical neuron.

.

2.2 The architecture of an artificial neural network:

To understand the concept of the architecture of an artificial neural network, we have to understand what a neural network consists of. To define a neural network that consists of a large number of artificial neurons, which are termed units arranged in a sequence of layers. Let's us look at various types of layers available in an artificial neural network.
Artificial Neural Network primarily consists of three layers:



Input Layer:
As the name suggests, it accepts inputs in several different formats provided by the programmer.
Hidden Layer:
The hidden layer presents in-between input and output layers. It performs all the calculations to find hidden features and patterns.
Output Layer:
The input goes through a series of transformations using the hidden layer, which finally results in output that is conveyed using this layer.

The artificial neural network takes input and computes the weighted sum of the inputs and includes a bias. This computation is represented in the form of a transfer function.

It determines weighted total is passed as an input to an activation function to produce the output. Activation functions choose whether a node should fire or not. Only those who are fired make it to the output layer. There are distinctive activation functions available that can be applied upon the sort of task we are performing.

Artificial neural nets model the human brain
An artificial neural network consists of a number of very simple and highly interconnected processors, also called neurons, which are analogous to the biological neurons in the brain. The neurons are connected by weighted links passing signals from one neuron to another. Each neuron receives a number of input signals through its connections; however, it never

produces more than a single output signal. The output signal is transmitted through the neuron's outgoing connection (corresponding to the biological axon). The outgoing connection, in turn, splits into a number of branches that transmit the same signal (the signal is not divided among these branches in any way). The outgoing branches terminate at the incoming connections of other neurons in the network.

Need of hidden layer
Each layer in a multilayer neural network has its own specific function. The input layer accepts input signals from the outside world and redistributes these signals to all neurons in the hidden layer. Actually, the input layer rarely includes computing neurons, and thus does not process input patterns. The output layer accepts output signals, or in other words a stimulus pattern, from the hidden layer and establishes the output pattern of the entire network. Neurons in the hidden layer detect the features; the weights of the neurons represent the features hidden in the input patterns. These features are then used by the output layer in determining the output pattern. With one hidden layer, we can represent any continuous function of the input signals, and with two hidden layers even discontinuous functions can be represented.

2.3 Different types of neural network

1)Feedforward Neural Network (FNN)
Basic Neural Network: It consists of input, hidden, and output layers. Information moves in only one direction from the input nodes to the output nodes.
Used for: Classification and regression tasks.

2. Convolutional Neural Network (CNN)
Convolutional Layers: Uses convolutional layers for detecting patterns and features in images.
Pooling Layers: Utilizes pooling layers to reduce the spatial dimensions of the output volume.
Used for: Image and video recognition, image generation.

3. Recurrent Neural Network (RNN):
Looping Architecture: Contains loops to allow information persistence, making them suitable for sequential data.
Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) Variants of RNNs designed to capture long-term dependencies.
Used for: Natural language processing, speech recognition, time series prediction.

4). Generative Adversarial Networks (GANs)
Two Networks: Comprises a generator that creates fake data samples and a discriminator that distinguishes between real and fake samples.
Used for Generating new, synthetic data samples; image-to-image translation tasks

5) Long Short-Term Memory (LSTM) Networks:
Memory Cells: Contains memory cells that can store information over long periods, addressing vanishing gradient problem.
Used for: Sequence prediction tasks requiring long-term dependencies..

2.4 Convolution Neural Network:

Unlike regular Neural Networks, in the layers of CNN, the neurons are arranged in 3 dimensions: width, height, depth. The neurons in a layer will only be connected to a small region of the layer (window size) before it, instead of all of the neurons in a fully connected manner. Moreover, the final output layer would have dimensions (number of classes), because by the end of the CNN architecture we will reduce the full image into a single vector of class scores.

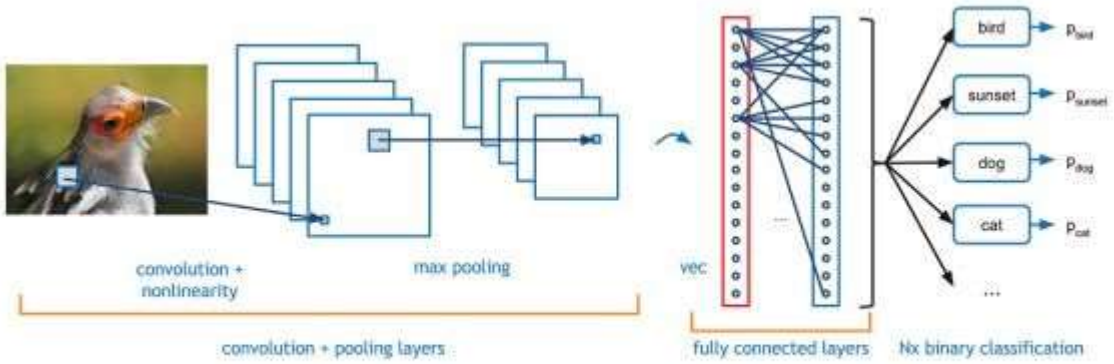


Figure 5.2: Convolution neural networks

1) Convolution Layer: In convolution layer we take a small window size [typically of length 5*5] that extends to the depth of the input matrix. The layer consists of learnable filters of window size. During every iteration we slid the window by stride size [typically 1] and compute the dot product of filter entries and input values at a given position. As we continue this process well create a 2-Dimensional activation matrix that gives the response of that matrix at every spatial position. That is, the network will learn filters that activate when they see some type of visual feature such as an edge of some orientation or a blotch of some colour

2)Pooling Layer: We use pooling layer to decrease the size of activation matrix and ultimately reduce the learnable parameters. There are two type of pooling.

a) **Max Pooling:** In max pooling we take a window size [for example window of size 2*2], and only take the maximum of 4 values. Well lid this window and continue this process, so well finally get a activation matrix half of its original Size.

b) **Average Pooling:** In average pooling we take average of all values in a window.

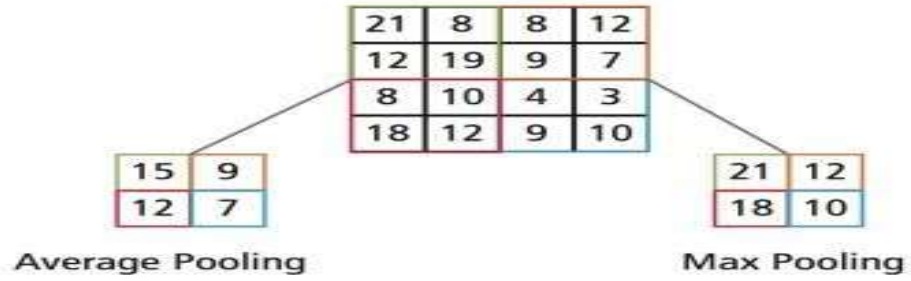


Figure 5.3: Types of pooling

2. Fully Connected Layer: In convolution layer neurons are connected only to a local region, while in a fully connected region, well connect the all the inputs to neurons.

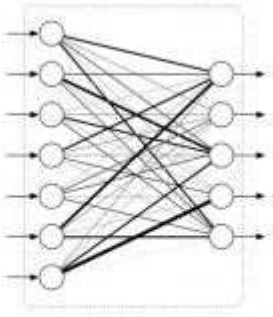


Figure 5.4: Fully Connected Layer

3. Final Output Layer: After getting values from fully connected layer, well connect them to final layer of neurons [having count equal to total number of classes], that will predict the probability of each image to be in different classes.

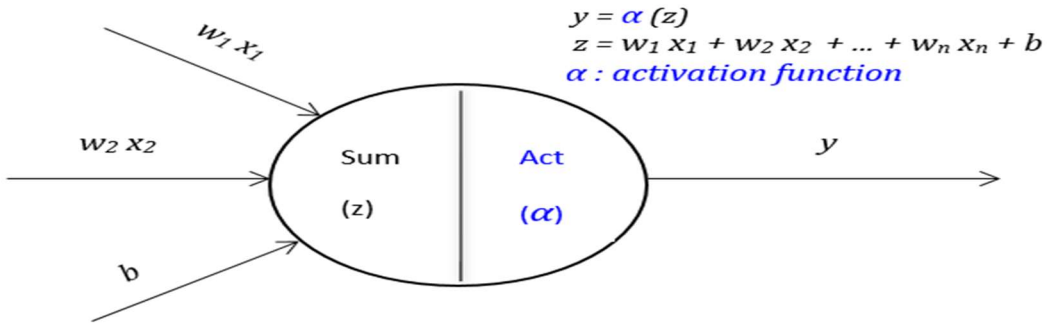
4. Activation function

Activation functions are functions used in a neural network to compute the weighted sum of inputs and biases, which is in turn used to decide whether a neuron can be activated or not. It manipulates the presented data and produces an output for the neural network that contains the parameters in the data. The activation functions are also referred to as ***transfer functions*** in some literature. These can either be linear or nonlinear depending on the function it represents and is used to control the output of neural networks across different domains. For a linear model, a linear mapping of an input function to output is performed in the hidden layers before the final prediction for each label is given.

The input vector x transformation is given by $f(x) = w^T x + b$ where, x = input, w = weight, and b = bias.

Linear results are produced from the mappings of the above equation and the need for the activation function arises here, first to convert these linear outputs into non-linear output for further computation, and then to learn the patterns in the data.
The output of these models is given by $y = (w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b)$

These outputs of each layer are fed into the next subsequent layer for multilayered networks until the final output is obtained, but they are linear by default.
The expected output is said to determine the type of activation function that has to be deployed in a given network.
However, since the outputs are linear in nature, the nonlinear activation functions are required to convert these linear inputs to non-linear outputs. These transfer functions, applied to the outputs of the linear models to produce the transformed non-linear outputs are ready for further processing. The non-linear output after the application of the activation function is given by $y = \alpha\,(w_1 x_1 + w_2 x_2 + \ldots + w_n x_n + b)$
where α is the activation function.



Need of Non-linear activation function
A neural network without an activation function is essentially just a linear regression model. The activation function does the non-linear transformation to the input making it capable to learn and perform more complex tasks.

Good Activation Function
A proper choice must be made in choosing the activation function to improve the results in neural network computing. All activation functions must be monotonic, differentiable, and quickly converging with respect to the weights for optimization purposes.

Types of Activation Functions

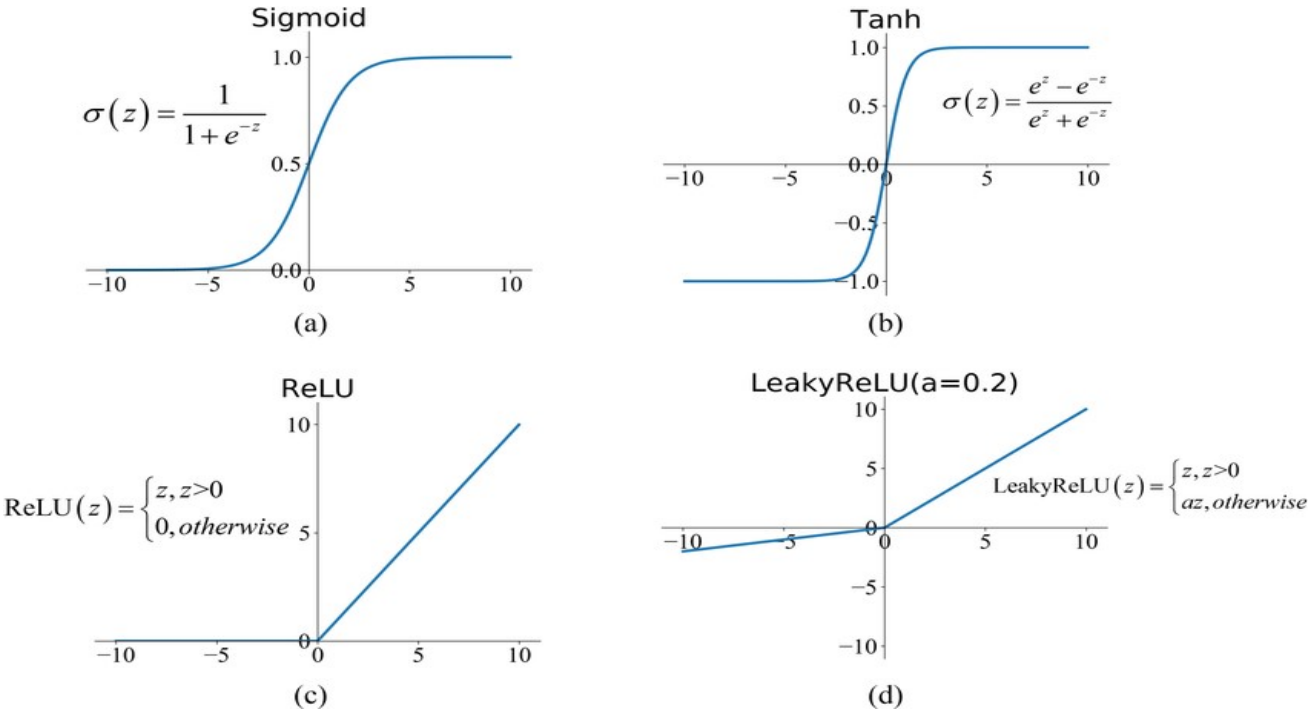
1 Linear Activation Functions
A linear function is also known as a straight- function where the activation is proportional to the input i.e., the weighted sum from neurons. The problem with this activation is that it cannot be defined in a specific range. Applying this function in all the nodes makes the activation function work like linear regression.

Another issue is the gradient descent when differentiation is done, it has a constant output which is not good because during backpropagation the rate of change of error is constant that can ruin the output and the logic of backpropagation.

2 Non-Linear Activation Functions

The non-linear functions are known to be the most used activation functions. It makes it easy for a neural network model to adapt with a variety of data and to differentiate between the outcomes.

These functions are mainly divided basis on their range or curves:



a) Sigmoid Activation Functions

Sigmoid takes a real value as the input and outputs another value between 0 and 1. The sigmoid activation function translates the input ranged in (-∞, ∞) to the range in (0,1)

Uses: Usually used in output layer of a binary classification, where result is either 0 or 1, as value for sigmoid function lies between 0 and 1 only so, result can be predicted easily to be **1** if value is greater than **0.5** and **0** otherwise

If S(x) is sigmoid function, then its few properties are.

$$S'(x) = \frac{e^x}{(1+e^x)^2} = S(x)\,(1-S(x))\,,$$

$$S''(x) = \frac{e^x(1-e^x)}{(1+e^x)^3} = S(x)(1-S(x))(1-2S(x))$$

$$S(x) = \frac{e^x}{1+e^x} = \frac{1}{1+e^{-x}}, \quad x \in (-\infty, \infty)$$

$$= \frac{1}{2} + \frac{1}{2}\tanh\left(\frac{x}{2}\right), \quad x \in (-\infty, \infty).$$ [4]

b) Tanh Activation Functions

The tanh function is just another possible function that can be used as a non-linear activation function between layers of a neural network. It shares a few things in common with the sigmoid activation function.

Unlike a sigmoid function that will map input values between 0 and 1, the Tanh will map values between -1 and 1. Like the sigmoid function, one of the interesting properties of the tanh function is that the derivative of tanh can be expressed in terms of the function itself. The activation that works almost always better than sigmoid function is Tanh function also known as **Tangent Hyperbolic function**. It's actually mathematically shifted version of the sigmoid function. Both are similar and can be derived from each other.

Usually used in hidden layers of a neural network as it's values lies between **-1 to 1** hence the mean for the hidden layer comes out be 0 or very close to it, hence helps in *centering the data* by bringing mean close to 0. This makes learning for the next layer much easier.

b) ReLU Activation Functions

It Stands for *Rectified linear unit*. It is the most widely used activation function. Chiefly implemented in *hidden layers* of Neural network

The formula is deceptively simple: $max(0,z)$. Despite its name, Rectified Linear Units, it's not linear and provides the same benefits as Sigmoid but with better performance.

Nature is non-linear, which means we can easily backpropagate the errors and have multiple layers of neurons being activated by the ReLU function.

ReLu is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations. At a time only a few neurons are activated making the network sparse making it efficient and easy for computation.

In simple words, RELU learns *much faster* than sigmoid and Tanh function.

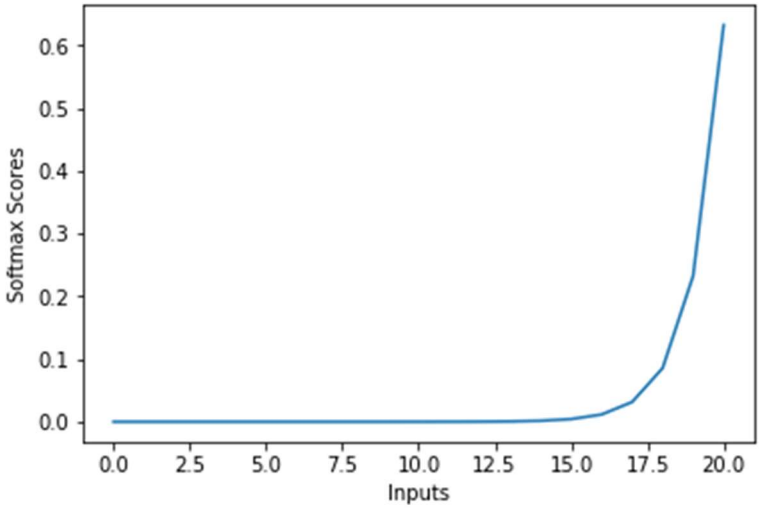
d) Leaky ReLU

Leaky ReLU is a variant of ReLU. Instead of being 0 when $z<0$, a leaky ReLU allows a small, non-zero, constant gradient α (normally, $\alpha=0.01$). However, the consistency of the benefit across tasks is presently unclear. Leaky ReLUs attempt to fix the "dying ReLU" problem..

e) SoftMax Activation Functions

SoftMax function calculates the probabilities distribution of the event over 'n' different events. In a general way, this function will calculate the probabilities of each target class over all possible target classes. Later the calculated probabilities will help determine the target class for the given inputs.

SoftMax Function



[4]

The SoftMax function is also a type of sigmoid function but is handy when we are trying to handle multi- class classification problems. The SoftMax function would squeeze the outputs for each class between 0 and 1 and would also divide by the sum of the outputs. If your output is for multi-class classification then, SoftMax is very useful to predict the probabilities of each class.

Choice of Activation Function in a Neural Network

Specifically, it depends on the problem type and the value range of the expected output. For example, to predict values that are larger than 1, tanh or sigmoid are not suitable to be used in the output layer, instead, ReLU can be used. On the other hand, if the output values must be in the range (0,1) or (-1, 1) then ReLU is not a good choice, and sigmoid or tanh can be used here. While performing a classification task and using the neural network to predict a probability distribution over the mutually exclusive class labels, the softmax activation function should be used in the last layer. However, regarding the hidden layers, as a rule of thumb, use ReLU as an activation for these layers.

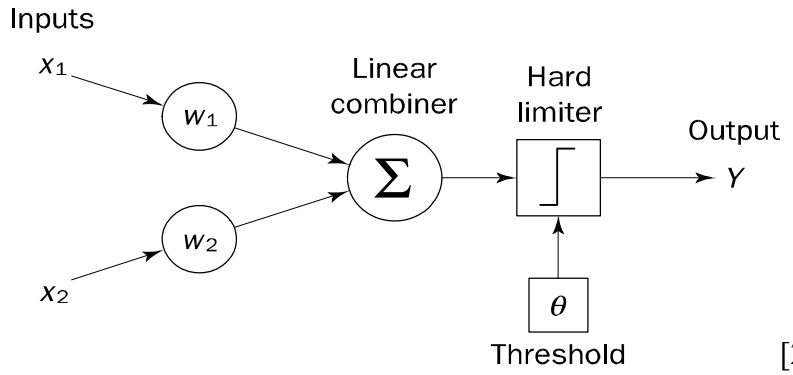
In the case of a binary classifier, the Sigmoid activation function should be used. The sigmoid activation function and the tanh activation function work terribly for the hidden layer. For hidden layers, ReLU or its better version leaky ReLU should be used. For a multiclass classifier, SoftMax is the best-used activation function. Though there are more activation functions known, these are known to be the most used activation functions.

4. Perceptron

In 1958, Frank Rosenblatt introduced a training algorithm that provided the first procedure for training a simple ANN: a perceptron (Rosenblatt, 1958). The perceptron is the simplest form of a neural network. It consists of a single neuron with adjustable synaptic weights and a hard limiter.

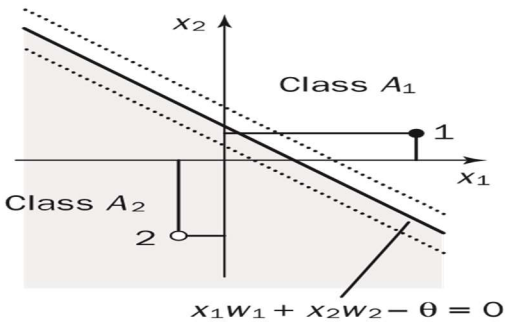
Perceptron's are the building blocks of artificial neural networks, inspired by the way biological neural networks in the human brain work. Developed by Frank Rosenblatt in 1957, a perceptron is a simplified model of a biological neuron that can make binary decisions. It takes multiple binary inputs (0 or 1), applies weights to these inputs, sums them up, and passes the result through an activation function to produce an output (0 or 1). Perceptron's are the simplest form of neural networks and were the foundation for more complex neural network architectures.

Perceptrons have limitations. They can only learn linear decision boundaries, meaning they can't solve problems that are not linearly separable. To overcome this limitation, multilayer Perceptrons (also known as feedforward neural networks) were developed, which can learn complex patterns by combining multiple Perceptrons in layers and introducing nonlinear activation functions like sigmoid or ReLU.

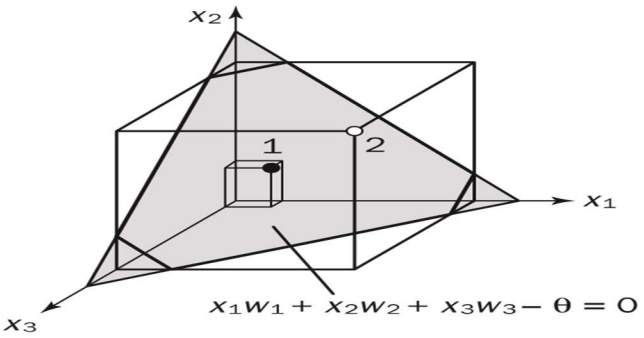


[2]

The aim of the perceptron is to classify inputs, or in other words externally applied stimuli x1; x2; ... ; xn, into one of two classes, say A1 and A2. Thus, in the case of an elementary perceptron, the n-dimensional space is divided by a hyperplane into two decision regions. The hyperplane is defined by the linearly separable function xn



(a)



(b) [2]

Perceptron Implementation in Python:

```
import numpy as np

class Perceptron:
    # Constructor to initialize weights, learning rate, and epochs
    def __init__(self, input_size, learning_rate=0.1, epochs=100):
        self.weights = np.zeros(input_size + 1)  # +1 for the bias
        self.learning_rate = learning_rate  # Set the learning rate
        self.epochs = epochs  # Set the number of training iterations

    # Prediction method using step function
    def predict(self, inputs):
        summation = np.dot(inputs, self.weights[1:]) + self.weights[0]  # Calculate the weighted sum
        return 1 if summation > 0 else 0  # Apply step function, return 1 if summation > 0, else 0

    # Training the Perceptron using training inputs and labels
    def train(self, training_inputs, labels):
        for _ in range(self.epochs):  # Iterate through training epochs
            for inputs, label in zip(training_inputs, labels):  # Iterate through training data
                prediction = self.predict(inputs)  # Make a prediction
                # Update weights based on prediction error and learning rate
                self.weights[1:] += self.learning_rate * (label - prediction) * inputs
                self.weights[0] += self.learning_rate * (label - prediction)  # Update bias

# Example usage:
# Training data (AND gate)
```

```
training_inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
labels = np.array([0, 0, 0, 1])
# Create and train the perceptron
perceptron = Perceptron(input_size=2)  # Initialize a Perceptron object with 2 input features
perceptron.train(training_inputs, labels)  # Train the Perceptron with the training data
# Test the perceptron
test_inputs = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
for inputs in test_inputs:
    prediction = perceptron.predict(inputs)  # Make predictions using trained Perceptron
    print(f"Input: {inputs}, Predicted Output: {prediction}")
```

5. Image processing

In the realm of neural networks, raw image data cannot be directly assimilated. It necessitates transformation into numerical values for the neural network's comprehension. This process is fundamental in image processing.

Firstly, it is imperative to grasp the dimensions of the image, denoted as, for instance, 200x200 pixels. Consider a grayscale image, where shades of grey are represented between the spectrum of black 0 and white 255 (it's just convention we can take any other value).
To translate this image into a format understandable by a neural network, it is divided into 40,000 boxes, with each box signifying a numerical value within the 0-255 range. These values indicate the intensity of black and white, and once organized, the data can be fed into the neural network.

Now, when dealing with a coloured image, any colour can be synthesized from a combination of red, green, and blue (RGB). For example, in a 200x200 pixel image, the data is organized into a 3D matrix with 200 rows and 200 columns. This matrix is essentially a stack of three matrices, where each matrix represents the intensity of the red, green, and blue colours respectively. The entries in these matrices signify the colour intensities. This organized data, now in a numerical form, can be directly input into a neural network.
In practical implementation, Python libraries such as NumPy and PIL (Pillow) prove instrumental. By leveraging these tools, the conversion process becomes streamlined. Here's how it can be done:

```
# Import required libraries
from PIL import Image  # Import the Image class from the Pillow library
import numpy as np  # Import the NumPy library

# For processing black and white images:
```

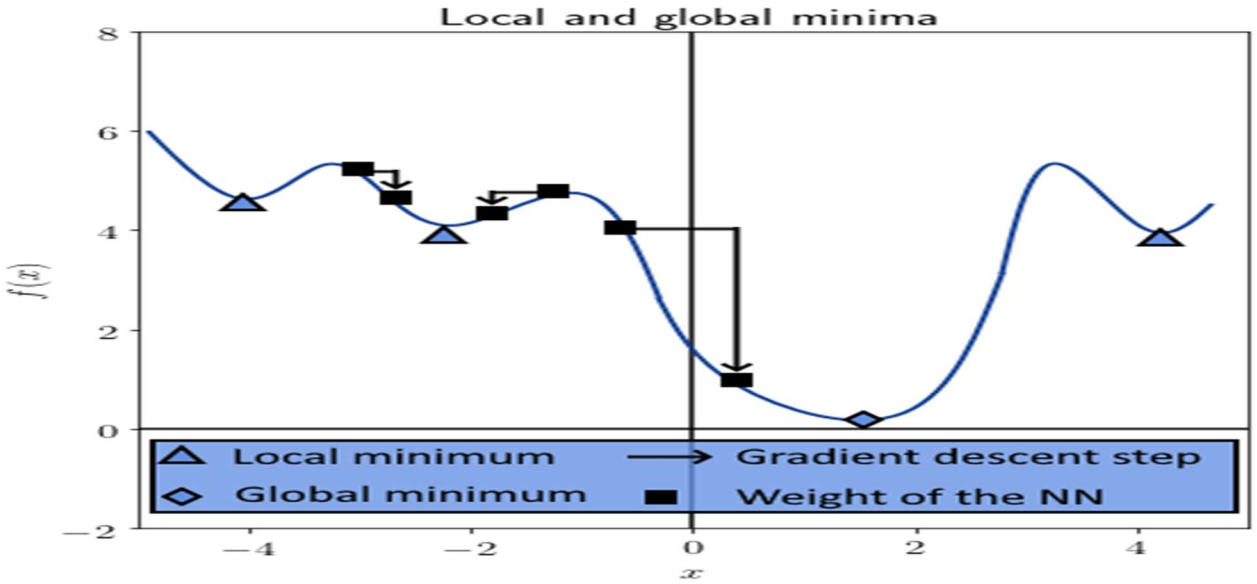
```
# Open the black and white image and convert it to grayscale ("L" mode)
bw_image = Image.open("black_white_image.jpg").convert("L")

# Convert the grayscale image to a NumPy array
bw_array = np.array(bw_image)

# Reshape the 2D array into a 1D array of 40000 values representing pixel values
bw_data = bw_array.reshape(1, -1)

# Normalize the pixel values to be between 0 and 1 (0 for black, 1 for white)
bw_data = bw_data / 255.0

# For processing colored images:

# Open the colored image
color_image = Image.open("colored_image.jpg")

# Convert the colored image to a NumPy array
color_array = np.array(color_image)

# Reshape the 3D array into a 2D array where each row represents a pixel and has 3 values (R, G, B)
color_data = color_array.reshape(-1, 3)

# Normalize the RGB values to be between 0 and 1 (0 for no color, 1 for full color)
color_data = color_data / 255.0
```

6 Gradient Descent

Gradient descent is an iterative process through which we optimize the parameters of a machine learning model. It's particularly used in neural networks, but also in logistic regression and support vector machines.

It is the most typical method for iterative minimization of a function. The goal of training a neural network is to find the set of parameters that minimizes this loss function

Its major limitation, though, consists of its guaranteed convergence to a local, not necessarily global, minimum:



Here's how Gradient Descent works in the context of neural networks:

1. Loss Function:

A neural network is trained to minimize a loss function (also known as the objective or cost function), which quantifies the difference between predicted values and actual values.

2. Gradient Calculation:

The gradient of the loss function with respect to the network's parameters (weights and biases) is calculated. The gradient indicates the direction in which the loss function is steepest.

Partial derivatives of the loss function with respect to each parameter are computed through a process called backpropagation. Backpropagation calculates the gradient efficiently using the chain rule of calculus.

3.Parameter Update:

The gradients indicate how the loss function will change if the parameters are adjusted.

The parameters (weights and biases) of the network are updated in the opposite direction of the gradient to minimize the loss. This updating process is known as the **learning step**.

The size of the steps is controlled by the learning rate. A smaller learning rate results in slower convergence but can avoid overshooting the optimal solution, while a larger learning rate can speed up the process but may overshoot the minimum.

4. Iterative Process:

Steps 2 and 3 are repeated iteratively for a predefined number of epochs or until the loss converges to a satisfactory level.

During each iteration, the gradients are calculated, and the parameters are updated, gradually reducing the loss.

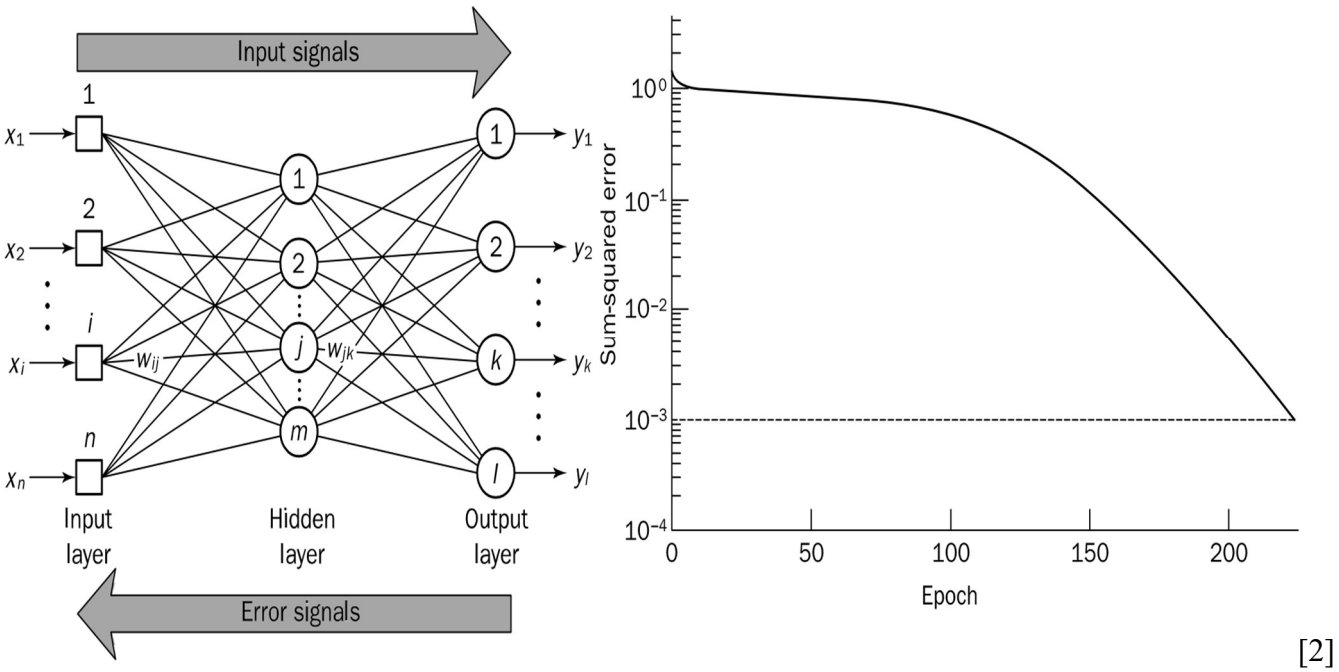
5. Convergence:
Gradient Descent continues until the algorithm converges to a minimum point in the loss function's landscape.
Convergence occurs when the changes in the loss become negligible or when a specific number of iterations is reached.

By adjusting the weights and biases in the direction that reduces the loss, Gradient Descent helps the neural network learn the patterns in the training data, improving its ability to make accurate predictions on unseen data.

It's worth mentioning that there are variations of Gradient Descent, such as Stochastic Gradient Descent (SGD) which optimize the algorithm's performance and efficiency. These variations involve updating the parameters based on a subset (mini batch) or a single data point from the training set, rather than the entire dataset, making the process faster, especially for large datasets.

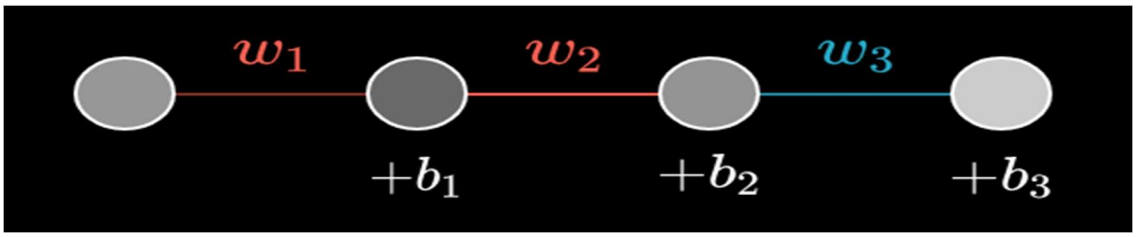
7. Back Propagation

We know, the neural network has neurons that work in correspondence with *weight*, *bias*, and their respective activation function. In a neural network, we would update the weights and biases of the neurons based on the error at the output. This process is known as ***back-propagation***. Activation functions make the back-propagation possible since the gradients are supplied along with the error to update the weights and biases.



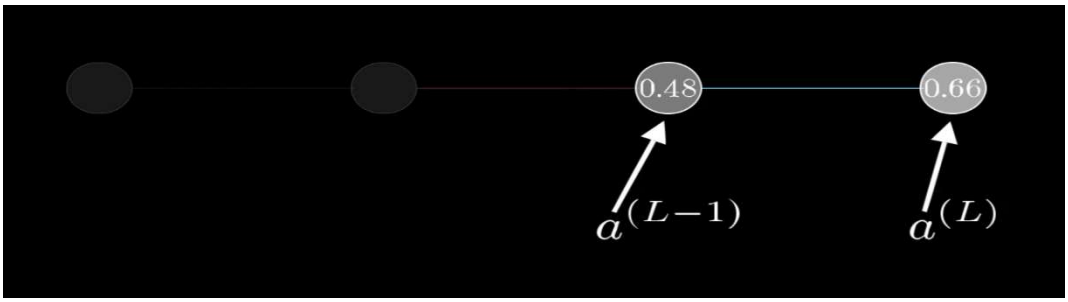
[2]

Calculus of Back Propagation



For Calculating the Gradient with Backpropagation, Let us start with an extremely simple network, where each layer has just one neuron.

> This network is determined by 3 weights (one for each connection) and 3 biases (one for each neuron, except the first), and our goal is to understand how changing each of them will affect the cost function. That way we know which adjustments will cause the most efficient decrease to the cost.



For now, let us just focus on the connection between the last two neurons. I will label the activation of that last neuron with a superscript L, indicating which layer it is in, so the activation of the previous neuron is a(L).
I will use a(L) instead of a^{L} for the sake of simplicity, don't confuse it with function

Let us say that for a certain training example, the desired output is y. That means that the cost for this one training example Co is

$$z^{(L)} = w^{(L)} a^{(L-1)} + b^{(L)} \longrightarrow \frac{\partial z^{(L)}}{\partial w^{(L)}} = a^{(L-1)}$$

$$a^{(L)} = \sigma\left(z^{(L)}\right) \longrightarrow \frac{\partial a^{(L)}}{\partial z^{(L)}} = \sigma'\left(z^{(L)}\right)$$

$$C_0 = \left(a^{(L)} - y\right)^2 \longrightarrow \frac{\partial C_0}{\partial a^{(L)}} = 2\left(a^{(L)} - y\right)$$

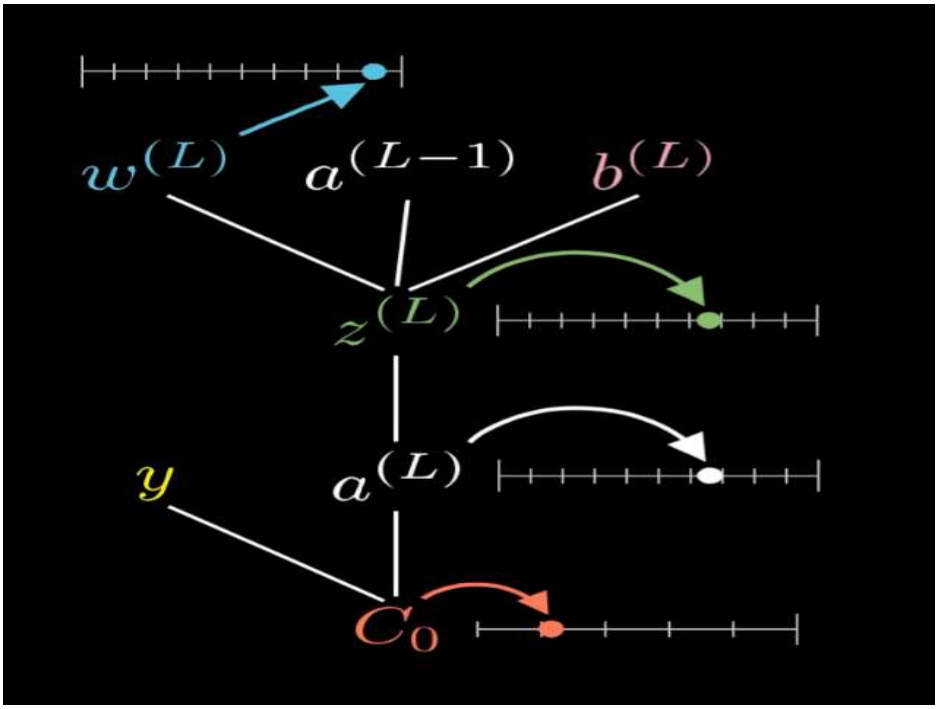
Last activation a(L) is determined by a weight, a bias, and the previous neuron's activation, all pumped through some special nonlinear function like a sigmoid or a ReLU as

$$a(L)=\sigma(w(L)a(L-1)+b(L))$$

It will make things easier for us to give a special name to this weighted sum, like z, with the same superscript as the activation:

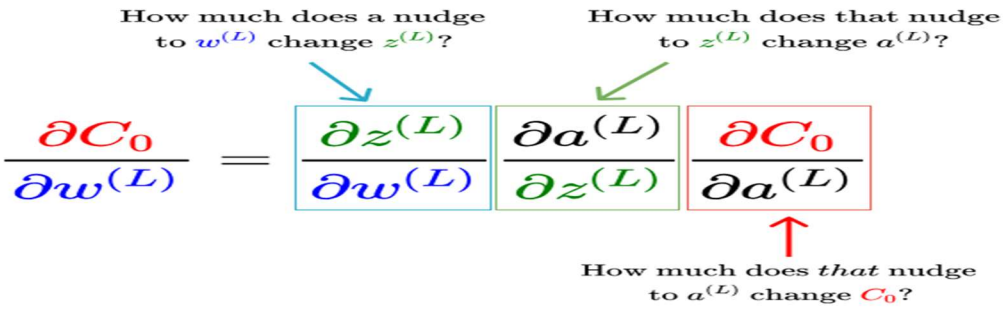
$$z(L)=w(L)a(L-1)+b(L)$$
$$a(L)=\sigma(z(L))$$

Our first goal is to understand how sensitive the cost Co is to small changes in the weight w(L). That is, we want to know the derivative $\partial C_0/\partial w(L)$.



When we see ∂w(L) term, think of it as meaning "some tiny nudge to w(L)", like a change by 0.01. And think of this ∂C0 term as meaning "whatever the resulting nudge to the cost is." We want their ratio.

Conceptually, this tiny nudge to w(L) causes some nudge to z(L), which in turn causes some change to a(L), which directly influences the cost, Co, as shown above



$$\frac{\partial C_0}{\partial w^{(L)}} = a^{(L-1)} \sigma'\left(z^{(L)}\right) 2\left(a^{(L)} - y\right)$$

This formula tells us how a nudge to that one particular weight in the last layer will affect the cost for that one particular training example. This is just one very specific piece of information, and we'll need to calculate a lot more to get the entire gradient vector.

Note: ∂z/∂w(L)= a[L-1] this means that the amount small nudge to weight influence the last layer depends upon activation the activation of last layer, this clearly explains the line, **The neuron that fire together wire together.**

The full cost function for the network (C) is the average all the individual costs for each training example:

$$C = \frac{1}{n} \sum_{k=0}^{n-1} C_k \qquad \frac{\partial C}{\partial w^{(L)}} = \frac{1}{n} \sum_{k=0}^{n-1} \frac{\partial C_k}{\partial w^{(L)}}$$

Next if we want the derivative of C with respect to the weight (rather than the derivative of Co with respect to the weight), we need to take the average of all the individual derivatives:

Above expression tells us how the overall cost of the network will change when we wiggle the last weight

Entries of gradient vector are the partial derivatives of the cost function C with respect to every weight and bias in the network. So, this derivative, ∂C/∂w(L) , is one of its entries!

To compute the full gradient, we will also need all the other derivatives with respect to all the other weights and biases in the entire network.

$$\nabla C = \begin{bmatrix} \frac{\partial C}{\partial w^{(1)}} \\ \frac{\partial C}{\partial b^{(1)}} \\ \vdots \\ \frac{\partial C}{\partial w^{(L)}} \\ \frac{\partial C}{\partial b^{(L)}} \end{bmatrix}$$

Let us work on computing the effect of bias in the last layer. The sensitivity of the cost function to a change in the bias is almost identical to the equation for a change to the weight and changing the bias associated with a neuron is the simplest way to change its activation. Unlike changing the weights or the activations from the previous layer, the effect of a change to the bias on the weighted sum is constant and predictable. And the expression for ∂Co/∂w(L) is exactly same except, all we've done is replaced to ∂z(L)/w(L) with ∂z(L)/ ∂b(L) which will be 1 here ,rest will be the same.

$$\frac{\partial C_0}{\partial b^{(L)}} = \frac{\partial z^{(L)}}{\partial b^{(L)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C_0}{\partial a^{(L)}} \qquad \frac{\partial C_0}{\partial a^{(L-1)}} = \frac{\partial z^{(L)}}{\partial a^{(L-1)}} \frac{\partial a^{(L)}}{\partial z^{(L)}} \frac{\partial C_0}{\partial a^{(L)}}$$

So, when the activation in the second-to-last layer is changed, the effect on $z(L)$ will be proportional to the weight.

But we do not really care about what happens when we change the activation directly, because we don't have control over that. All we can change, when trying to improve the network through gradient descent, is the values of the weights and biases. This is where the propagation backwards comes in.

Even though we will not be able to directly change that activation, it is helpful to keep track of, because we can just keep iterating this chain rule backwards to see how sensitive the cost function is to previous weights and biases.

For the general case equation will be same just we have to keep track of more indices

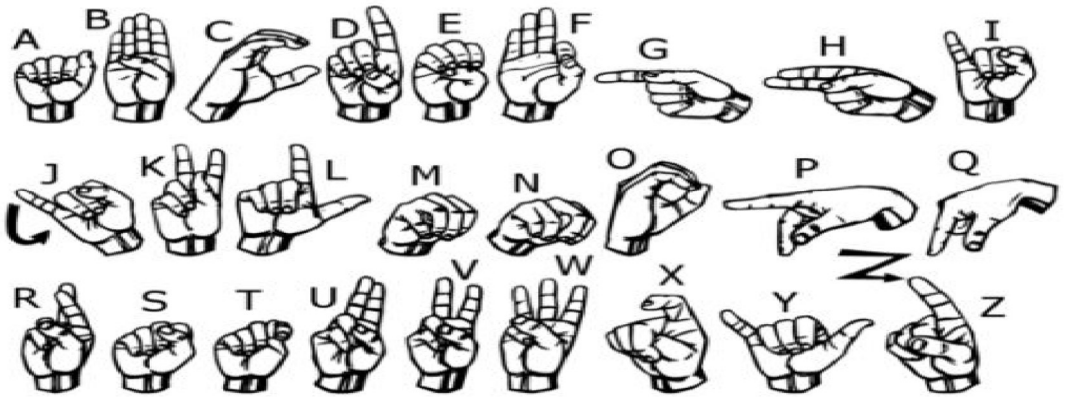
$$\frac{\partial C_0}{\partial w_{jk}^{(L)}} = \frac{\partial z_j^{(L)}}{\partial w_{jk}^{(L)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial C_0}{\partial a_j^{(L)}}$$

$$\frac{\partial C_0}{\partial a_k^{(L-1)}} = \underbrace{\sum_{j=0}^{n_L-1} \frac{\partial z_j^{(L)}}{\partial a_k^{(L-1)}} \frac{\partial a_j^{(L)}}{\partial z_j^{(L)}} \frac{\partial C_0}{\partial a_j^{(L)}}}_{\text{Sum over layer L}}$$

[4]

8 DATA SET

American Sign Language (ASL) is a complete, natural language that has the same linguistic properties as spoken languages, with grammar that differs from English. ASL is expressed by the movement of hands and faces. This dataset consists of 27,455 images of hand signs, each image is of 28 x 28 size and in grayscale format. The dataset format is patterned to match closely with the classic MNIST. Images in the dataset belong to a label from 0–25 representing letters from A-Z (but no cases of 9=J or 25=Z as they involve hand motion). The training data (27,455 cases) and the test data (7,172 cases) are approximately half the size of standard MNIST but otherwise similar to a header row of the label, pixel1, pixel…. Pixel784. The original hand gesture image data represented multiple users repeating gestures against different backgrounds. The Sign Language MNIST data came from greatly extending the small number (1704) of the colour images included as not cropped around the hand region of interest.



9 CODING PART

1 Training

```
# Import Libraries
from keras.models import Sequential
from keras.layers import Convolution2D, MaxPooling2D, Flatten, Dense

# Step 1: Building the CNN

# Initialize the CNN
classifier = Sequential()

# Add Convolutional Layers
# First Convolution Layer and Pooling
classifier.add(Convolution2D(32, (3, 3), input_shape=(64, 64, 1), activation='relu'))
classifier.add(MaxPooling2D(pool_size=(2, 2)))

# Second Convolution Layer and Pooling
classifier.add(Convolution2D(32, (3, 3), activation='relu'))
classifier.add(MaxPooling2D(pool_size=(2, 2)))

# Flatten the Layers to Prepare for Fully Connected Layers
classifier.add(Flatten())

# Add Fully Connected Layers
# Hidden Layer
classifier.add(Dense(units=128, activation='relu'))

# Output Layer (6 units for 6 classes, softmax activation for multi-class classification)
classifier.add(Dense(units=6, activation='softmax'))

# Compile the CNN
# Using 'adam' optimizer, 'categorical_crossentropy' loss for multi-class, and accuracy
metric
classifier.compile(optimizer='adam', loss='categorical_crossentropy', metrics=['accuracy'])

# Step 2: Preparing the train/test data and training the model

# Import necessary libraries for data augmentation
from keras.preprocessing.image import ImageDataGenerator

# Data Augmentation and Loading Data

# Data Augmentation for Training Data
train_datagen = ImageDataGenerator(
    rescale=1./255,  # Rescale pixel values to the range [0, 1]
```

```
    shear_range=0.2,  # Shear transformations
    zoom_range=0.2,   # Random zooms
    horizontal_flip=True  # Horizontal flips
)

# Data Augmentation for Test Data (only rescaling)
test_datagen = ImageDataGenerator(rescale=1./255)

# Load Training Data (with data augmentation) and Test Data
training_set = train_datagen.flow_from_directory('data/train',
                                                 target_size=(64, 64),
                                                 batch_size=5,
                                                 color_mode='grayscale',
                                                 class_mode='categorical')

test_set = test_datagen.flow_from_directory('data/test',
                                            target_size=(64, 64),
                                            batch_size=5,
                                            color_mode='grayscale',
                                            class_mode='categorical')

# Training the Model
# Using fit_generator for training with augmented data
classifier.fit_generator(training_set,
                         steps_per_epoch=600,  # No of images in training set
                         epochs=10,
                         validation_data=test_set,
                         validation_steps=30)  # No of images in test set

# Step 3: Saving the Model

# Save the Model Architecture to JSON File
model_json = classifier.to_json()
with open("model-bw.json", "w") as json_file:
    json_file.write(model_json)

# Save the Model Weights to HDF5 File
classifier.save_weights('model-bw.h5')
```

2 Predicting

```
import numpy as np
from keras.models import model_from_json
import operator
import cv2
import sys, os
```

```
# Loading the pre-trained model architecture from a JSON file
json_file = open("model-bw.json", "r")
model_json = json_file.read()
json_file.close()
loaded_model = model_from_json(model_json)

# Load pre-trained weights into the model
loaded_model.load_weights("model-bw.h5")
print("Loaded model from disk")

# Open a connection to the computer's camera (assuming the default camera index 0)
cap = cv2.VideoCapture(0)

# Dictionary to map model output indices to gesture labels
categories = {0: 'ZERO', 1: 'ONE', 2: 'TWO', 3: 'THREE', 4: 'FOUR', 5: 'FIVE'}

while True:
    # Read a frame from the video capture
    _, frame = cap.read()

    # Flip the frame horizontally to create a mirror image
    frame = cv2.flip(frame, 1)

    # Define the region of interest (ROI) for hand gesture detection
    x1 = int(0.5 * frame.shape[1])
    y1 = 10
    x2 = frame.shape[1] - 10
    y2 = int(0.5 * frame.shape[1])

    # Draw a bounding box around the ROI
    cv2.rectangle(frame, (x1 - 1, y1 - 1), (x2 + 1, y2 + 1), (255, 0, 0), 1)

    # Extract the ROI
    roi = frame[y1:y2, x1:x2]

    # Resize and preprocess the ROI for model prediction
    roi = cv2.resize(roi, (64, 64))
    roi = cv2.cvtColor(roi, cv2.COLOR_BGR2GRAY)
    _, test_image = cv2.threshold(roi, 120, 255, cv2.THRESH_BINARY)

    # Reshape the image to match the input shape expected by the model
    test_image = test_image.reshape(1, 64, 64, 1)

    # Make a prediction using the loaded model
    result = loaded_model.predict(test_image)

    # Map model output probabilities to gesture labels
```

```
    prediction = {label: result[0][i] for i, label in categories.items()}

    # Sort predictions by probability in descending order
    prediction = sorted(prediction.items(), key=operator.itemgetter(1), reverse=True)

    # Display the top predicted gesture label on the frame
    cv2.putText(frame, prediction[0][0], (10, 120), cv2.FONT_HERSHEY_PLAIN, 1, (0, 255, 255), 1)

    # Display the frame with predictions
    cv2.imshow("Frame", frame)

    # Break the loop if the 'Esc' key is pressed
    interrupt = cv2.waitKey(10)
    if interrupt & 0xFF == 27:  # ASCII code for 'Esc' key
        break
# Release the camera and close all OpenCV windows
cv2.destroyAllWindows()
cap.release()
# Import required libraries
from sklearn.metrics import confusion_matrix  # Import the confusion_matrix function
from sklearn.metrics
import numpy as np  # Import the NumPy library
# Example true labels and predicted labels
true_labels = np.array([1, 0, 1, 2, 2, 1, 0, 2, 1])  # True labels from the dataset
predicted_labels = np.array([1, 0, 1, 2, 1, 1, 0, 2, 2])  # Predicted labels from the model
# Create a confusion matrix
confusion_mat = confusion_matrix(true_labels, predicted_labels)  # Compute the confusion
matrix
print("Confusion Matrix:")  # Print the label names
print(confusion_mat)  # Print the confusion matrix
```

9 Results:

I have achieved an accuracy of approx. 84% in our model using the algorithm and using the combination of layer 1 and layer 2 we achieve an accuracy up to 92.0%

This is only for ASL for other sign language I must collect the dataset and model the neural network accordingly.

10 Conclusion:

I achieved final accuracy of 92.0% on this dataset. I can improve this prediction after implementing two layers of algorithms in which we verify and predict symbols which are more like each other.

11 Future Scope:

I am thinking of improving the preprocessing to predict gestures in low light conditions with a higher accuracy and I will also try to deploy it as an app

13 References

1) **Artificial Intelligence: A Guide to Intelligent Systems, Second Edition 2005, Michael Negnevitsky, Pearson Education Limited**
2) **Neural Networks and Deep Learning Michael Nielsen Determination Press, 2015**
3) **"Deep Learning" by Ian Goodfellow, Yoshua Bengio, and Aaron Courville (1st Edition, MIT Press)**
4) **Online Resources**
    1. **3Blue1Brown - But what is a Neural Network?**
    2. **https://youtu.be/Bc2Gey7bmhk?si=usKk7eEpxffe5p2n**