REPUBLIQUE ALGERIENNE DEMOCRATIQUE ET POPULAIRE
MINISTERE DE L'ENSEIGNEMENT SUPERIEURE ET DE LA RECHERCHE SCIENTIFIQUE

**UNIVERSITE IBN KHALDOUN - TIARET**

# MEMOIRE

Présenté à :

FACULTÉ MATHEMATIQUES ET INFORMATIQUE
DÉPARTEMENT D'INFORMATIQUE

Pour l'obtention du diplôme de :

MASTER

Spécialité : *génie logiciel*

Par :

**Hatem M'hamed Amine**

Sur le thème

---

# Le deep learning pour la classification des images dans différents système de couleur

---

Soutenu publiquement le juillet 2019 à Tiaret devant le jury composé de :

| | | | |
|---|---|---|---|
| Mr MERATI Medjeded | MCB | Université Ibn-Khaldoun Tiaret | Président |
| Mr CHENINE Abdelkader | MCA | Université Ibn-Khaldoun Tiaret | Encadreur |
| Mr MAZZOUG Karim | MCA | Université Ibn-Khaldoun Tiaret | Examinateur |

# Abstract

Image classification is fundamental in the field of artificial intelligence, recently Deep Residual Learning [1] and the new coming model CapsulNet[2] have shown state-of-the-art performance for image classification tasks , they takes data-sets mostly as input in the form of RGB images even though we have many other colorspaces available.
In this thesis we try to understand the impact of image color-space on the performance of CNN models in Image classification .

We evaluate this on CIFAR10 [3]data-set by converting it into five other color-spaces HLS, HSV, LUV, LAB, YUV and trained each one of them in two different deep learning architecture models namely ResNet20 and CapsulNet , the results obtained show a minor change in accuracy but even a small percentage may make the difference, in the other hand LUV is good alternative it show improvement about 0.92% comparison to RGB in ResNet20 and 0.39% in CapsulNet.



**Keywords:** Deep learning, Color-spaces, Convolutional Neural Networks, ResNet, CapsuleNet

# Acknowledgment

Praise be to Allah, Lord of the Worlds **الْحَمْدُ للهِ رَبِّ الْعَالَمِينَ**

I would like to express my special thanks of gratitude to my Project Guide Prof. CHENINE Abdelkader , who gave me this wonderful opportunity to work under his guidance on topic that always fascinated me artificial intelligence especially sub-field deep learning , which helped me to get a lot of exposure of Academic Research and also to learn so many new things.

I would also like to thank all the teachers who accompanied me throughout my studies at Ibn Khaldun Tiaret University and All employees of computer science department

# Contents

# List of Figures

# List of Tables

# INTRODUCTION

Deep learning has turned applications that previously required vision expertise into engineering challenges solvable by non-vision experts. Deep learning transfers the logical burden from an application developer, who develops and scripts a rules-based algorithm, to an engineer training the system. It also opens a new range of possibilities to solve applications that have never been attempted without a human. In this way, deep learning makes machine vision easier to work with, while expanding the limits of what a computer and camera can accurately inspect.

Deep learning convolutional neural network models have shown greatest performance in image recognition which sometimes exceeds the human vision, but we continue to face some issues with over-fitting and vanishing gradient. To get over, data augmentation, batch normalization and dropout are used. Can we use different image color-space than the usually used RGB to get better performance? This what we shall show in this thesis by converting CIFAR10 data-set into five other color-spaces namely HLS, HSV, LUV, LAB, YUV and train each one of them in two different deep learning architecture models : ResNet20 [1] and CapsulNet[2] .

## 1. Background

This is not the first time such an experimentation is done to understand the performance of CNN on different color spaces, to our knowledge there is two controversy published papers, in [4] the authors claim that image color space have impact over CNN performance, they said that LUV color space is a best alternative to work with CNN model while YUV color space is the worst one. A more recent paper [5] shows that image color space by itself have no effect over CNN performance, they said that the accuracy did not vary too much using different color spaces but it can be used as a sort of data augmentation by combined different image color-spaces from the original data-set using dense-net model, this will help to deal with common issues such as the vanishing gradient problem and the problem of over-fitting and get better performance.

## 2. The Scope of this Research

This thesis focus on study of the impact of color-space in image classification by deep learning model, especially Deep Residual Learning (ResNet) won the first place in the ILSVRC 2015 classification competition with top-5 error rate of 3.57% , and the new model capsule Network by experimenting using Keras [6] which is a high-level neural networks API, written in Python and capable of running on top of TensorFlow. There are other architectures like AlexNet, VGG, GoogLeNet and framework like Caffe, Pytorch . We also focus in CPU computing .

## 3. Research Goals and Questions

Our goal is to answer a simple question: The RGB color space is the most color space used for datasets in image classification using deep learning model, can the performance of deep learning model be improved by using different color spaces?

## 4. Structure of the manuscript

**Chapter 1** is a theoretical background it focuses in the context and definition of deep learning it also describes the artificial neuron which is the building block of artificial neural network and Convolutional neural network (CNN), it also describes the gradient descent and the back-propagation algorithm with some mathematical notation, it also describe briefly the architecture ResNet and CapsulNet .

**Chapter 2** describes image color spaces especially RGB, HSV, HSL, YUV, LUV, LAB, sRGB and how to convert from a RGB to other colors spaces.

**Chapter 3** describes the tool, library and the data-set CIFAR10 used in the experimentation. The conversion process of the data-set to other colors spaces, it also briefly describes the implementation. It ended by analyzing the results obtained.

# CHAPTER I
# DEEP LEARNING AND ITS APPLICATION

## I.1 Historical Context

### I.1.1 Artificial intelligence [7]

Artificial intelligence has a rich history going back to 1950, when a handful of pioneers from the nascent field of computer science started asking whether computers could be made to "think" a question whose ramifications are still in exploring today. A concise definition of the field would be as follows: ***the effort to automate intellectual tasks normally performed by humans***.

Artificial intelligence is a general field that encompasses machine learning and deep learning (see Figure 1), but that also includes many more approaches that don't involve any learning early chess programs, for instance, only involved hard-coded rules crafted by programmers, and didn't qualify as machine learning, for a fairly long time, many experts believed that human-level artificial intelligence could be achieved by having programmers handcraft a sufficiently large set of explicit rules for manipulating knowledge. This approach is known as symbolic AI , and it was the dominant paradigm in AI from the 1950s to the late 1980s. It reached its peak popularity during the expert systems boom of the 1980s.

Although symbolic AI proved suitable to solve well-defined, logical problems that could be easily described formally, such as playing chess, it turned out to be intractable to figure out explicit rules for solving more complex, fuzzy problems, such as image classification, speech recognition, and language translation.

A new approach arose to take symbolic AI 's place: machine learning.

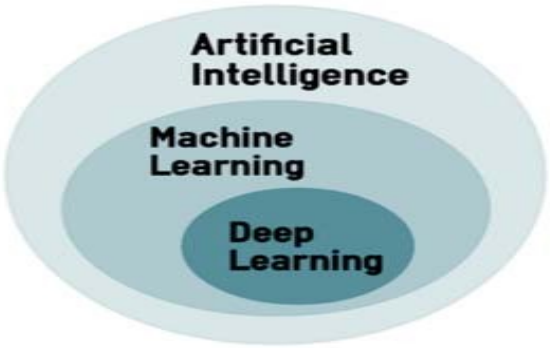



**Figure 1** Deep Learning is a subfield of Machine Learning which is a subfield of AI

### I.1.2 Machine learning [7]

Machine learning arises from this question: could a computer go beyond "what we know how to order it to perform" and learn on its own how to perform a specified task? , Could a computer surprise us? Rather than programmers crafting data-processing rules by hand, could a computer automatically learn these rules by looking at data?

This question opens the door to a new programming paradigm. In classical programming, the paradigm of symbolic AI, humans input rules (a program) and data to be processed according to these rules, and outcome answers , With machine learning, humans input data as well as the answers expected from the data, and outcome the rules (a program). These rules can then be applied to new data to produce original answers (see Figure 2).

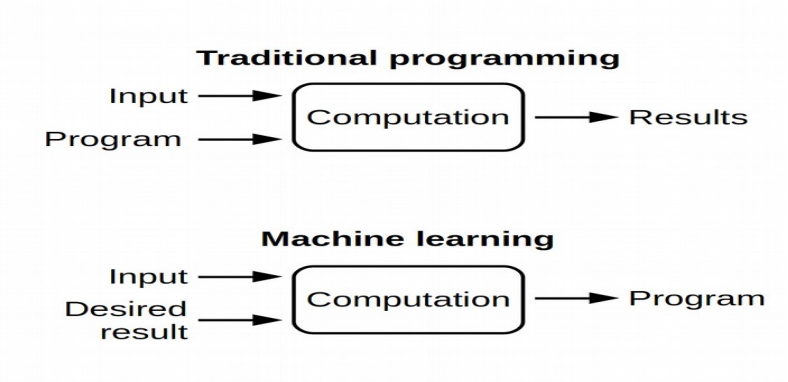



**Figure 2** traditional programming vs machine learning

A ***machine-learning system is trained rather than explicitly programmed*** As a subbranch of Artificial intelligence (AI) it focuses on teaching computers how to learn without the need to be programmed for specific tasks, learning in the context of machine learning describes an automatic search process for better representations.
A popular definition of learning in the context of computer programs is "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P , if its performance at tasks in T , as measured by P , improves with experience E" [8]
Machine learning systems, with shallow or deep architectures (see Figure3), have ability to learn and improve with experience. The process of machine learning begins with the raw data which is used for extracting useful information that helps in decision-making.

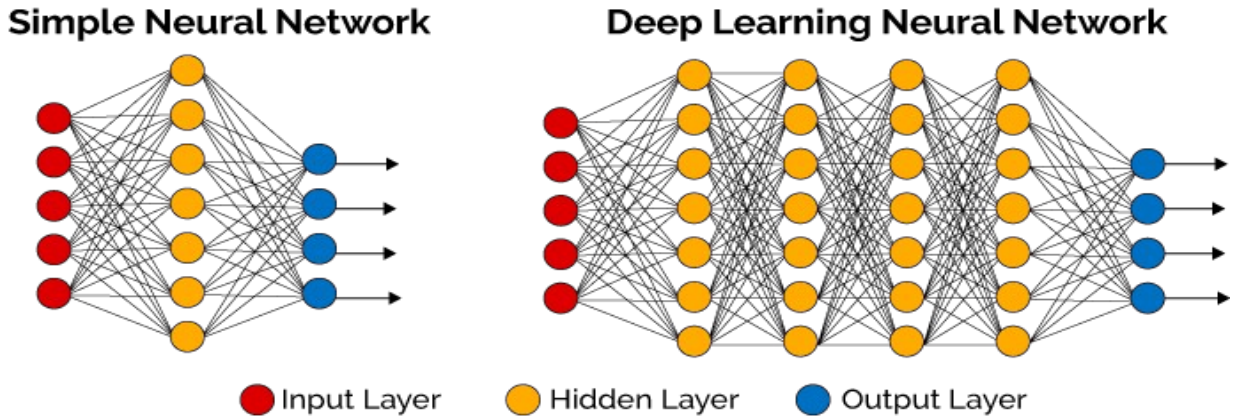



**Figure 3** shallow learning vs deep learning

The primary aim is to allow a machine to learn useful information just like humans do, at abstract level, machine learning can be carried out using following approaches :

**I.1.2.1 Supervised learning** adapts a system such that for a given input data it produces a target output. The learning data is made up of tuples (attributes, label) where “attributes” represent the input data and “label” represents the target output. The goal here is to adapt the system so that for a new input the system can predict the target output. Supervised learning can use both continuous and discrete types of input data.

**I.1.2.2 Unsupervised learning** involves data that comprises of input vectors without any target output. There are different objectives in unsupervised learning, such as clustering ,density estimation, and visualization.
The goal of clustering is to discover groups of similar data items on the basis of measured or perceived similarities between the data items. The purpose of density estimation is to determine the distribution of the data within the input space. In visualization, the data is projected down from a high-dimensional space to two or three dimensions to view the similar data items.

**I.1.2.3 Semi-supervised** learning first uses unlabeled data to learn a feature representation of the input data and then uses the learned feature representation to solve the supervised task.
The training data-set can be divided into two parts: the data samples with corresponding labels and the data samples where the labels are not known. Semi-supervised learning can involve not providing with an explicit form of error at each-time but only a generalized reinforcement is received giving indication of how the system should change its behavior, and this is sometimes referred to as reinforcement learning. Reinforcement learning has been successful in applications as diverse as autonomous helicopter flight, robot legged locomotion, cell-phone network routing, marketing strategy selection, factory control and efficient web-page indexing.

**I.1.2.4 Shallow Learning**
Shallow architectures are a simple artificial neural networks, they perform good on many common machine learning problems, and they are still used in a vast majority of today’s machine learning applications. However, there has been an increased interest in deep architectures recently, in the hope to find means to solve more complex real-world problems (e.g., image analysis or natural language understanding) for which shallow architectures are unable to learn models adequately.

## I.2 Deep Learning

*“Deep learning methods are representation-learning methods with multiple levels of representation, obtained by composing simple but nonlinear modules that each transform the representation at one level (starting with the raw input) into a representation at a higher, slightly more abstract level. [. . . ] The key aspect of deep learning is that these layers are not designed by human engineers: they are learned from data using a general-purpose learning procedure” [9]*

Deep learning is a new area of machine learning which has gained popularity in recent past , it use artificial neural networks (ANN) slightly inspired by the structure of neurons located in the human brain .
Informally, The word deep in deep learning isn’t a reference to any kind of deeper understanding achieved by the approach, rather, it stands for this idea of successive layers of representations, how many hidden layers contribute to a model of the data is called the depth of the model, other appropriate names for the field could have been layered representations learning and hierarchical representations learning.

Modern deep learning often involves tens or even hundreds of successive layers of representations and they're all learned automatically from exposure to training data.
Another definition defines deep learning [10] as neural networks with large number of parameters and layers in one of four fundamental network architectures:

(1) Unsupervised pretrained networks
(2) Convolutional neural networks
(3) Recurrent neural networks
(4) Recursive neural networks

Automatic feature extraction[11] is another one of the great advantages that deep learning has over traditional machine learning algorithms (see Figure 4 )
By feature extraction, we mean that the network's process of deciding which characteristics of a dataset can be used as indicators to label that data reliably. Historically, machine learning practitioners have spent months, years, and sometimes decades of their lives manually creating exhaustive feature sets for the classification of data.

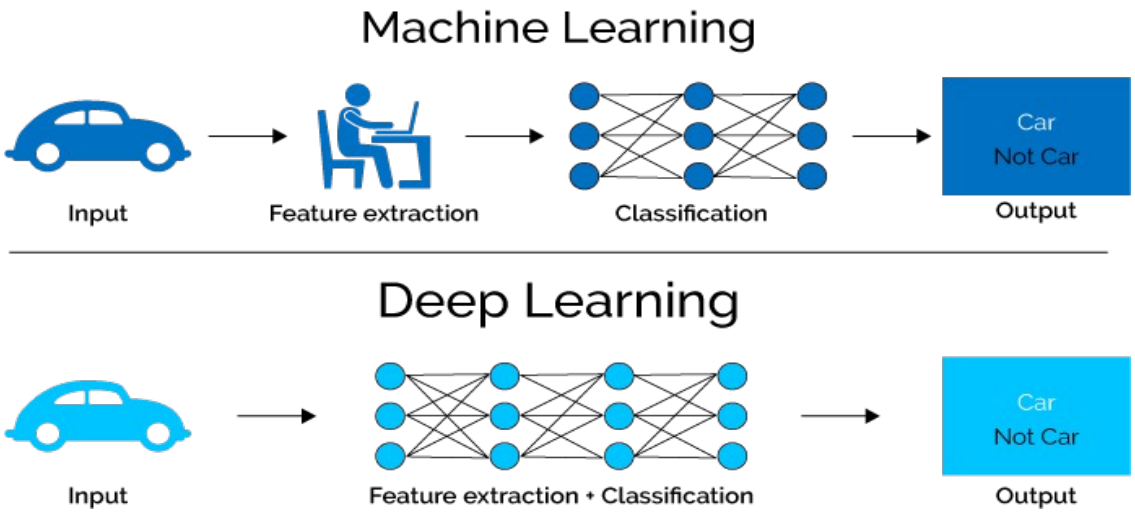



**Figure 4** Feature extraction in machine learning vs deep learning

Deep learning is still far from being a mature and well-understood field, many real-world applications such as vision-based detection and recognition, product recommendation, speech recognition and synthesis, energy conservation, drug discovery, finance, and marketing are already using deep learning algorithms.
A field that is not completely mature is a double-edged sword. On one edge, it offers a lot of opportunities for discovery and exploitation. There are many unsolved problems in deep learning. This translates into opportunities to be the first to market product development, publication, or recognition.
The other edge is that it would be difficult to trust a not completely well-understood field in a mission-critical environment. We can safely say that if asked, very few machine learning engineers will ride an auto-pilot plane controlled by a deep learning system. There is a lot of work to be done to gain this level of trust.

### I.2.1 The Neuron

The foundational unit of the human brain is the neuron. A tiny piece of the brain, about the size of grain of rice, contains over 10,000 neurons, each of which forms an average of 6,000 connections with other neurons, it's this massive biological network that enables us to experience the world around us.

At its core, the neuron is optimized to receive information from other neurons, process this information in a unique way, and send its result to other cells. This process is summarized in Figure 5 ,

The neuron receives its inputs along antennae-like structures called dendrites. Each of these incoming connections is dynamically strengthened or weakened based on how often it is used (this is how we learn new concepts!),

and it's the strength of each connection that determines the contribution of the input to the neuron's output. After being weighted by the strength of their respective connections, the inputs are summed together in the cell body. This sum is then transformed into a new signal that's propagated along the cell's axon and sent off to other neurons.[10]

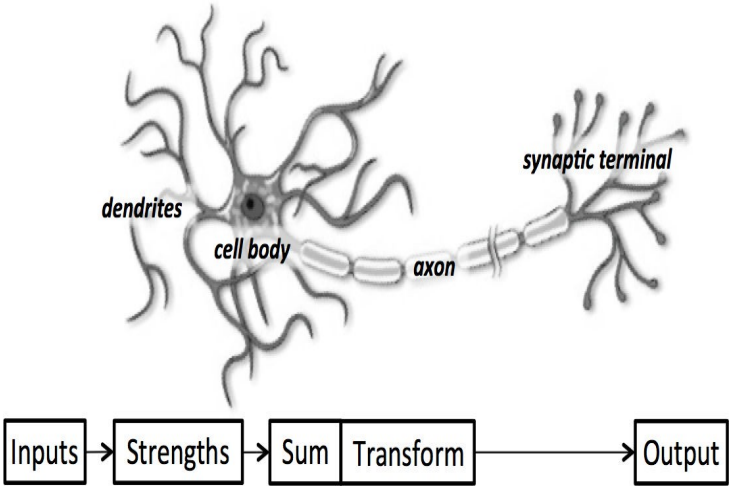



**Figure 5** structur of Neuron

### I.2.2 Artificial neuron:

this functional understanding of the neurons in our brain is translated into an artificial model that can be represented on computer. Such a model is described in Figure 6 , it takes in some number of inputs, $x_1, x_2, \ldots, x_n$, each of which is multiplied by a specific weight, $w_1, w_2, \ldots, w_n$. These weighted inputs are, as before, summed together to produce the logit of the neuron, $z = \sum_{i=0}^{n} w_i x_i$ In many cases, the logit also includes bias, which is a constant .

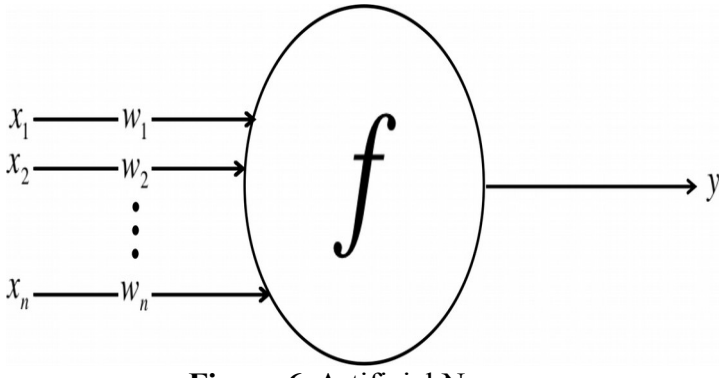



**Figure 6** Artificial Neuron

The logit is then passed through a function f to produce the output $y=f(z)$ . This output can be transmitted to other neurons, mathematically the artificial neuron functionality can be re-expressing in vector form, the inputs as a vector x = [$x_1$ $x_2$ ... $x_n$ ] and the weights of the neuron as w = [$w_1$ $w_2$ ... $w_n$ ] , and the output of the neuron as $y=f(x.w+b)$ , where b is the bias term. In other words, the output is computing by performing the dot product of the input and weight vectors, adding in the bias term to produce the logit, and then applying the transformation function [10].

### I.2.3 Artificial Neural Networks

Neural networks are one type of model for machine learning; they have been around for at least 50 years. in the mid-1980s and early 1990s, many important architectural advancements were made in neural networks. However, the amount of time and data needed to get good results slowed adoption. In the early 2000s computational power expanded exponentially and the industry saw a "Cambrian explosion" of computational techniques that were not possible prior to this, This made the interest come back in Neural networks [10].

Neural networks are a computational model that shares some properties with the animal brain in which many simple units are working in parallel with no centralized control unit.

The weights between the units are the primary means of long-term information storage in neural networks. **Updating the weights is the primary way the neural network learns new information.**

The most well-known and simplest-to-understand neural network is the feed-forward multilayer neural network (see figure 7). It has an input layer, one or many hidden layers, and a single output layer. Each layer can have a different number of neurons and each layer is fully connected to the adjacent layer.

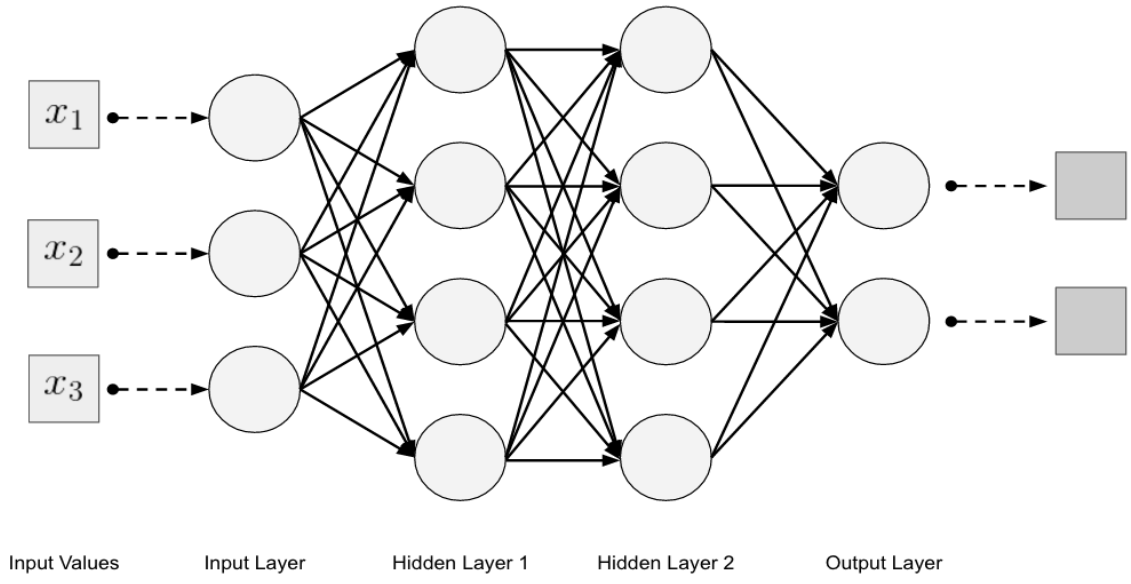



**Figure 7** Artificial Neural Networks

A feed-forward multilayer neural network can represent any function, given enough artificial neuron units. It is generally trained by a learning algorithm called Back-propagation learning. It uses gradient descent on the weights of the connections in a neural network to minimize the error on the output of the network.[10]

**I.2.3.1 Input layer.** This layer is how we get input data (vectors) fed into our network. The number of neurons in an input layer is typically the same number as the input feature to the network. Input layers are followed by one or more hidden layers Input layers in classical feed-forward neural networks are fully connected to the next hidden layer, yet in other network architectures, the input layer might not be fully connected.

**I.2.3.2 Hidden layer.** There are one or more hidden layers in a feed-forward neural network. The weight values on the connections between the layers are how neural networks encode the learned information extracted from the raw training data. Hidden layers are the key to allowing neural networks to model nonlinear functions.

**I.2.3.3 Output layer.** We get the answer or prediction from our model from the output layer. Given that we are mapping an input space to an output space with the neural network model, the output layer gives us an output based on the input from the input layer. Depending on the setup of the neural network, the final output may be a real-valued output (regression) or a set of probabilities (classification). This is controlled by the type of activation function we use on the neurons in the output layer.
The output layer typically uses either a softmax or sigmoid activation function for classification.

**I.2.3.4 Activation Functions**
activation functions are used to propagate the output of one layer's nodes forward to the next layer (up to and including the output layer) , it has also the ability to filter out data , some usual Activation functions are :

**I.2.3.4.1 Linear**

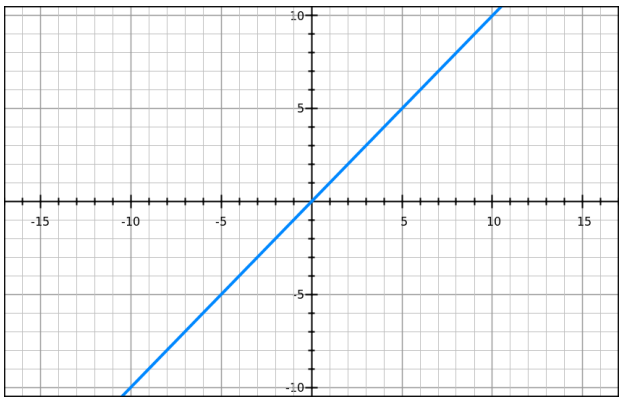



**Figure 8** identity function

A linear transform is basically the identity function (دالـة محايدة), and $f(x)=wx$ where the dependent variable has a direct, proportional relationship with the independent variable. In practical terms, it means the function passes the signal through unchanged (figure 8) Linear neurons are easy to compute with, but they run into serious limitations. In fact, it can be shown that any feed-forward neural network consisting of only linear
neurons can be expressed as a network with no hidden layers. This is problematic because *hidden layers are what enable us to learn important features from the input data. In other words, in order to learn complex relationships, we need to use neurons that employ some sort of nonlinearity.*

**I.2.3.4.2 None linear activation functions**

There are three major types of neurons that are used in practice that introduce nonlinearities in their computations.

**a) Sigmoid** neuron, which uses the function:

$$f(z)=\frac{1}{1+e^{-z}} \qquad (1)$$

Intuitively, this means that when the logit is very small, the output of a logistic neuron is very close to 0. When the logit is very large, the output of the logistic neuron is close to 1. In-between these two extremes, the neuron assumes an S-shape, as shown in Figure 9

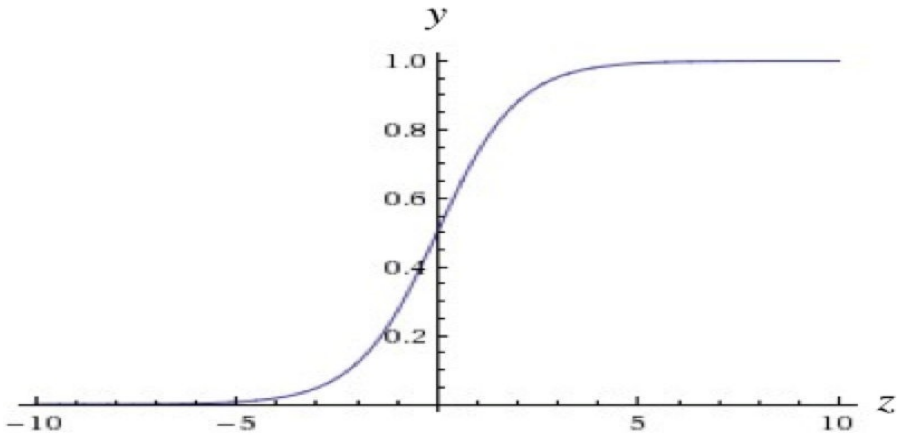



**Figure 9** The output of a sigmoid neuron as z varies

**b) Tanh** neurons use a similar kind of S-shaped nonlinearity, but instead of ranging from 0 to 1, the output of tanh neurons range from −1 to 1, they use $f(z)=\tanh(z)$ , tanh represents the ratio of the hyperbolic sine to the hyperbolic cosine: tanh(x) = sinh(x) / cosh(x) .

The resulting relationship between the output y and the logit z is described by Figure 10 When S-shaped nonlinearities are used, the tanh neuron is often preferred over the sigmoid neuron because it is zero-centered.

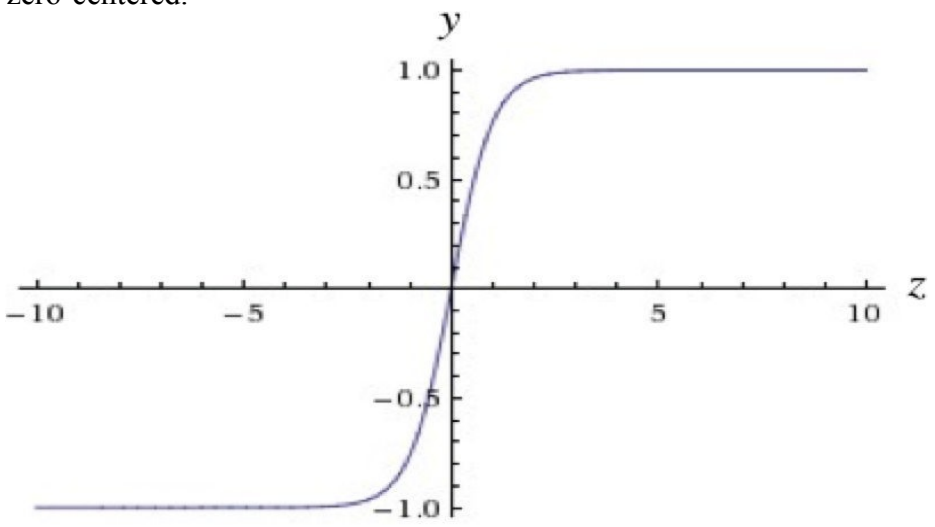



**Figure 10** The output of a tanh neuron as z varies

**c) ReLU** a different kind of nonlinearity is used by the **restricted linear unit** neuron. It uses the function $f(z)=max(0,z)$ (Figure 11) .
The ReLU has recently become the neuron of choice for many tasks (especially in computer vision) for a number of reasons, one of it as strategies to combat the potential pitfalls , because the gradient of a ReLU is either zero or a constant, it is possible to reign in the vanishing exploding gradient issue. ReLU activation functions have shown to train better in practice than sigmoid activation functions.

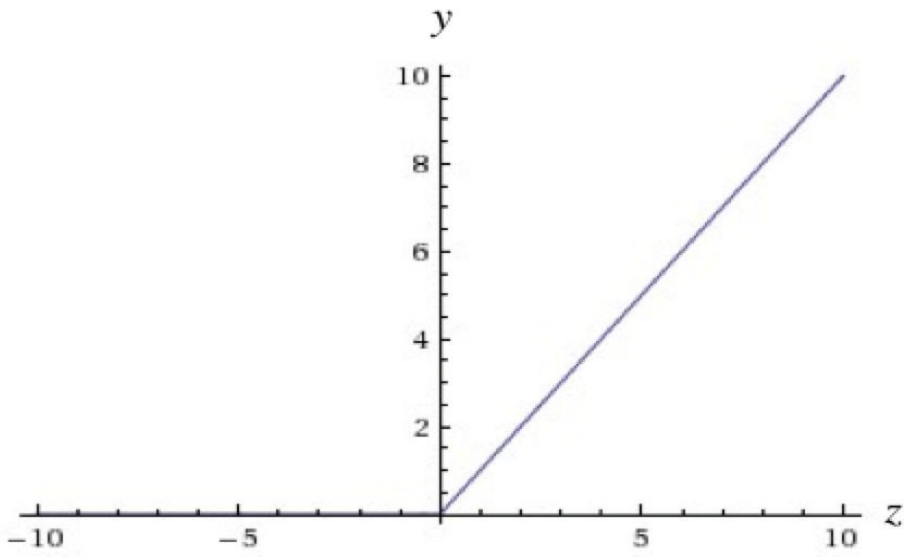



**Figure 11** The output of a ReLU neuron as z varies

**d) Softmax**
Softmax is a generalization of logistic regression inasmuch as it can be applied to continuous data (rather than classifying binary) and can contain multiple decision boundaries. It handles multinomial labeling systems. Softmax is the function often find at the output layer of a classifier.(see Figure12)

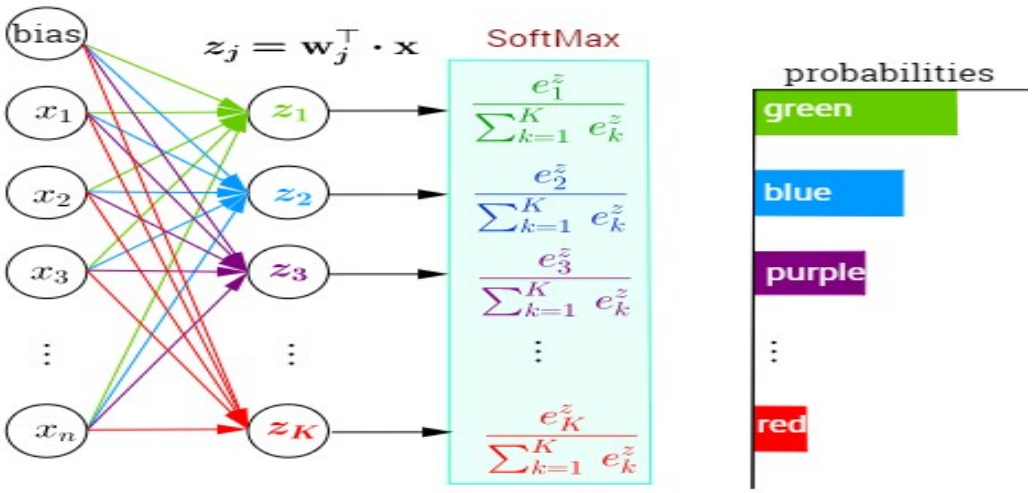



**Figure 12** Softmax

The output of a neuron in a softmax layer depends on the outputs of all the other neurons in its layer. This is because we require the sum of all the outputs to be equal to 1. Letting $z_i$ be the logit of the i th softmax neuron, we can achieve this normalization by setting its output to:

$$y_i = \frac{e^{z_i}}{\sum_j e^{z_j}} \qquad (2)$$

A strong prediction would have a single entry in the vector close to 1, while the remaining entries were close to 0. A weak prediction would have multiple possible labels that are more or less equally likely.

### I.2.3.5 Loss Functions

Loss functions [10]quantify how close a given neural network is to the ideal toward which it is training. The idea is simple. We calculate a metric based on the error we observe in the network's predictions. We then aggregate these errors over the entire dataset and average them and now we have a single number representative of how close the neural network is to its ideal.

Looking for this ideal state is equivalent to finding the parameters (weights and biases) that will minimize the "loss" incurred from the errors. In this way, loss functions help re-frame training neural networks as an optimization problem.

We want to train the neuron so that we pick the optimal weights possible the weights that minimize the errors we make on the training examples.

In most cases, these parameters cannot be solved for analytically, but, more often than not, they can be approximated well with iterative optimization algorithms like gradient descent.

In this case, let's say we want to minimize the square error over all of the training examples that we encounter. More formally, if we know that $t^{(i)}$ is the true answer for the $i^{(th)}$ training example and $y^{(i)}$ is the value computed by the neural network, we want to minimize the value of the error function E :

$$E = \frac{1}{2}\sum_i \left(t^{(i)} - y^{(i)}\right)^2 \qquad (3)$$

## I.2.4 Gradient Descent

Gradient is defined as the generalization of the derivative of a function in one dimension to a function f in several dimensions. It is represented as a vector of n partial derivatives of the function f. It is useful in optimization in that the gradient points in the direction of the greatest rate of increase of the function for which the magnitude is the slope of the graph in that direction. Gradient descent calculates the slope of the loss function by taking a derivative, which should be a familiar term from calculus. On a two-dimensional loss function, the derivative would simply be the tangent of any point on the parabola.

As we know from trigonometry, a tangent is just a ratio: the opposite side (which measures vertical change) over the adjacent side (which measures horizontal change) of a right triangle.

One definition of a curve is a line of constantly changing slope. The slope of each point on the curve is represented by the tangent line touching that point. Because slopes are derived from two points, how exactly does one find the slope of one point on a curve? We find the derivative by calculating the slope of a line between two points on the curve separated by a small distance and then slowing decreasing that distance until it approaches zero. In calculus, this is a limit.

Let's visualize how we might minimize the squared error over all of the training examples by simplifying the problem. Let's say our linear neuron only has two inputs (and thus only two

weights, w 1 and w 2 ). Then we can imagine a three-dimensional space where the horizontal dimensions correspond to the weights w 1 and w 2 , and the vertical dimension corresponds to the value of the error function E . In this space , points in the horizontal plane correspond to different settings of the weights, and the height at those points corresponds to the incurred error. If we consider the errors we make over all possible weights, we get a surface in this three-dimensional space, in particular, a quadratic bowl as shown in (Figure 13)

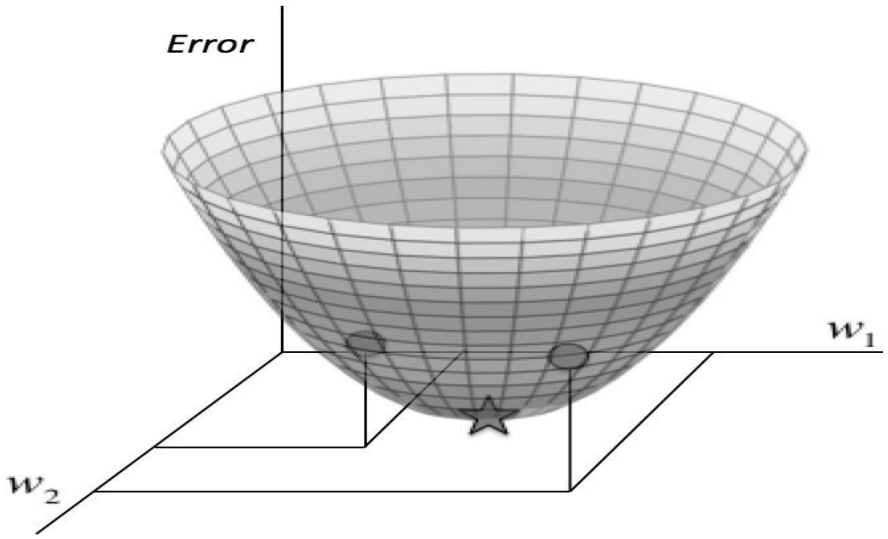



**Figure 13** The quadratic error surface for a linear neuron

Now we can develop a high-level strategy for how to find the values of the weights
that minimizes the error function. Suppose we randomly initialize the weights of our network so we find ourselves somewhere on the horizontal plane. By evaluating the gradient at our current position, we can find the direction of steepest descent, and we can take a step in that direction. Then we'll find ourselves at a new position that's closer to the minimum than we were before. We can reevaluate the direction of steepest descent by taking the gradient at this new position and taking a step in this new direction. It's easy to see that, as shown in Figure 14 , following this strategy will eventually get us to the point of minimum error. This algorithm is known as gradient descent, and it is used to tackle the problem of training individual neurons and the more general challenge of training entire networks.

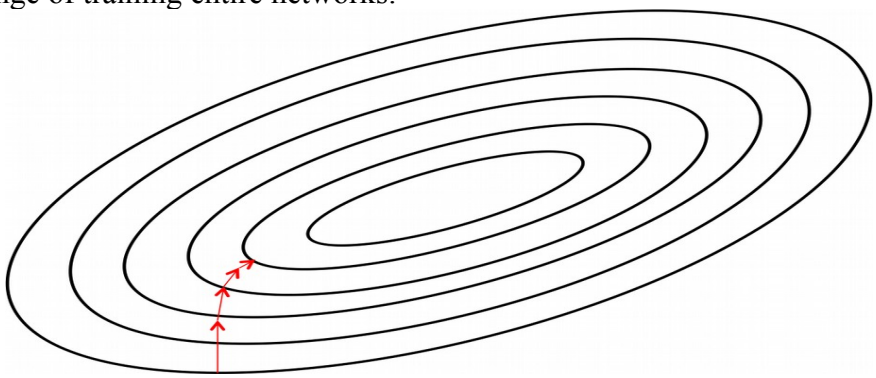


**Figure 14** Visualizing the error surface as a set of contours

learning algorithms also require a couple of additional parameters to carry out the training process. One of these so-called hyperparameters is the learning rate.
In practice, at each step of moving perpendicular to the contour, we need to determine how far we want to walk before recalculating our new direction. This distance needs to depend on the steepness of the surface. Why? The closer we are to the minimum, the shorter we want to step forward. We know we are close to the minimum, because the surface is a lot flatter, so we can use the steepness as an indicator of how close we are to the minimum. However, if our error surface is rather mellow, training can potentially take a large amount of time. As a result, we often multiply the gradient by a factor **ε** , the learning rate. Picking the learning rate is a hard problem , if we pick a learning rate that's too small, we risk taking too long during the training process. But if we pick a learning rate that's too big (see Figure 15), we'll mostly likely start diverging away from the minimum.[10]

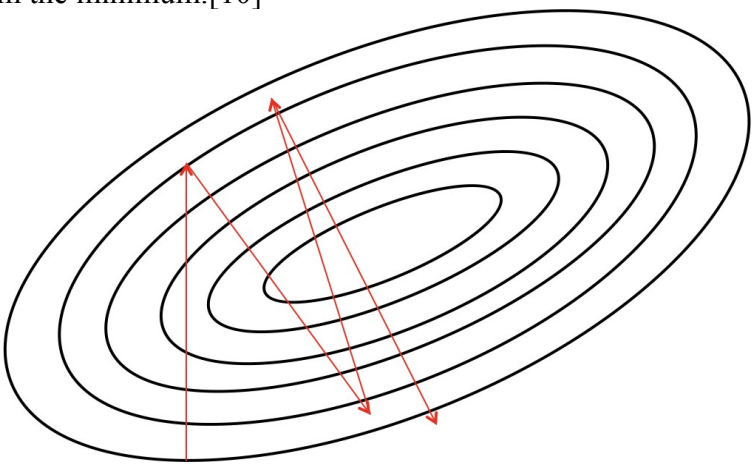


**Figure 15** Convergence is difficult when our learning rate is too large

In order to calculate how to change each weight, we evaluate the gradient, which is essentially the partial derivative of the error function with respect to each of the weights. In other words, we want:

$$\Delta w_k = -\epsilon \frac{\partial E}{\partial w_k} = -\epsilon \frac{\partial (\frac{1}{2} \sum_i (t^{(i)} - y^{(i)})^2)}{\partial w_k} = \sum_i \epsilon (t^{(i)} - y^{(i)}) \frac{\partial y_i}{\partial w_k} = \sum_i \epsilon X_k^{(i)} (t^{(i)} - y^{(i)}) \quad (4)$$

Applying this method of changing the weights at every iteration, we are finally able to utilize gradient descent.

### I.2.4.1 Gradient Descent with Sigmoidal Neurons [10]

training neurons and neural networks that utilize nonlinearities like sigmoidal neuron as a model, we merely need to assume that the bias is a weight on an incoming connection whose input value is always one.
Let's recall the mechanism by which logistic neurons compute their output value from their inputs:

$$z = \sum_k w_k x_k$$
$$y = \frac{1}{1 + e^{-z}} \quad (5)$$

The neuron computes the weighted sum of its inputs, the logit z . It then feeds its logit into the input function to compute y , its final output. Fortunately for us, these functions have very nice derivatives, which makes learning easy! For learning, we want to compute the gradient of the

error function with respect to the weights. To do so, we start by taking the derivative of the logit with respect to the inputs and the weights:

$$\begin{aligned}\frac{\partial z}{\partial w_k} &= x_k \\ \frac{\partial z}{\partial x_k} &= w_k\end{aligned} \quad (6)$$

Also, quite surprisingly, the derivative of the output with respect to the logit is quite simple if you express it in terms of the output:

$$\frac{\partial y}{\partial z} = \frac{e^{-z}}{(1+e^{-z})^2} = \frac{1}{1+e^{-z}}\frac{e^{-z}}{1+e^{-z}} = \frac{1}{1+e^{-z}}\left(1-\frac{1}{1+e^{-z}}\right) = y(1-y) \quad (7)$$

We then use the ***chain rule*** to get the derivative of the output with respect to each weight:

$$\frac{\partial y}{\partial w_k} = \frac{\partial y}{\partial z}\frac{\partial z}{\partial w_k} = x_k y(1-y) \quad (8)$$

Putting all of this together, we can now compute the derivative of the error function with respect to each weight:

$$\frac{\partial E}{\partial w_k} = \sum_i \frac{\partial E}{\partial y^{(i)}}\frac{\partial y^{(i)}}{\partial w_k} = -\sum_i x_k^{(i)} y^{(i)}(1-y^{(i)})(t^{(i)}-y^{(i)}) \quad (9)$$

Thus, the final rule for modifying the weights becomes:

$$\Delta w_k = \sum_i \epsilon x_k^{(i)} y^{(i)}(1-y^{(i)})(t^{(i)}-y^{(i)}) \quad (10)$$

### I.2.5 The Backpropagation Algorithm [10]

to training multilayer neural networks we'll use an approach known as backpropagation, pioneered by David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams . [12] So what's the idea behind backpropagation? We don't know what the hidden units ought to be doing, but what we can do is compute how fast the error changes as we change a hidden activity. From there, we can figure out how fast the error changes when we change the weight of an individual connection. Essentially, we'll be trying to find the path of steepest descent! The only catch is that we're going to be working in an extremely high-dimensional space. We start by calculating the error derivatives with respect to a single training example.

Each hidden unit can affect many output units. Thus, we'll have to combine many separate effects on the error in an informative way. Our strategy will be one of dynamic programming. Once we have the error derivatives for one layer of hidden units, we'll use them to compute the error derivatives for the activities of the layer below. And once we find the error derivatives for the activities of the hidden units, it's quite easy to get the error derivatives for the weights leading into a hidden unit. We'll redefine some notation for ease of discussion and refer to the (Figure 16 ) The subscript we use will refer to the layer of the neuron. The symbol y will refer to the activity of a neuron, as usual. Similarly, the symbol z will refer to the logit of the neuron.

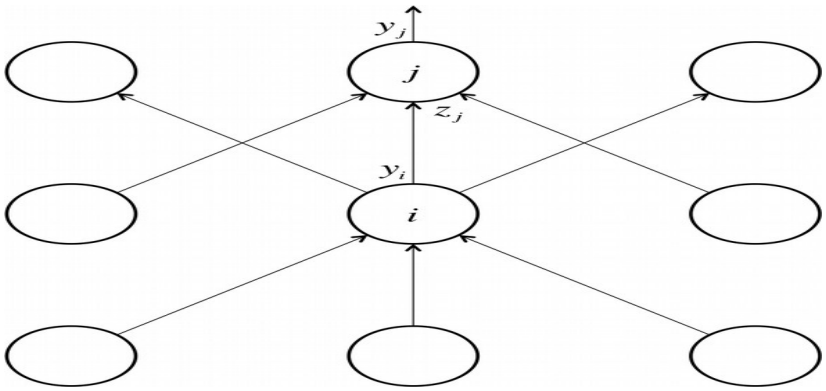



**Figure 16** Reference diagram for the derivation of the backpropagation algorithm

We start by taking a look at the base case of the dynamic programming problem. Specifically, we calculate the error function derivatives at the output layer:

$$E=\frac{1}{2}\sum_{j\in output}(t_j-y_j)^2\Rightarrow\frac{\partial E}{\partial y_j}=-(t_j-y_j) \quad (10)$$

Now we tackle the inductive step. Let's presume we have the error derivatives for layer j . We now aim to calculate the error derivatives for the layer below it, layer i To do so, we must accumulate information about how the output of a neuron in layer i affects the logits of every neuron in layer j . This can be done as follows, using the fact that the partial derivative of the logit with respect to the incoming output data from the layer beneath is merely the weight of the connection $w_{ij}$

$$\frac{\partial E}{\partial y_i}=\sum_j\frac{\partial E}{\partial z_j}\frac{\partial z_j}{\partial y_i}=\sum_i w_{ij}\frac{\partial E}{\partial z_j} \quad (11)$$

Furthermore, we observe the following:

$$\frac{\partial E}{\partial z_j}=\frac{\partial E}{\partial y_j}\frac{\partial y_j}{\partial z_j}=y_j(1-y_j)\frac{\partial E}{\partial y_j} \quad (12)$$

Combining these two together, we can finally express the error derivatives of layer i in terms of the error derivatives of layer j :

$$\frac{\partial E}{\partial y_i}=\sum_j w_{ij}y_j(1-y_j)\frac{\partial E}{\partial y_j} \quad (13)$$

Then once we've gone through the whole dynamic programming routine, having filled up the table appropriately with all of our partial derivatives (of the error function with respect to the hidden unit activities), we can then determine how the error changes with respect to the weights. This gives us how to modify the weights after each training example:

$$\frac{\partial E}{\partial w_{ij}}=\frac{\partial z_j}{\partial w_{ij}}\frac{\partial E}{\partial z_j}=y_i y_j(1-y_j)\frac{\partial E}{\partial y_j} \quad (14)$$

Finally, to complete the algorithm, just as before, we merely sum up the partial derivatives over all the training examples in our dataset. This gives us the following modification formula:

$$\Delta W_{ij}=-\sum_{k\in dataset}\epsilon y_i^{(k)}y_j^{(k)}(1-y_j^{(k)})\frac{\partial E^{(k)}}{\partial y_j^{(k)}} \quad (15)$$

### I.2.6 Convolutional Neural Networks (CNNs / ConvNets) [13]

Convolutional Neural Network also known as ConvNet or CNN are inspired by the biological visual cortex. The visual cortex has small regions of cells that are sensitive to specific regions of the visual field. Different neurons in the brain respond to different features. For example, certain neurons fire only in the presence of lines of a certain orientation, some neurons fire when exposed to vertical edges and some when shown horizontal or diagonal edges, this idea of certain neurons having a specific task is the basis behind ConvNets.

ConvNets have shown excellent performance on several applications such as image classification, object detection, speech recognition, natural language processing, and medical image analysis. Convolutional neural networks are powering core of computer vision that has many applications which include self-driving cars, robotics, and treatments for the visually impaired. The main concept of ConvNets is to obtain local features from input (usually an image) at higher layers and combine them into more complex features at the lower layers.

#### I.2.6.1 Convolution Operation

Convolution is a mathematical operation performed on two functions and is written as (f * g), where f and g are two functions.

The output of the convolution operation for domain n is defined as

$$(f*g)(n)=\sum_{m} f(m)g(n-m) \quad (16)$$

For time-domain functions, n is replaced by t. The convolution operation is commutative in nature, so it can also be written as

$$(f*g)(n)=\sum_{m} f(n-m)\,g(m) \quad (17)$$

Convolution operation is one of the important operations used in digital signal processing and is used in many areas which includes statistics, probability, natural language processing, computer vision, and image processing and can be applied to higher dimensional functions as well.

It can be applied to a two-dimensional function by sliding one function on top of another, multiplying and adding. Convolution operation can be applied to images (treated as two-dimensional functions) to perform various transformations.

An example of a two-dimensional filter, a two-dimensional input, and a two-dimensional feature map is shown in Figure 17 Let the 2D input (i.e., 2D image) be denoted by A, the 2D filter of size m × n be denoted by K, and the 2D feature map be denoted by F. Here, the image A is convolved with the filter K and produces the feature map F. This convolution operation is denoted by A*K and is mathematically given as

$$f(i,j)=(A*k)(i,j)=\sum_{m}\sum_{n} A(m,n)k(i-m,j-n) \quad (18)$$

The convolution operation is commutative in nature, so we can write

$$f(i,j)=(A*k)(i,j)=\sum_m \sum_n A(i-m,j-n)k(m,n) \quad (19)$$

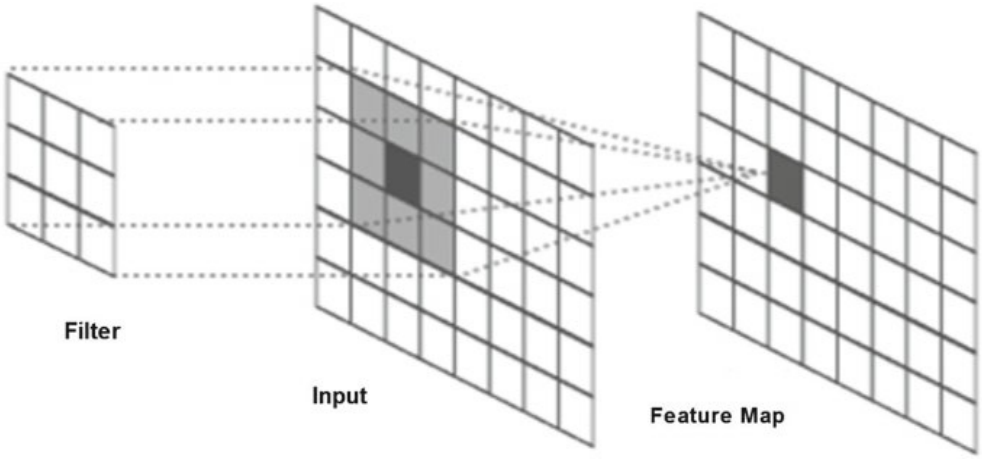



**Figure 17** Convolution operation

The kernel K is flipped relative to the input. If the kernel is not flipped, then convolution operation will be same as cross-correlation operation that is given below:

$$f(i,j)=(A*k)(i,j)=\sum_m \sum_n A(i+m,j+n)k(m,n) \quad (20)$$

Many CNN libraries use cross-correlation function as convolution function because cross-correlation is more convenient to implement than convolution operation itself.

### I.2.6.2 Architecture of CNN

In a traditional neural network, Each hidden layer is made up of a number of neurons, where each neuron is fully connected to all neurons in the preceding layer. The problem with the fully connected neural network is that its densely connected network architecture does not scale well to large images. For large images, the most preferred approach is to use convolutional neural network.

Convolutional neural network is a deep neural network architecture designed to process data that has a known, grid-like topology, for example, 1D time-series data, 2D or 3D data such as images and speech signal, and 4D data such as videos. ConvNets have three key features: ***local receptive field, weight sharing, and subsampling (pooling).***

**(i) Local Receptive Field**

In a traditional neural network, each neuron or hidden unit is connected to every neuron in previous layer or every input unit. Convolutional neural networks, however have local receptive field architecture, , each hidden unit can only connect to a small region of the input called local receptive field. This is accomplished by making the filter/weight matrix smaller than the input. With local receptive field, neurons can extract elementary visual features like edges, corners, end points, etc.

**(ii) Weight Sharing**
Weight sharing refers to using the same filter/weights for all receptive fields in a layer. In ConvNet, since the filters are smaller than the input, each filter is applied at every position of the input , same filter is used for all local receptive fields.

**(iii) Subsampling (Pooling)**
Subsampling reduces the spatial size of the input, thus reducing the parameters in the network. There are few subsampling techniques available, and the most common subsampling technique is max-pooling.

ConvNet consists of a sequence of different types of layers to achieve different tasks, a typical convolutional neural network consists of the following layers:

- Convolutional layer,
- Activation function layer (ReLU),
- Pooling layer,
- Fully connected layer and

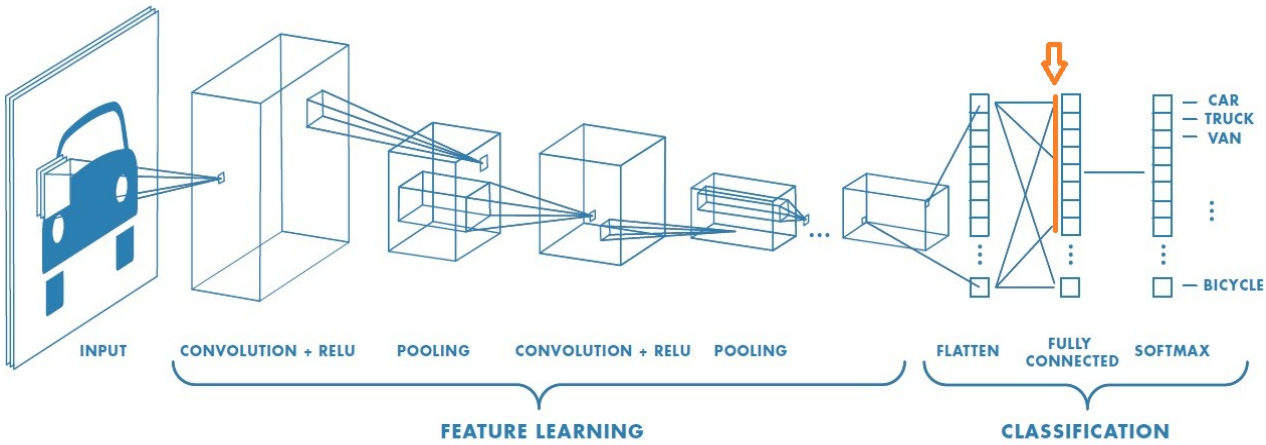



**Figure 18** use of ConvNet in deep learning

These layers are stacked up to make a full ConvNet architecture. Convolutional and activation function layers are usually stacked together followed by an optional pooling layer. Fully connected layer makes up the last layer of the network, and the output of the last fully connected layer produces the class scores of the input image as showing in figure 18.
In addition to these main layers mentioned above, ConvNet may include optional layers like batch normalization layer to improve the training time and dropout layer to address the overfitting issue.

**I.2.6.2.1 Convolution Layer**
Convolution layer is the core building block of a convolutional neural network which uses convolution operation (represented by *) in place of general matrix multiplication. Its parameters consist of a set of learnable filters also known as kernels. The main task of the convolutional layer is to detect features found within local regions of the input image that are common throughout the dataset and mapping their appearance to a feature map.
A feature map is obtained for each filter in the layer by repeated application of the filter across subregions of the complete image, convolving the filter with the input image, adding a bias term, and then applying an activation function.

The input area on which a filter is applied is called local receptive field.
The size of the receptive field is same as the size of the filter, (Figure 19 ) shows how a filter (T-shaped) is convolved with the input to get the feature map.
Feature map is obtained after adding a bias term and then applying a nonlinear function to the output of the convolution operation. The purpose of nonlinearity function is to introduce nonlinearity in the ConvNet model.

**I.2.6.2.1.1Filters/Kernels**
The weights in each convolutional layer specify the convolution filters and there may be multiple filters in each convolutional layer. Every filter contains some feature like edge, corner, etc. and during forward pass, each filter is slid across the width and height of the input generating feature map of that filter.

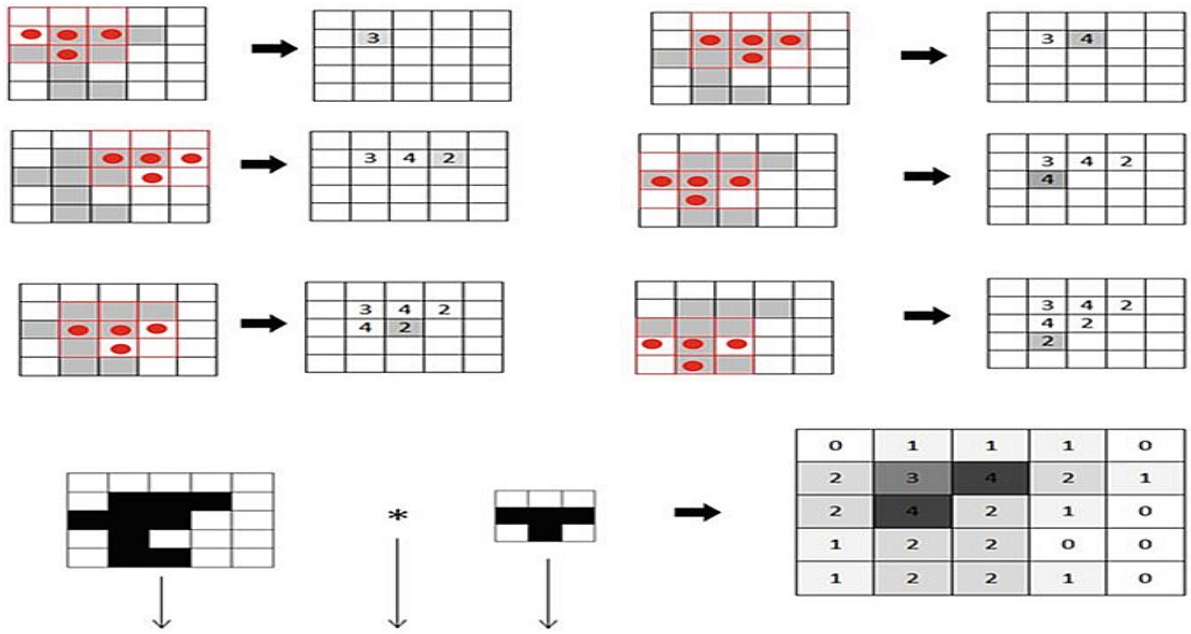



**Figure 19** Example of convolution operation

**I.2.6.2.1.2 Hyperparameters**
Convolutional neural network architecture has many hyperparameters that are used to control the behavior of the model. Some of these hyperparameters control the size of the output while some are used to tune the running time and memory cost of the model, the four important hyperparameters in the convolution layer of the ConvNet are given below:

**a. Filter Size:** Filters can be of any size greater than 2 × 2 and less than the size ofthe input but the conventional size varies from 11 × 11 to 3 × 3. The size of a filter is independent of the size of input.

**b. Number of Filters:** There can be any reasonable number of filters. AlexNet used 96 filters of size 11 × 11 in the first convolution layer. VGGNet used 96 filters of size 7 × 7, and another variant of VGGNet used 64 filters of size 11 × 11 in first convolution layer.

**c. Stride:** It is the number of pixels to move at a time to define the local receptive field for a filter. Stride of one means to move across and down a single pixel. The value of stride should not be too small or too large. Too small stride will lead to heavily overlapping receptive fields and too large value will overlap less and the resulting output volume will have smaller dimensions spatially.

**d. Zero Padding:** This hyperparameter describes the number of pixels to pad the input image with zeros. Zero padding is used to control the spatial size of the output volume.

Each filter in the convolution layer produces a feature map of size ([A − K + 2P]/S)+ 1 where A is the input volume size, K is the size of the filter, P is the number of padding applied and S is the stride.

Suppose the input image has size 128 × 128, and 5 filters of size 5 × 5 are applied, with single stride and zero padding, i.e., A # 128, F # 5, P # 0 and S # 1. The number of feature maps produced will be equal to the number of filters applied, i.e., 5 and the size of each feature map will be ([128 − 5 + 0]/1) + 1 # 124. Therefore, the output volume will be 124 × 124 × 5.

### I.2.6.2.2 Pooling Layer

In ConvNets, the sequence of convolution layer and activation function layer is followed by an optional pooling or down-sampling layer to reduce the spatial size of the input and thus reducing the number of parameters in the network.

A pooling layer takes each feature map output from the convolutional layer and down-samples it, pooling layer summarizes a region of neurons in the convolution layer. There are few pooling techniques available and the most common pooling technique is max-pooling.

Max-pooling simply outputs the maximum value in the input region.

The input region is a subset of input (usually 2 × 2). For example, if input region is of size 2 × 2, the max-pooling unit will output the maximum of the four values as shown in Figure 20 Other options for pooling layers are average pooling and L2-norm pooling.

Pooling layer operation discards less significant data but preserves the detected features in a smaller representation. The intuitive reasoning behind pooling operation is that feature detection is more important than feature's exact location. This strategy works well for simple and basic problems but it has its own limitations and does not work well for some problems.

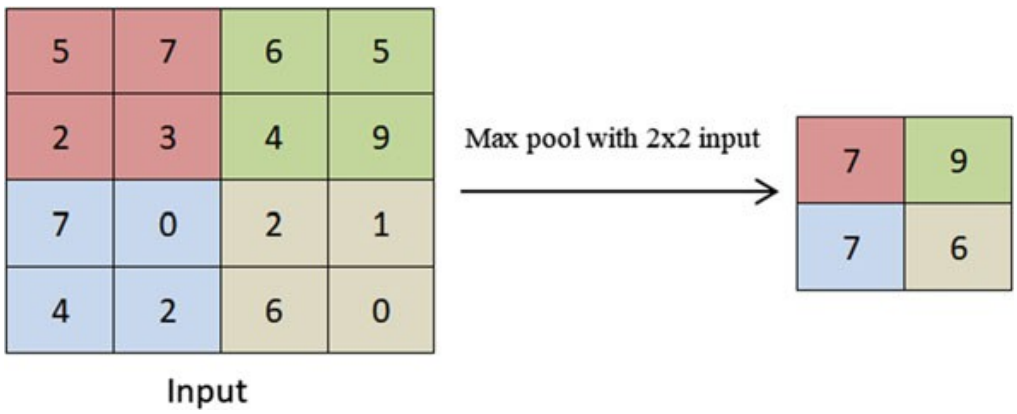



**Figure20** Max-pooling

## I.3 Regularization for Deep Learning

"A central problem in machine learning especially in deep learning is how to make an algorithm that will perform well not just on the training data, but also on new inputs. Many strategies used in machine learning are explicitly designed to reduce the test error, possibly at the expense of increased training error. These strategies are known collectively as regularization. " [14].

Regularization is often done by putting some extra constraints on a machine learning model, such as adding restrictions on the parameter values or by adding extra terms in the objective function (penalizes the weight matrices) that can be thought of as corresponding to a soft constraint on the parameter values. If chosen correctly these can lead to a reduced testing error. An effective regularizer is said to be the one that makes a profitable trade by reducing variance significantly while not overly increasing the bias.
Regularization helps us control our model capacity, ensuring that our models are better at making (correct) classifications on data points that they were not trained on, which we call the ability to generalize. If we don't apply regularization, our classifiers can easily become too complex and overfit to our training data, in which case we lose the ability to generalize to our testing data .
However, too much regularization can be a bad thing. We can run the risk of underfitting , in which case our model performs poorly on the training data and is not able to model the relationship between the input data and output class labels (because we limited model capacity too much).
Our goal when building deep learning classifiers is to obtain a model that fit our training data nicely, but avoid overfitting. Regularization can help us obtain this type of desired fit (Figure 21)

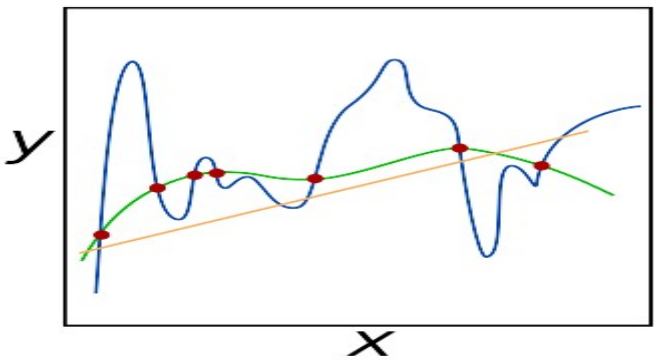



**Figure 21**example of underfitting (orange line), overfitting (blue line), and generalizing (green line)

Overfitting and underfitting are detected by looking to the training and test error behave (Figure 22) At the left end of the graph, training error and generalization error are both high. This is the underfitting regime As we increase capacity, training error decreases, but the gap between training and generalization error increases. Eventually, the size of this gap outweighs the decrease in training error, and we enter the overfitting regime, where capacity is too large, above the optimal capacity

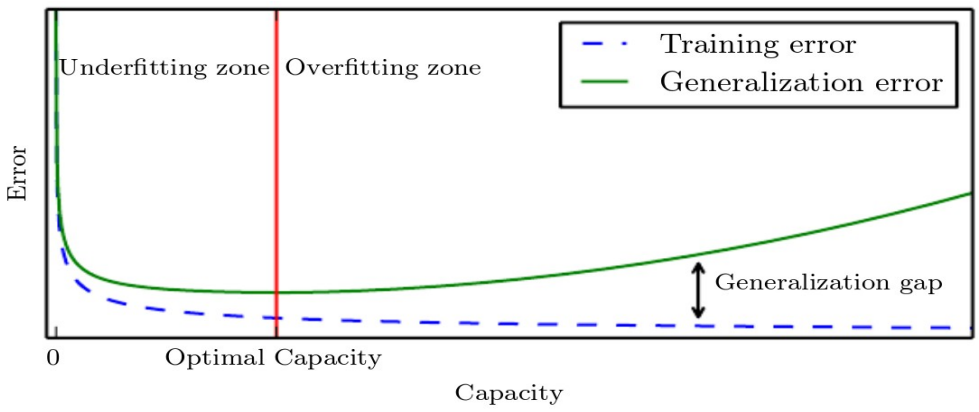



**Figure 22** Typical relationship between capacity and error.

There are many Regularization technique in use some of them are briefly describe below.

### I.3.1 Dataset Augmentation

The best way to make a machine learning model generalize better is to train it on more data. Of course, in practice, the amount of data we have is limited. One way to get around this problem is to create fake data and add it to the training set.

Dataset augmentation is a common practice to virtually increase the size of training dataset, and is also used as a regularization technique, making the model more robust to slight changes in the input data has been a particularly effective technique for a specific classification problem: object recognition. Images are high dimensional and include an enormous range of factors of variation, many of which can be easily simulated.

Operations like translating the training images a few pixels in each direction can often greatly improve generalization, even if the model has already been designed to be partially translation invariant by using the convolution and pooling techniques.

Many other operations, such as rotating the image or scaling the image, have also proved quite effective (see Figure 23 ).

One must be careful not to apply transformations that would change the correct class. For example, optical character recognition tasks require recognizing the difference between “b” and “d” and the difference between “6” and “9,” so horizontal flips and 180° rotations are not appropriate ways of augmenting datasets for these tasks.

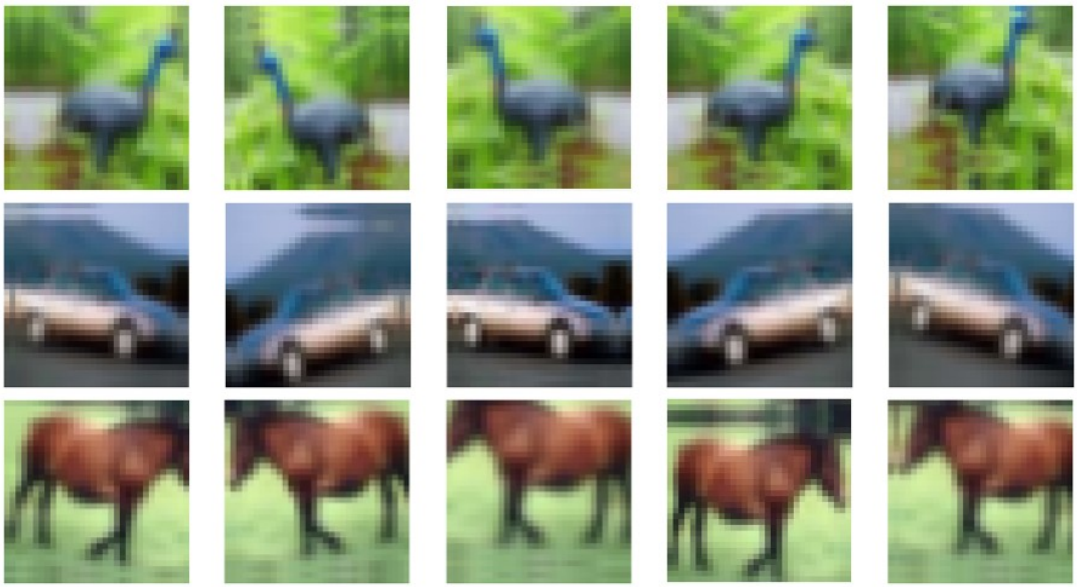


**Figure 23** Example of Data Augmentation on the CIFAR10 dataset

“Injecting noise in the input to a neural network (Sietsma and Dow, 1991) can also be seen as a form of data augmentation. For many classification and even some regression tasks, the task should still be possible to solve even if small random noise is added to the input. Neural networks prove not to be very robust to noise, however (Tang and Eliasmith, 2010). One way to improve the robustness of neural networks is simply to train them with random noise applied to their inputs. ” [14]

### I.3.2 Dropout [14]

In order to overcome the problem of overfitting, a dropout layer can be introduced in the model in which some neurons along with their connections are randomly dropped from the network during training (See Figure 24). A reduced network is left; incoming and outgoing edges to a dropped-out node are also removed. Only the reduced network is trained on the data in that stage.

The removed nodes are then reinserted into the network with their original weights. Dropout notably reduces overfitting and improves the generalization of the model.

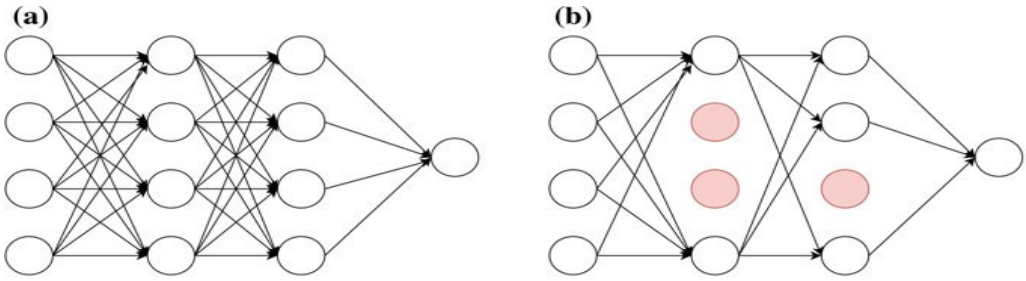



**Figure 24** a simple neural network, b neural network after dropout

## I.4 Deep Learning Architectures

Many supervised deep learning architectures have evolved over the last few years, achieving top scores on many tasks. In this paragraph we will discussed briefly one of the most recent supervised CNN architectures proposed by researchers ResNet .

### I.4.1 ResNet [1]

As the number of layers of deep networks increases, its accuracy improves and the accuracy saturates once the network has converged. However, if the depth is further increased, then the performance starts getting degraded rapidly. This degradation is caused by adding more layers to an already converged deep model which results in higher training error. Thus, there is a need for a strategy that obtains an optimal deep network for a given application.

ResNet was proposed with a residual learning framework that lets new layers to fit a residual mapping. It is easier to push the residual to zero when a model has converged than to fit the mapping by a stack of nonlinear layers.

Given an underlying mapping H(x) to be fit by a few stacked layers, where x is the input to these layers, the residual learning uses the residual function.

$$F(x)=H(x)-x \qquad (21)$$

It is easier to optimize the residual mapping than to optimize the original, and it can be realized by a feedforward neural network with shortcut connection as shown in (Figure 25) The shortcut link simply accomplishes identity mapping, and the output of the shortcut link is added to the outcomes of the stacked layers .

The identity shortcut link does not add calculation complexity or parameters.

The residual function F uses a stack of 2 or 3 layers (more layers are also possible) as shown in (Figure 26 ), the building block is defined by

$$y=F(x,\{W_i\})+x \qquad (22)$$

where x and y represent the input and output vectors of layers considered.

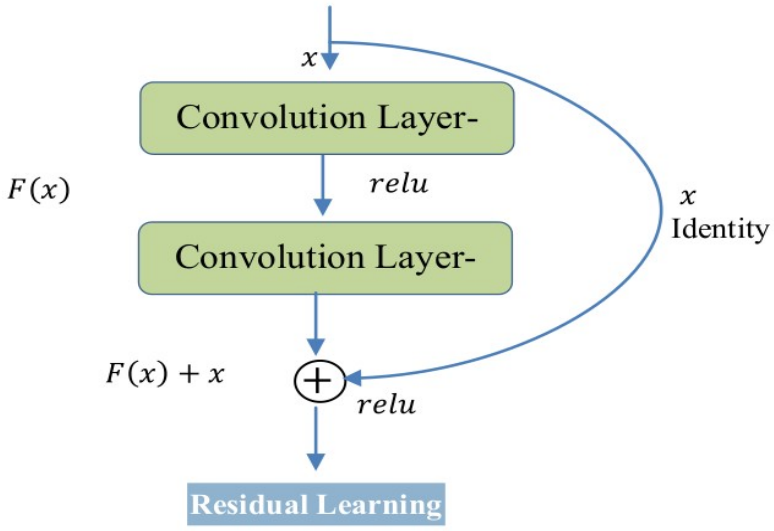



**Figure 25** ResNet residual learning building block

The function F(x, {W i }) represents the residual mapping which is to be learnt. The linear projection $W_s$ is performed by a shortcut link to match the dimensions as in

$$y = F(x, \{W_i\}) + W_s x \quad (23)$$

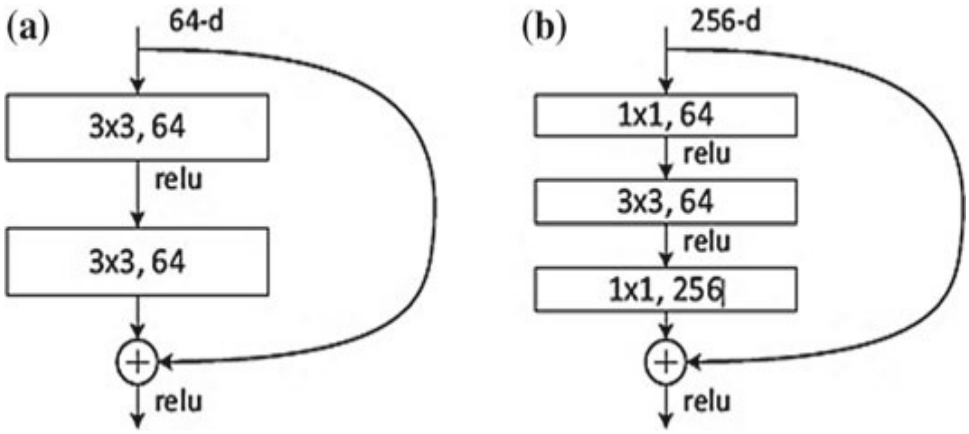



**Figure 26** a ResNet building block, b "Bottleneck" ResNet building block

The architecture diagram of ResNet-34 is shown in (Figure 27)

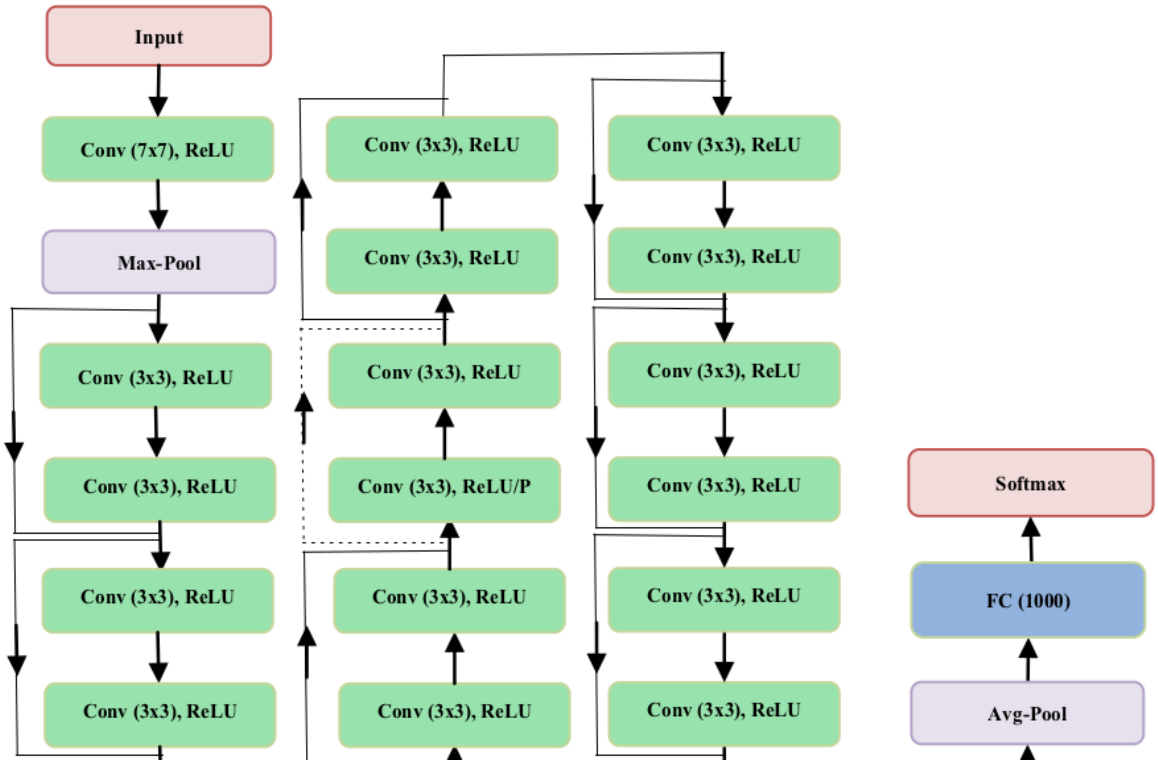



**Figure 27** Architecture diagram of ResNet-34 layers

### I.4.2 Capsule Network [13]

Every day there are improvements in deep learning new Regularization technique are invented and new architecture are created but the main innovation came from a work conducted by the godfather of deep learning Geoffrey E. Hinton how published a scientific paper Dynamic Routing Between Capsules **[2] .**

The human brain has a mechanism to route image data to parts of the brain where it is perceived. Convolutional neural networks use layers of filters to extract high-level features from image data but the routing mechanism is absent in it.
Capsule Network (CapsNet) has been proposed that provides a routing mechanism. A CapsNet can have many capsule layers, where each layer is comprised of a number of capsules. A capsule is a group of neurons that can perform computations on their inputs and then compute an output in the form of a vector. The computations of the neurons within a capsule can represent various properties like pose, size, position, deformation, orientation, etc. of an entity (object or a part of an object) that is present in a given image. CapsNet uses the length of the output vector to represent the existence of an entity. The length of the output vector of a capsule is not allowed to exceed 1 by applying a nonlinearity that leaves the orientation of the vector unchanged but scales down its magnitude.

A CapsNet proposed incorporating a routing mechanism between two capsule layers. The routing mechanism makes a capsule in one layer to communicate to some or all capsules in the next layer.

Architecture diagram of a simple capsule network is shown in Figure 28 and Table 1 gives details of various layers of the capsule network.

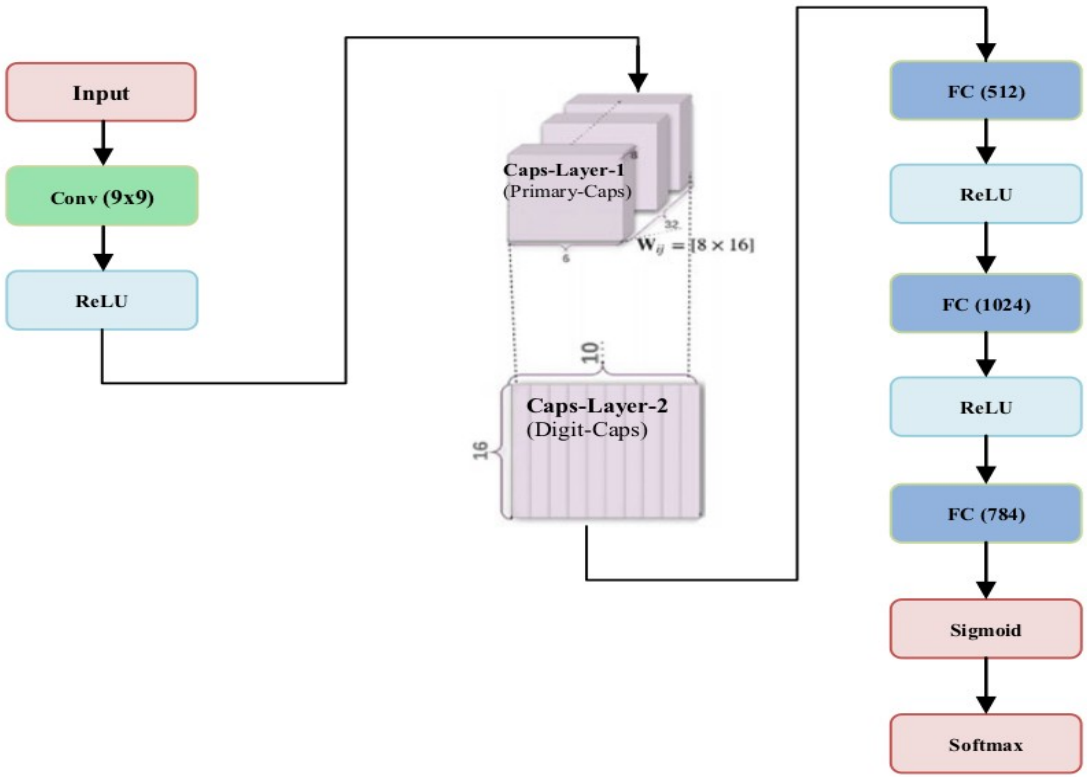



**Figure28** Architecture diagram of a simple capsule network

**Table 1** detail of various layers of simple capsule network

| Layer name | Input size | filter size | filters | capsules | stride | padding | future maps | Output size |
|---|---|---|---|---|---|---|---|---|
| Conv 1 | 28 x 28 | 9 x 9 | 256 | | 1 | 0 | 256 | 20 x 20 |
| Capsule layer 1 | 20 x 20 | 9x 9 | 8 | 32 | 2 | 0 | 32 (each with depth 8) | 6 x 6 |
| Capsule layer 2 | 6 x 6 | 10 digital capsule one for each digit | | | | | | |
| fully connected | 512 Neurones | | | | | | | |
| fully connected | 1024 Neurones | | | | | | | |
| fully connected | 784 (which after reshaping give back 28 x 28 decoded image) | | | | | | | |
| softmax | 10 classes | | | | | | | |

The first layer of this simple CapsNet is a convolutional layer that uses 256 filters of size of 9×9 with a stride of 1. It uses ReLU activation function. This layer converts pixel intensities into features which are used as inputs to the primary capsules.
The second layer of the CapsNet is the first capsule layer. This layer has 32 primary capsules. Each primary capsule has eight convolutional filters of size 9 × 9 used with a stride of two. Each primary capsule receives all 256 feature maps of size 20 × 20 produced by the first layer.
The primary capsules' layer produces 32 feature maps of size 6 × 6. Each feature map has a depth of 8, each feature map is an 8D vector.
The next layer is the second capsule layer which has one 16D capsule for each digit class. Each capsule in this layer receives input from all the capsules in the first capsule layer.
The simple CapsNet has a routing mechanism between the two capsule layers only. Initially, a capsule output from the first capsule layer is sent to all capsules in the second capsule layer with equal probability. A dynamic routing mechanism is used to ensure that the output of a capsule is sent to the appropriate capsules in the next capsule layer and is determined by coupling coefficients.
The coupling coefficients between a capsule in the first layer and all the capsules in the next layer sum to 1 and are determined by a routing softmax.
The coupling coefficients can be learnt discriminatively at the same time as all other weights.

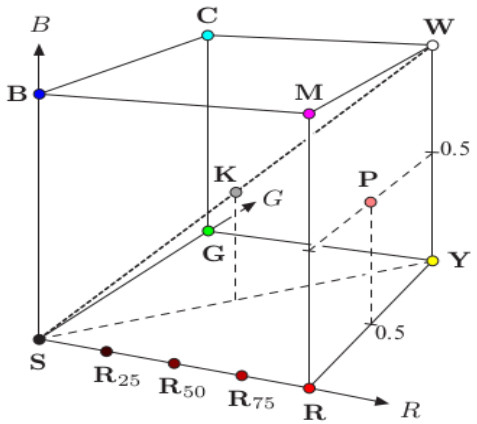
# CHAPTER II
# COLOR SYSTEMS

Color images are involved in every aspect of our lives, where they play an important role in everyday activities such as television, photography, and printing. Color perception is a fascinating and complicated phenomenon that has occupied the interest of scientists, psychologists, philosophers, and artists for hundreds of years . In this chapter,we focus on those technical aspects of color that are most important for working with digital color images. Our emphasis will be on understanding the various representations of color and correctly utilizing them when converting the Cifar10 dataset.

## II.1 RGB Color Images[15]

The RGB color schema encodes colors as combinations of the three primary colors: red (R), green (G), and blue (B). This scheme is widely used for transmission, representation, and storage of color images on both analog devices such as television sets and digital devices such as computers, digital cameras, and scanners. For this reason, many image-processing and graphics programs use the RGB schema as their internal representation for color images, and most language libraries use it as their standard image representation.

RGB is an additive color system, which means that all colors start with black and are created by adding the primary colors. You can think of color formation in this system as occurring in a dark room where you can overlay three beams of light—one red, one green, and one blue—on a sheet of white paper. To create different colors, you would modify the intensity of each of these beams independently. The distinct intensity of each primary color beam controls the shade and brightness of the resulting color. The colors gray and white are created by mixing the three primary color beams at the same intensity. A similar operation occurs on the screen of a color television or CRT based computer monitor, where tiny, close-lying dots of red, green, and blue phosphorous are simultaneously excited by a stream of electrons to distinct energy levels (intensities), creating a seemingly continuous color image.




| Point | Color | RGB Value R | G | B |
|---|---|---|---|---|
| S | Black | 0.00 | 0.00 | 0.00 |
| R | Red | 1.00 | 0.00 | 0.00 |
| Y | Yellow | 1.00 | 1.00 | 0.00 |
| G | Green | 0.00 | 1.00 | 0.00 |
| C | Cyan | 0.00 | 1.00 | 1.00 |
| B | Blue | 0.00 | 0.00 | 1.00 |
| M | Magenta | 1.00 | 0.00 | 1.00 |
| W | White | 1.00 | 1.00 | 1.00 |
| K | 50% Gray | 0.50 | 0.50 | 0.50 |
| $R_{75}$ | 75% Red | 0.75 | 0.00 | 0.00 |
| $R_{50}$ | 50% Red | 0.50 | 0.00 | 0.00 |
| $R_{25}$ | 25% Red | 0.25 | 0.00 | 0.00 |
| P | Pink | 1.00 | 0.50 | 0.50 |

**Figure 29** Representation of the RGB color space as a three-dimensional unit cube.

The RGB color space can be visualized as a three-dimensional unit cube in which the three primary colors form the coordinate axis. The RGB values are positive and lie in the range [0, C max ]; for most digital images, C max = 255. Every possible color C $_i$ corresponds to a point within the RGB color cube of the form $C_i = (R_i, G_i, B_i)$ where $0 \leq R\,i, G\,i, B\,i \leq C\,max$ .

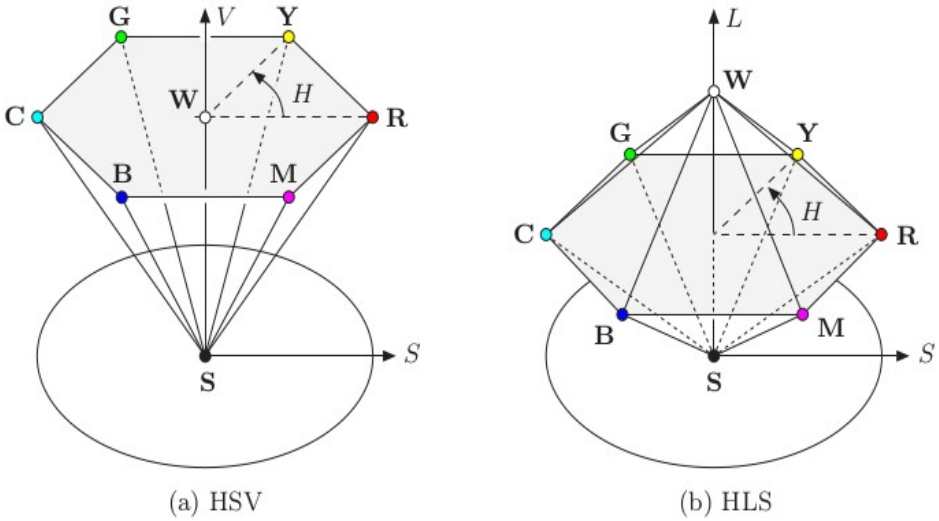
RGB values are often normalized to the interval [0, 1] so that the resulting color space forms a unit cube (Figure 29). The point S = (0, 0, 0) corresponds to the color black, W = (1, 1, 1) corresponds to the color white, and all the points lying on the diagonal between S and W are shades of gray created from equal color components R = G = B. Figure 30 shows a color test image and its corresponding RGB color components, displayed here as intensity images.

**R** **G** **B**

**Figure 30** A color image and its corresponding RGB channels.

### II.2 HSV Color Space[15]

In the HSV color space, colors are specified by the components hue, saturation, and value. It is traditionally shown as an upside-down, six-sided pyramid (Figure31 (a) ), where the vertical axis represents the V (brightness) value, the horizontal distance from the axis the S (saturation) value, and the angle the H (hue) value. The black point is at the tip of the pyramid and the white point lies in the center of the base. The three primary colors red, green, and blue and the pairwise mixed colors yellow, cyan and magenta are the corner points of the base. While this space is often represented as a pyramid, according to its mathematical definition, the space is actually a cylinder, as shown below (Figure 33).




(a) HSV (b) HLS

**Figure 31** HSV color space vs HLS color space

### II.2.1 RGB→HSV

To convert from RGB to the HSV color space, we first find the saturation of the RGB color components R, G, B ∈ [0, $C_{max}$ ], with $C_{max}$ being the maximum component value (typically 255), as

$$S_{HSV}=\begin{cases}\dfrac{C_{rng}}{C_{high}} & for\ C_{high}>0\\ 0 & otherwise\end{cases} \quad (24)$$

and the luminance (value)

$$V_{HSV}=\frac{C_{high}}{C_{max}} \quad (25)$$

with $C_{high}$ , $C_{low}$ , and $C_{rng}$ defined as

$$C_{high}=max(R,G,B),C_{low}=min(R,G,B),C_{rng}=C_{high}-C_{low} \quad (26)$$

Finally, we need to specify the hue value $H_{HSV}$ . When all three RGB color components have the same value (R = G = B), then we are dealing with an achromatic (gray) pixel. In this particular case $C_{rng}$ = 0 and thus the saturation value $S_{HSV}$ = 0, consequently the hue is undefined. To compute $H_{HSV}$ when $C_{rng}$ > 0, we first normalize each component using

$$R' = \frac{C_{high} - R}{C_{rng}}, G' = \frac{C_{high} - G}{C_{rng}}, B' = \frac{C_{high} - B}{C_{rng}} \quad (27)$$

Then, depending on which of the three original color components had the maximal value, we compute a preliminary hue H' as

$$H' = \begin{cases} B' - G' & if\ R = C_{high} \\ R' - B' + 2 & if\ G = C_{high} \\ G' - R' + 4 & if\ B = C_{high} \end{cases} \quad (28)$$

Since the resulting value for H' lies on the interval [−1 . . . 5], we obtain the

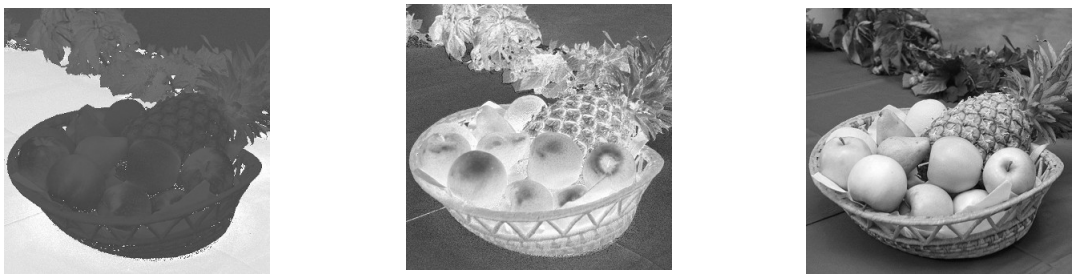
final hue value by normalizing to the interval [0, 1] as

$$H_{HSV} = \frac{1}{6} \cdot \begin{cases} (H' + 6) & for \quad H' < 0 \\ H' & otherwise \end{cases} \quad (29)$$

Hence all three components $H_{HSV}$, $S_{HSV}$, and $V_{HSV}$ will lie within the interval [0, 1]. The hue value $H_{HSV}$ can naturally also be computed in another angle interval, for example in the 0 to 360 ◦ interval using

$$H°_{HSV} = H_{HSV} \ . \ 360 \quad (30)$$

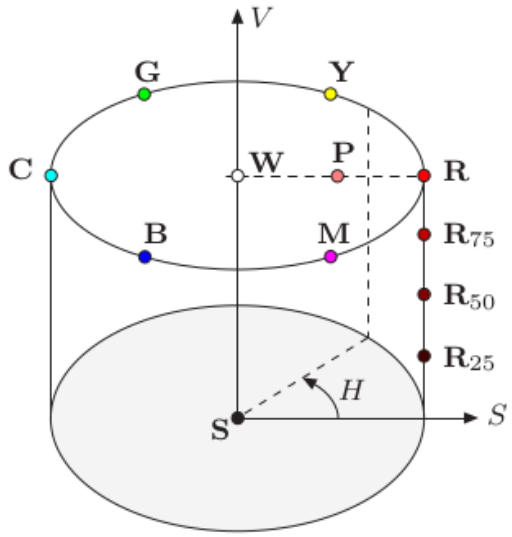
Under this definition, the RGB space unit cube is mapped to a cylinder with height and radius of length 1 , In contrast to the traditional representation all HSB points within the entire cylinder correspond to valid color coordinates in RGB space. The mapping from RGB to the HSV space is nonlinear, as can be noted by examining how the black point stretches , Figure 32 shows the individual HSV components of the test image as grayscale images



**H HSV** **S HSV** **V HSV**

**Figure 32** HSV color components

The darker areas in the $h_{HSV}$ component correspond to the red and yellow colors, where the hue angle is near zero.




RGB/HSV Values

| Pt. | Color | R | G | B | H | S | V |
|---|---|---|---|---|---|---|---|
| S | Black | 0.00 | 0.00 | 0.00 | — | 0.00 | 0.00 |
| R | Red | 1.00 | 0.00 | 0.00 | 0 | 1.00 | 1.00 |
| Y | Yellow | 1.00 | 1.00 | 0.00 | 1/6 | 1.00 | 1.00 |
| G | Green | 0.00 | 1.00 | 0.00 | 2/6 | 1.00 | 1.00 |
| C | Cyan | 0.00 | 1.00 | 1.00 | 3/6 | 1.00 | 1.00 |
| B | Blue | 0.00 | 0.00 | 1.00 | 4/6 | 1.00 | 1.00 |
| M | Magenta | 1.00 | 0.00 | 1.00 | 5/6 | 1.00 | 1.00 |
| W | White | 1.00 | 1.00 | 1.00 | — | 0.00 | 1.00 |
| $R_{75}$ | 75% Red | 0.75 | 0.00 | 0.00 | 0 | 1.00 | 0.75 |
| $R_{50}$ | 50% Red | 0.50 | 0.00 | 0.00 | 0 | 1.00 | 0.50 |
| $R_{25}$ | 25% Red | 0.25 | 0.00 | 0.00 | 0 | 1.00 | 0.25 |
| P | Pink | 1.00 | 0.50 | 0.50 | 0 | 0.5 | 1.00 |

**Figure 33** HSV color space. The illustration shows the HSV color space as a cylinder with the coordinates H (hue) as the angle, S (saturation) as the radius, and V (brightness value) as the distance along the vertical axis, which runs between the black point S and the white point W. The table lists the (R, G, B) and (H, S, V ) values of the color points marked on the graphic. Pure colors (composed of only one or two components) lie on the outer wall of the cylinder (S = 1), as exemplified by the gradually saturated reds (R 25 , R 50 , R 75 , R).

### II.3 HLS Color Space[15]

The HLS color space (hue, luminance, saturation) is very similar to the HSV space, and the hue component is in fact completely identical in both spaces. The luminance and saturation values also correspond to the vertical axis and the radius, respectively, but are defined differently than in HSV space. The common representation of the HLS space is as a double pyramid (Figure 31 (b)), with black on the bottom tip and white on the top. The primary colors lie on the corner points of the hexagonal base between the two pyramids.Even though it is often portrayed in this intuitive way, mathematically the HLS space is again a cylinder (see Figure35 ).

#### II.3.1 RGB→HLS

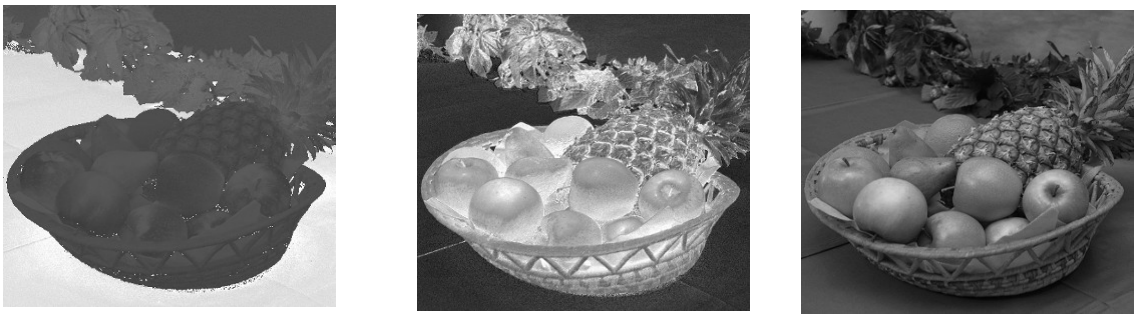
In the HLS model, the hue value $H_{HLS}$ is computed in the same way as in the HSV model

$$H_{HLS} = H_{HSV} \quad (31)$$

The other values, L HLS and S HLS , are computed as follows (for $C_{high}$ , $C_{low}$ ,and $C_{rng}$ )

$$L_{HLS} = \frac{C_{high} + C_{low}}{2} \quad (32)$$

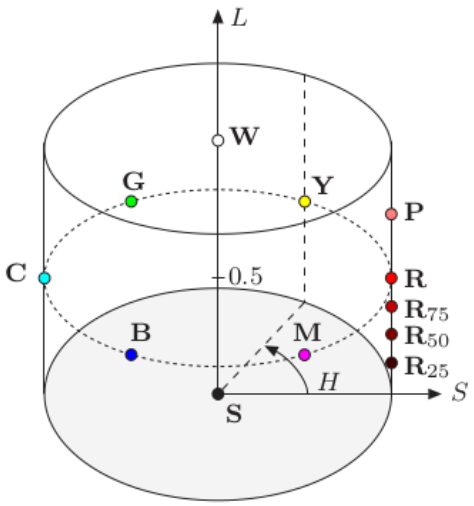
$$S_{HLS}=\begin{cases}0 & for\ L_{HLS}=0\\ 0.5.\dfrac{C_{rng}}{L_{HLS}} & for\ 0<L_{HLS}\leqslant 0.5\\ 0.5.\dfrac{C_{rng}}{1}-L_{HLS} & for\ 0.5<L_{HLS}<1\\ 0 & for\ L_{HLS}=1\end{cases} \quad (33)$$



**H HLS** **S HLS** **L HLS**

**Figure 34** HLS color components H HLS (hue), S HLS (saturation), and L HLS (luminance).

The individual HLS components of the test image as grayscale images are showing in (Figure 34) . The unit cube in the RGB space is mapped to a cylinder with height and length 1 (Figure 35). In contrast to the HSV space (Figure35 ), the primary colors lie together in the horizontal plane at L HLS = 0.5 and the white point lies outside of this plane at L HLS = 1.0. Using these nonlinear transformations, the black and the white points are mapped to the top and the bottom planes of the cylinder, respectively.




RGB/HLS Values

| Pt. | Color | R | G | B | H | S | L |
|---|---|---|---|---|---|---|---|
| S | Black | 0.00 | 0.00 | 0.00 | — | 0.00 | 0.00 |
| R | Red | 1.00 | 0.00 | 0.00 | 0 | 1.00 | 0.50 |
| Y | Yellow | 1.00 | 1.00 | 0.00 | 1/6 | 1.00 | 0.50 |
| G | Green | 0.00 | 1.00 | 0.00 | 2/6 | 1.00 | 0.50 |
| C | Cyan | 0.00 | 1.00 | 1.00 | 3/6 | 1.00 | 0.50 |
| B | Blue | 0.00 | 0.00 | 1.00 | 4/6 | 1.00 | 0.50 |
| M | Magenta | 1.00 | 0.00 | 1.00 | 5/6 | 1.00 | 0.50 |
| W | White | 1.00 | 1.00 | 1.00 | — | 0.00 | 1.00 |
| $R_{75}$ | 75% Red | 0.75 | 0.00 | 0.00 | 0 | 1.00 | 0.375 |
| $R_{50}$ | 50% Red | 0.50 | 0.00 | 0.00 | 0 | 1.00 | 0.250 |
| $R_{25}$ | 25% Red | 0.25 | 0.00 | 0.00 | 0 | 1.00 | 0.125 |
| P | Pink | 1.00 | 0.50 | 0.50 | 0/6 | 1.00 | 0.75 |

**Figure 35** HLS color space.

### II.4 TV Color Spaces YUV [15]

YUV is the basis for the color encoding used in analog television in both the North American NTSC and the European PAL systems. The luminance component Y is computed, just as in Eqn. from the RGB components as

$$Y = 0.299 . R + 0.587 . G + 0.114 . B \quad (34)$$

under the assumption that the RGB values have already been gamma corrected according to the TV encoding standard (γ NTSC = 2.2 and γ PAL = 2.8, seeSec. 4.7) for playback.

The UV components are computed from a weighted difference between the luminance and the blue or red components as

$$U = 0.492 . (B - Y) \quad \text{and} \quad V = 0.877 . (R - Y) \quad (35)$$

**Y** **U** **V**

**Figure 36** YUV color components

### II.5 Colorimetric Color Spaces

To make colors appear similar or even identical on different media modalities, we need a representation that is independent of how a particular device reproduces these colors. Color systems that describe colors in a measurable,device-independent fashion are called colorimetric or calibrated

#### II.5.1 CIE Color Spaces [16]

The XYZ color system, developed by the CIE (Commission Internationale d'Èclairage) in the 1920s and standardized in 1931, is the foundation of most colorimetric color systems that are in use today

#### II.5.1.1 CIE XYZ color space

The CIE XYZ color scheme was developed after extensive measurements of human visual perception under controlled conditions. It is based on three imaginary primary colors X, Y , Z, which are chosen such that all visible colors can be described as a summation of positive-only

components, where the Y component corresponds to the perceived lightness or luminosity of a color. All visible colors lie inside a three-dimensional cone-shaped region (Figure 37) which interestingly enough does not include the primary colors themselves.

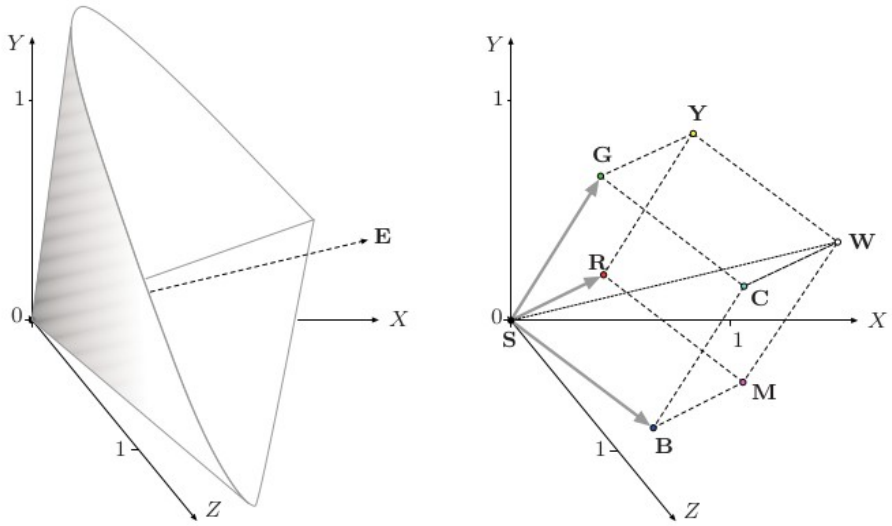



**Figure 37** CIE XYZ color space.

Some common color spaces, and the RGB color space in particular, conveniently relate to XYZ space by a linear coordinate transformation, as shown in Figure 37 (b), the RGB color space is embedded in the XYZ space as a distorted cube, and therefore straight lines in RGB space map to straight lines in XYZ again. The CIE XYZ scheme is (similar to the RGB color space) nonlinear with respect to human visual perception, that it a particular fixed distance in XYZ is not perceived as a uniform color change throughout the entire color space.

The XYZ coordinates of the RGB color cube (based on the primary colors defined by ITU-R BT.709) are listed in Table 2

**Table 2** Coordinates of the RGB color cube in CIE XYZ space. The X, Y, Z values refer to standard (ITU-R BT.709) primaries and white point D65 , x, y denote the corresponding CIE chromaticity coordinates.

| Pt. | Color | $R$ | $G$ | $B$ | $X$ | $Y$ | $Z$ | $x$ | $y$ |
|---|---|---|---|---|---|---|---|---|---|
| **S** | black | 0.00 | 0.00 | 0.00 | 0.0000 | 0.0000 | 0.0000 | 0.3127 | 0.3290 |
| **R** | red | 1.00 | 0.00 | 0.00 | 0.4125 | 0.2127 | 0.0193 | 0.6400 | 0.3300 |
| **Y** | yellow | 1.00 | 1.00 | 0.00 | 0.7700 | 0.9278 | 0.1385 | 0.4193 | 0.5052 |
| **G** | green | 0.00 | 1.00 | 0.00 | 0.3576 | 0.7152 | 0.1192 | 0.3000 | 0.6000 |
| **C** | cyan | 0.00 | 1.00 | 1.00 | 0.5380 | 0.7873 | 1.0694 | 0.2247 | 0.3288 |
| **B** | blue | 0.00 | 0.00 | 1.00 | 0.1804 | 0.0722 | 0.9502 | 0.1500 | 0.0600 |
| **M** | magenta | 1.00 | 0.00 | 1.00 | 0.5929 | 0.2848 | 0.9696 | 0.3209 | 0.1542 |
| **W** | white | 1.00 | 1.00 | 1.00 | 0.9505 | 1.0000 | 1.0888 | 0.3127 | 0.3290 |

## II.6 sRGB

was developed (jointly by Hewlett-Packard and Microsoft) with the goal of creating a precisely specified color space for display-oriented applications, such as computer graphics or multimedia , based on standardized mappings with respect to the colorimetric CIE XYZ color space. This includes precise specifications of the three primary colors, the white reference point, ambient lighting conditions, and gamma values. Interestingly, the sRGB color specification is the same as the one specified many years before for the European PAL/SECAM television standards.

sRGB exhibits a relatively small gamut which, however, includes most colors that can be reproduced by current computer and video monitors.

Although sRGB was not designed as a universal color space, its CIE-based specification at least permits more or less exact conversions to and from other color spaces.

**Linear vs. nonlinear color components**

sRGB is a nonlinear color space with respect to the XYZ coordinate system, and it is important to carefully distinguish between the linear and nonlinear RGB component values. The nonlinear values (denoted R',G' ,B') represent the actual color tuples, the data values read from an image file or received from a digital camera. These values are precorrected with a fixed Gamma (≈ 2.2) such that they can be easily viewed on a common color monitor without any additional conversion.

The corresponding linear components (denoted R, G, B) relate to the CIE XYZ color space by a linear mapping and can thus be computed from X, Y, Z coordinates and vice versa by simple matrix multiplication,

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = M_{RGB} \cdot \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = M^{-1}{}_{RGB} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix} \tag{36}$$

respectively, with

$$M_{RGB} = \begin{pmatrix} 3.240479 & -1.537150 & -0.498535 \\ -0.969256 & 1.875992 & 0.041556 \\ 0.055648 & -0.204043 & 1.057311 \end{pmatrix} \tag{37}$$

$$M^{-1}{}_{RGB} = \begin{pmatrix} 0.412453 & 0.357580 & 0.180423 \\ 0.212671 & 0.715160 & 0.072169 \\ 0.019334 & 0.119193 & 0.950227 \end{pmatrix} \tag{38}$$

### II.6.1 Transformation CIE XYZ→sRGB

To transform a given XYZ color to sRGB (Figure38 ), we first compute the linear R, G, B values by multiplying the (X, Y, Z) coordinate vector with the matrix $M_{RGB}$

$$\begin{pmatrix} R \\ G \\ B \end{pmatrix} = M_{RGB} \cdot \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \quad (39)$$

Subsequently, a modified gamma correction with γ = 2.4 (which corresponds to an effective gamma value of ca. 2.2) is applied to the linear R, G, B values,

$$R' = f_\gamma(R), G' = f_\gamma(G), B' = f_\gamma(B) \quad (40)$$

with

$$f_\gamma(c) = \begin{cases} 1.055 \cdot C^{\frac{1}{2.4}} - 0.055 & for\ c > 0.0031308 \\ 12.92 \cdot c & for\ \ c \leqslant 0.0031308 \end{cases} \quad (41)$$

The resulting nonlinear sRGB components R' , G' , B' are limited to the interval [0, 1]. To obtain discrete numbers, the R ' , G' , B' values are finally scaled linearly to the 8-bit integer range [0, 255].

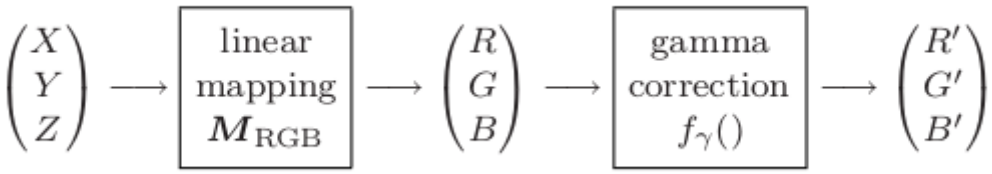



**Figure 38** Color transformation from CIE XYZ to sRGB

### II.6.2 Transformation sRGB→CIE XYZ

To compute the reverse transformation from sRGB to XYZ, the given (nonlinear) R ' , G' , B' values (in the range [0, 1]) are first linearized by inverting the gamma correction

$$R = f^{-1_\gamma}(R'), \quad G = f^{-1_\gamma}(G'), \quad B = f^{-1_\gamma}(B') \quad (42)$$

$$with \quad \left(f^{-1}\right)_\gamma(c') = \begin{cases} \left(\frac{c'+0.055}{1.055}\right)^{2.4} & for\ c' > 0.03928 \\ \frac{c'}{12.92} & for\ \ c' \leqslant 0.03928 \end{cases} \quad (43)$$

Subsequently, the linearized (R, G, B) vector is transformed to XYZ coordinates by multiplication with the inverse of the matrix $M_{RGB}$

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = M^{-1}{}_{RGB} \cdot \begin{pmatrix} R \\ G \\ B \end{pmatrix} \quad (44)$$

Table 3 lists the nonlinear and the linear RGB component values for selected color points. Note that component values of 0 and 1 are not affected by the gamma correction because these values map to themselves. The coordinates of the extremal points of the RGB color cube are therefore identical in nonlinear and linear RGB spaces. However, intermediate values are strongly affected by the gamma correction, as illustrated by the coordinates for the color points K . . . P, which emphasizes the importance of differentiating between linear and nonlinear color coordinates.

### II.6.3 Calculating with sRGB values

Due to the wide use of sRGB in digital photography, graphics, multimedia, Internet imaging, etc., there is a probability that a given image is encoded in sRGB colors. If, for example, a JPEG image is opened with ImageJ or Java, the pixel values in the resulting data array are media-oriented (i. e., nonlinear R ' , G' , B' components of the sRGB color space). Unfortunately, this fact is often overlooked by programmers, with the consequence that colors are incorrectly manipulated and reproduced.

As a general rule, any arithmetic operation on color values should always be performed on the linearized R, G, B components, which are obtained from the nonlinear R ' , G' , B' values through the inverse gamma function $f_\gamma{}^{-1}$ and converted back again with $f_\gamma$

**Table 3** CIE XYZ coordinates for selected sRGB colors.

| | | sRGB *nonlinear* | | | sRGB *linearized* | | | CIE XYZ | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Pt. | Color | $R'$ | $G'$ | $B'$ | $R$ | $G$ | $B$ | $X_{65}$ | $Y_{65}$ | $Z_{65}$ |
| **S** | black | 0.00 | 0.0 | 0.0 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 | 0.0000 |
| **R** | red | 1.00 | 0.0 | 0.0 | 1.0000 | 0.0000 | 0.0000 | 0.4125 | 0.2127 | 0.0193 |
| **Y** | yellow | 1.00 | 1.0 | 0.0 | 1.0000 | 1.0000 | 0.0000 | 0.7700 | 0.9278 | 0.1385 |
| **G** | green | 0.00 | 1.0 | 0.0 | 0.0000 | 1.0000 | 0.0000 | 0.3576 | 0.7152 | 0.1192 |
| **C** | cyan | 0.00 | 1.0 | 1.0 | 0.0000 | 1.0000 | 1.0000 | 0.5380 | 0.7873 | 1.0694 |
| **B** | blue | 0.00 | 0.0 | 1.0 | 0.0000 | 0.0000 | 1.0000 | 0.1804 | 0.0722 | 0.9502 |
| **M** | magenta | 1.00 | 0.0 | 1.0 | 1.0000 | 0.0000 | 1.0000 | 0.5929 | 0.2848 | 0.9696 |
| **W** | white | 1.00 | 1.0 | 1.0 | 1.0000 | 1.0000 | 1.0000 | 0.9505 | 1.0000 | 1.0888 |
| **K** | 50% gray | 0.50 | 0.5 | 0.5 | 0.2140 | 0.2140 | 0.2140 | 0.2034 | 0.2140 | 0.2330 |
| $\mathbf{R}_{75}$ | 75% red | 0.75 | 0.0 | 0.0 | 0.5225 | 0.0000 | 0.0000 | 0.2155 | 0.1111 | 0.0101 |
| $\mathbf{R}_{50}$ | 50% red | 0.50 | 0.0 | 0.0 | 0.2140 | 0.0000 | 0.0000 | 0.0883 | 0.0455 | 0.0041 |
| $\mathbf{R}_{25}$ | 25% red | 0.25 | 0.0 | 0.0 | 0.0509 | 0.0000 | 0.0000 | 0.0210 | 0.0108 | 0.0010 |
| **P** | pink | 1.00 | 0.5 | 0.5 | 1.0000 | 0.2140 | 0.2140 | 0.5276 | 0.3812 | 0.2482 |

### II.7 CIE LUV (CIE L ∗ u ∗ v ∗)color space [16]

commonly known by its abbreviation CIELUV, is a color space adopted by the International Commission on Illumination (CIE) in 1976, as a simple-to-compute transformation of the 1931 CIE XYZ color space, but which attempted perceptual uniformity. It is extensively used for applications such as computer graphics which deal with colored lights. Although additive mixtures of different colored lights will fall on a line in CIELUV's uniform chromaticity diagram (dubbed the CIE 1976 UCS), such additive mixtures will not, contrary to popular belief, fall along a line in the CIELUV color space unless the mixtures are constant in lightness.

The non-linear relations for $L^*$, $u^*$, and $v^*$ are given below:

$$L^* = \begin{cases} \left(\frac{29}{3}\right)^3 Y/Y_n, & Y/Y_n \leqslant \left(\frac{6}{29}\right)^3 \\ 116\left(Y/Y_n\right)^{\frac{1}{3}} - 16 & Y/Y_n > \left(\frac{6}{29}\right)^3 \end{cases} \quad (45)$$

$$u^* = 13L^*.\left(u' - u'_n\right) \quad (46)$$

$$v^* = 13L^*.\left(v' - v'_n\right) \quad (47)$$

The quantities $u'_n$ and $v'_n$ are the ($u'$, $v'$) chromaticity coordinates of a "specified white object" – which may be termed the white point – and $Y_n$ is its luminance. In reflection mode, this is often (but not always) taken as the ($u'$, $v'$) of the perfect reflecting diffuser under that illuminant. (For example, for the 2° observer and standard illuminant C, $u'_n$ = 0.2009, $v'_n$ = 0.4610.) Equations for $u'$ and $v'$ are given below

$$u' = \frac{4X}{X + 15Y + 3Z} = \frac{4x}{-2x + 12Y + 3} \quad (48)$$

$$v' = \frac{9Y}{X + 15Y + 3Z} = \frac{9y}{-2x + 12Y + 3} \quad (49)$$

For typical images, $u^*$ and $v^*$ range ±100. By definition, $0 \leq L^* \leq 100$.
to convert rgb to CIELUV first we convert RGB to CIEXYZ and then we convert CIEXYZ to CIELUV

### II.8 The 1976 CIE L ∗ a ∗ b ∗ Color Space [16]

In 1976, the CIE convention recommended the CIE L ∗ a ∗ b ∗ , or CIELAB, color space, mainly for use in the plastic, textile, and paint industries. As in the Hunter space, the luminance is represented along the z axis in a Cartesian system of coordinates, with values from zero for black to 100 for a perfectly white body [constant reflectance R(λ) = 1]. The positive a ∗ axis represents

the amount of purplish red, while the negative a ∗ axis represents the amount of green. The positive b ∗ axis represents the amount of yellow and the negative b ∗ axis represents the amount of blue. The maximum possible magnitude of the values on these axes is a function of the luminance, between ±100 and ±200 for a ∗ and b ∗ , respectively.

The transformation equations used for passing from the CIE x, y, z system to the CIE L ∗ a ∗ b ∗ system are

$$L^* = 116 f\left(\frac{Y}{Y_n}\right) - 16 \quad (50)$$

$$a^* = 500\left(f\left(\frac{X}{X_n}\right) - f\left(\frac{Y}{Y_n}\right)\right) \quad (51)$$

$$b^* = 200\left(f\left(\frac{Y}{Y_n}\right) - f\left(\frac{Z}{Z_n}\right)\right) \quad (52)$$

where

$$f(t) = \begin{cases} \sqrt[3]{t} & if\ t > \delta^3 \\ \frac{t}{3\delta^2} + \frac{4}{29} & otherwise\ \delta = \frac{6}{29} \end{cases} \quad (53)$$

Here, $X_n$, $Y_n$ and $Z_n$ are the CIE XYZ tristimulus values of the reference white point (the subscript n suggests "normalized") , Under Illuminant D65 with normalization Y = 100, the values are

$$X_n = 95.0489\,;\, Y_n = 100\,;\, Z_n = 108.8840 \quad (54)$$

Search in color space style active and new color space are invented every decade to enhance performance in image processing algorithms like gamut mapping, lossy image compression, image enhancement, image segmentation, image denoising etc , or to minimum computational cost for real time or quasi-real time processing , in 2017 a new uniform color space named $J_z A_z B_z$ [17] has been developed , it is proposed for color and imaging applications that include wide color gamut and high dynamic range and it is perceptually uniform over a wide gamut, linear in iso-hue directions, and can predict both small and large color differences as well as lightness in high dynamic range environments.

# CHAPTER III
# EXPERIMENTATION AND RESULTS

The main focus throughout this thesis has been to show if color space has impact over the performance of deep learning model in image classification .
It begins by describing the characteristics of the CIFAR10 dataset , and the process of conversion form the original dataset to other colors space , flowed by a short description of the Framework keras[6] and Tensorflow , and a brief description of the implementation , followed by a section containing the experimental results and performance evaluation , Finally, a short discussion of the results .

## III.1 CONVERTING CIFAR -10 DATASET TO DIFFERENT COLORSPACE

The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.
The dataset is divided into five training batches and one test batch, each with 10000 images. The test batch contains exactly 1000 randomly-selected images from each class.
The training batches contain the remaining images in random order, but some training batches may contain more images from one class than another. Between them, the training batches contain exactly 5000 images from each class, it is stored in a files that contain a Python ”pickled” object produced with cPickle .

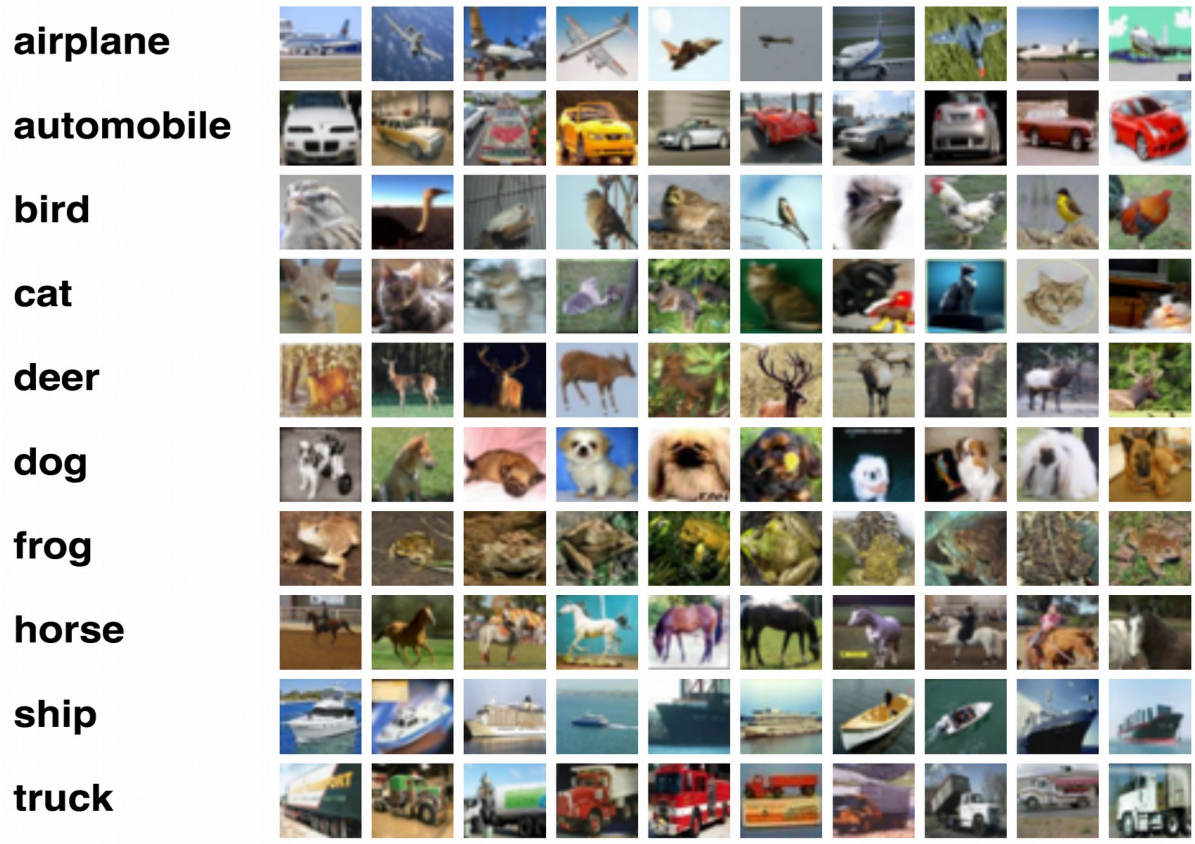



**Figure 39** The CIFAR-10 dataset.

each of the batch files contains a dictionary with the following elements:
data – a 10000x3072 numpy array of uint8s. Each row of the array stores a 32x32 colour image. The first 1024 entries contain the red channel values, the next 1024 the green, and the final 1024 the blue.
labels – a list of 10000 numbers in the range 0-9. The number at index i indicates the label of the ith image in the array data. The number of columns(10000)indicates the number of sample data.
As stated in the CIFAR-10 [2] dataset, the row vector, (3072) represents one color image of 32x32 pixels as showing in (Figure 40)

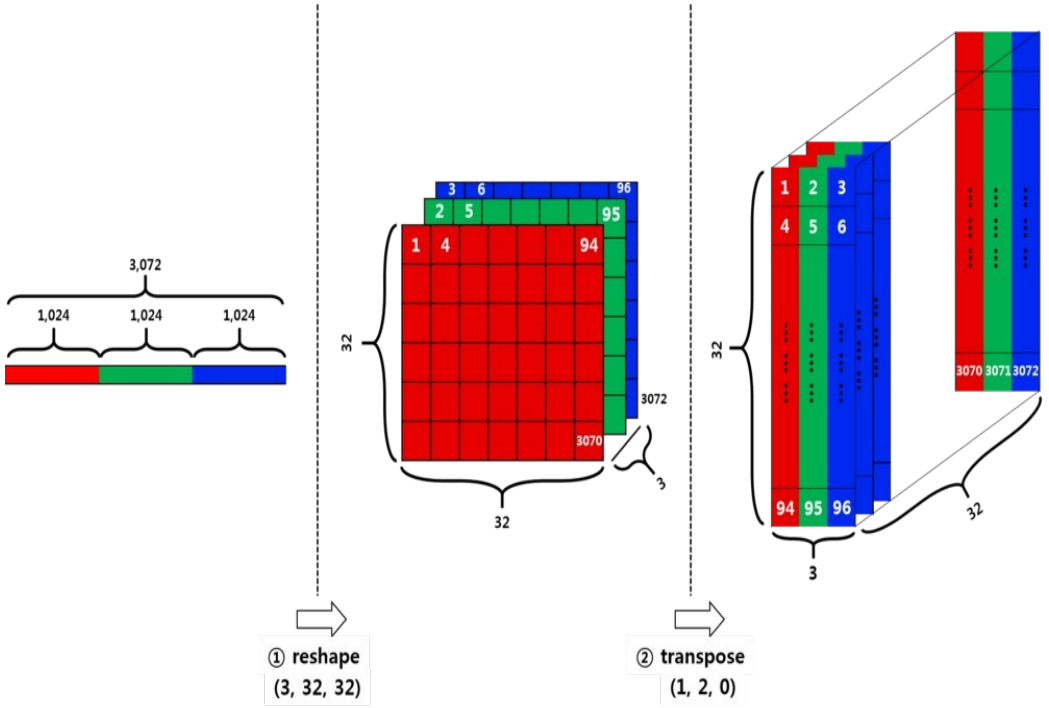



**Figure 40** reshape and transpose

we have converted the dataset to the choosing color space Figure41 by first loading the dataset from the batch file to numpy array using the function load_data() from the module keras.datasets.cifar10 , then converting it to float32 numpy array to reduce loss of accuracy due to the use of unsigned byte and normalised by divide by 255 to get value in the intervail of [0, 1] to avoid exploding/vanishing gradient problems due to the input passed to the activation functions , CIFAR have images present in the sRGB format so there is NO need to do linearization by inverting the gamma correction because the central color space in Scikit image library is sRGB.

**Figure 41** Visualization of image from CIFAR10 data batch 1 index 7 in LAB ,YUV, LUV, HSV,RGB and HSL colourspaces respectively (from left)

and finally for HSV , YUV , LUV , LAB use the appropriate function from scikit-image[18] to convert each image in the dataset , and for HSL use colorsys[19] module from Python Standard Library , then we store the dataset in pickle file to be loaded during the training , the process of converting the dataset to other color space is shown in the below Figure42

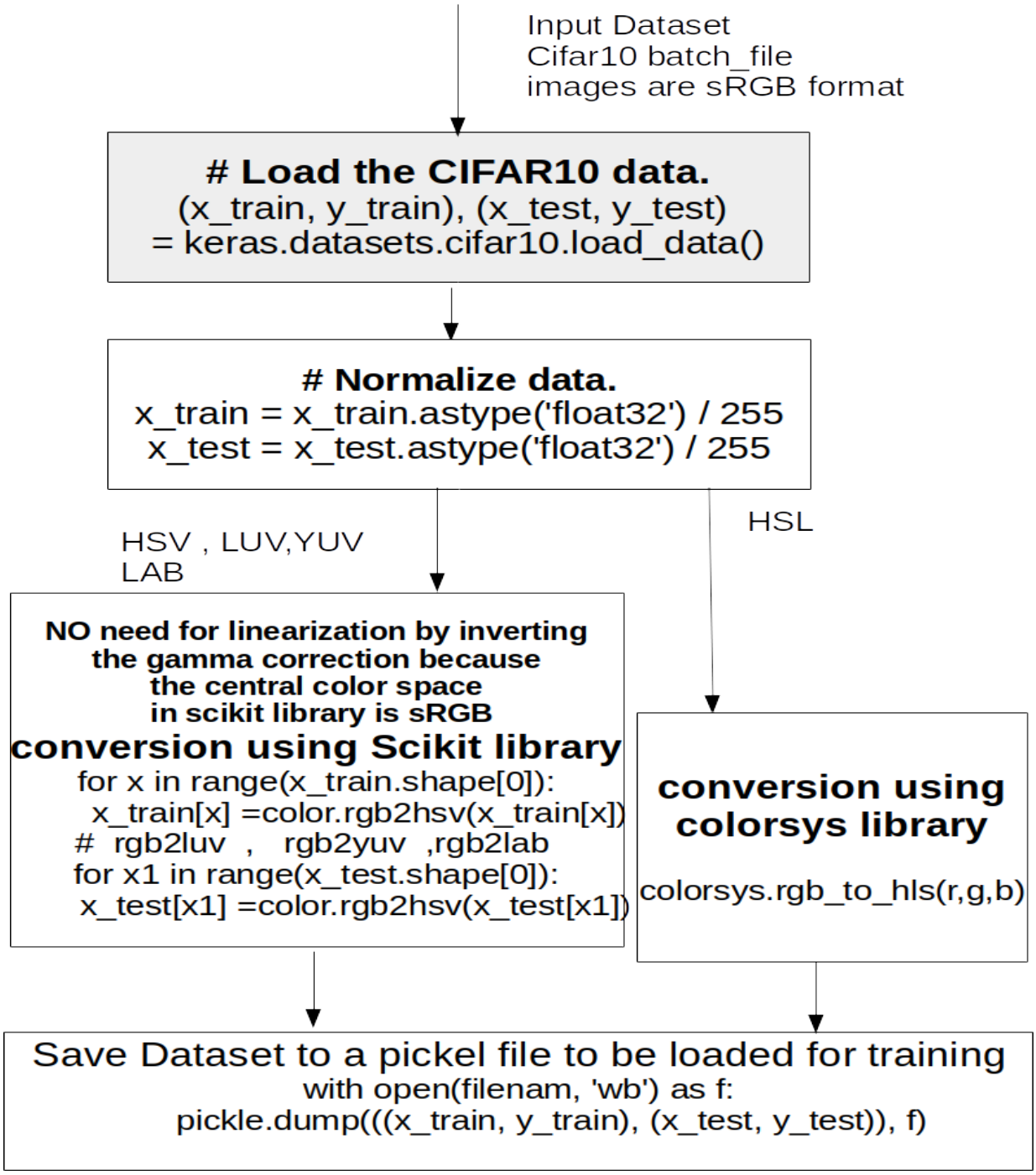



**Figure 42** the process of converting the dataset to another color space

### III.2 Libraries

In order to simplify the process of implementing deep learning model several software libraries have been developed. These provide tools that automatically execute many of the tasks required to set up a functional neural network. Though the libraries are based on the same theoretical , they differ in their approach on how to implement them. Below are short summaries of keras and Tensorflow

#### III.2.1 Keras

Keras [6] is a high-level wrapper which runs on top of Theano Additionally, it is also able to run on top of tensorflow. It is designed to minimise overhead, to allow for fast and easy prototyping of deep learning model .

#### III.2.2 Tensorflow

Originally developed by Google as part of the Google Brain project, Tensorflow was made open source in late 2015. Tensorflow [20]is a Python library and stands apart by being the only one of the major libraries developed from the ground up by a major corporation, while the others have their origin in the research community.

Tensorflow has some integrated quality of life tools such as TensorBoard, which allows the user to easily produce graphs visualizing things such as learning rate, model weights, loss functions and more. Tensorflow is also the only library that can distribute the workload not just across GPUs on the same device but on several connected devices, which can be a major computational advantage.

#### III.2.3 scikit-image

scikit-image[18] is an open-source image processing library for the Python programming language, It includes a collection of algorithms for image processing, segmentation, geometric transformations, color space manipulation, analysis, filtering, morphology, feature detection, and more.

### III.3 Implementation

For the implementation, the script CIFAR10_CNN_Capsule[21] and CIFAR10_ResNet[22] from Keras was used as a base for this project. It was modified to be able to convert the dataset to the five colors space HSV , LUV , LAB , HSL ,YUV using scikit-image [18] and colorsys [19]module from Python Standard Library, and additional functionality relevant to this project was implemented, including the save and load of the training model the record of the history of training to be able to compare the performance of each model and the design of the graphic user interface to load the training model , display the dataset and predict there class Figure43, also accuracy evaluation by class was implemented .

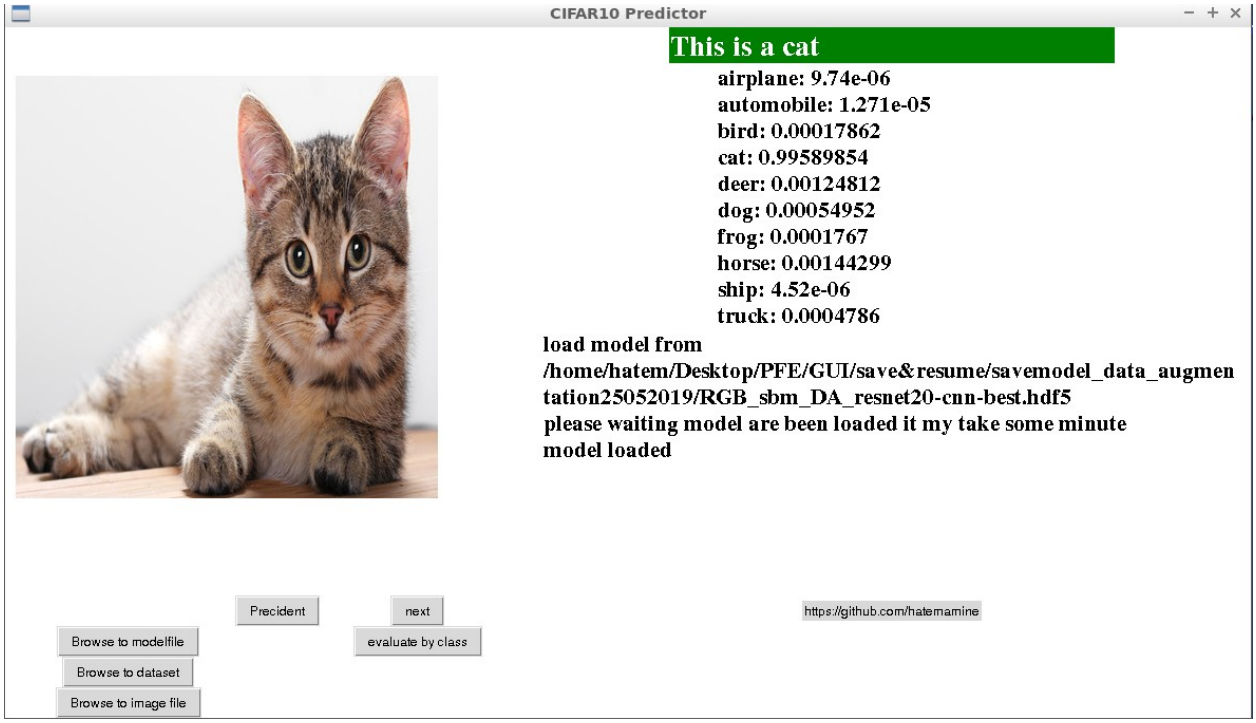



**Figure 43** GUI Cifar10 Predictor

## III.4 EXPERIMENTS

After coverting the dataset to diffrent coulorspace HSL ,HSV ,LUV, YUV , LAB . We have trained the modes in two different architecture model ResNet20 and CapsulNet with fixed hyper-parameters we set batch size equal to 128 (the number of training examples in one forward/backward pass) and the number of epochs to 50 ,even if resnet20 can give an accuracy of 91.25 with 200 epochs , we choice 50 epoch because of the limited computation resources and because we went just to understand the effect of colourspace on the performance of the CNN models, not to get the maximum performance of the model .

All the models were trained using CPU intel core I5-6200u with 4GB Ram with keras [9] as backend. ResNet20 is just 0.27M learnable parameters It took around 4 hours to train each model without data augmentation and 6 hours to train each model with data augmentation in ResNet20.

For CapsulNet it took 3 hours to train each model with data augmentation We store the weights of the CNN model for every epochs while training and record the accuracy and loss for every epoch to be able to evaluate and compare the results for each model .

### III.5 Discussion RESULTS

from the experiments that we have conducted the results shown in table 4 was obtained, we observed that there is a minor change in accuracy in ResNet20 and also in CapsulNet , but even a small percentage may make the difference, an the other hand LUV is good alternative it show improvement about 0.92% comparison to RGB in ResNet20 ,and 0.39% in CapsuNet.

**Table 4** comparison of results for different color spaces on cifar-10 with resnet20 and capsulnet

| | Validation Accuracy | | |
|---|---|---|---|
| | ResNet20 WITHOUT DATA AUGMENTATION | ResNet20 WITH DATA AUGMENTATION | CapsulNet WITH DATA AUGMENTATION |
| RGB | 74.76% | 85.02% | 83.15% |
| HSL | 73.48% | 83.73% | 81.65% |
| HSV | 75.12% | 84.81% | 82.56% |
| LUV | 75.59% | **85.94%** | **83.54%** |
| YUV | 75.96% | 85.88% | 83.31% |
| LAB | **76.52%** | 85.66% | 82.94% |

The performance of the models can be further understood by looking at the per class recognition accuracy of the CNN model ResNet20 and CapsulNet on test set of different colourspace which is shown in Table 5 , 6 and 7 we can observe that some color space are well situated for some class in some model but it not persistent as we change from ResNet20 to CapsuleNet or form using data augmentation the highest accuracy change from color space to another taking for example the class ship in ResNet20 without data augmentation the highest accuracy 91.10% is obtained using LAB colorspace , for ResNet20 with data augmentation the highest accuracy 92.50% is obtained using YUV colorspace , and for CapsulNet with data augmentation the highest accuracy 91.50% is obtained using RGB colorspace .

**Table 5** per class accuracy of resnet 20 in different color space ( without data augmentation )

| | Colourspace | | | | | |
|---|---|---|---|---|---|---|
| Class Name | RGB | HSL | HSV | LUV | YUV | LAB |
| airplane | 78.70% | 72.20% | **80.90%** | 77.30% | 75.40% | 76.60% |
| automobile | **90.90%** | 84.00% | 85.50% | 86.90% | 78.60% | 90.00% |
| bird | 65.40% | 63.30% | 49.80% | **82.10%** | 74.00% | 70.20% |
| cat | 36.80% | **69.90%** | 64.70% | 47.20% | 54.30% | 57.10% |
| deer | 66.20% | 74.80% | 77.00% | 78.20% | **81.30%** | 67.60% |
| dog | **79.40%** | 59.80% | 68.30% | 67.40% | 66.70% | 63.60% |
| frog | 83.20% | 74.00% | **86.60%** | 73.90% | 83.50% | 84.20% |
| horse | 72.30% | 66.40% | 68.40% | **81.30%** | 74.80% | 74.60% |
| ship | 89.00% | 83.60% | 86.20% | 87.90% | 90.50% | **91.10%** |
| truck | 85.70% | 86.80% | 83.80% | 73.70% | 80.50% | **90.20%** |

**Table 6** per class accuracy of resnet 20 in different colorspace ( with data augmentation )

| | Colourspace | | | | | |
|---|---|---|---|---|---|---|
| Class Name | RGB | HSL | HSV | LUV | YUV | LAB |
| airplane | 83.10% | **92.70%** | 91.60% | 89.40% | 89.40% | 87.50% |
| automobile | 95.00% | 88.90% | 96.20% | 87.20% | **97.80%** | 78.50% |
| bird | **83.60%** | 78.30% | 71.00% | 73.30% | 80.20% | 81.30% |
| cat | 65.10% | **78.40%** | 67.90% | 75.60% | 77.80% | 76.70% |
| deer | 91.00% | 83.70% | 81.20% | **93.70%** | 84.90% | 90.70% |
| dog | 74.00% | 66.80% | 79.20% | 78.50% | 65.40% | **82.90%** |
| frog | 90.80% | 88.40% | 89.00% | 88.50% | 89.50% | **91.20%** |
| horse | 87.20% | 89.60% | 93.80% | 88.10% | **94.80%** | 79.50% |
| ship | 84.90% | 78.70% | 85.10% | 90.90% | **92.50%** | 91.30% |
| truck | 95.50% | 91.80% | 93.10% | 94.20% | 86.50% | **97.00%** |

**Table 7** per class accuracy of capsulenet in different colorspace ( with data augmentation )

| | Colourspace | | | | | |
|---|---|---|---|---|---|---|
| Class Name | RGB | HSL | HSV | LUV | YUV | LAB |
| airplane | **86.40%** | 84.80% | 83.30% | 81.00% | 81.90% | **86.40%** |
| automobile | 0.942 | 92.50% | 90.30% | 92.50% | **94.20%** | 89.50% |
| bird | 76.90% | 70.90% | 74.30% | 77.00% | **78.00%** | 70.80% |
| cat | 67.40% | 61.70% | 68.50% | **71.20%** | 64.40% | 68.00% |
| deer | 79.40% | 80.00% | 80.30% | **84.00%** | 81.30% | 82.20% |
| dog | 70.20% | **81.90%** | 71.90% | 75.10% | 77.10% | 71.50% |
| frog | **93.10%** | 88.20% | 89.70% | 90.40% | 90.90% | 90.20% |
| horse | 84.90% | 81.80% | 87.00% | 86.00% | 86.30% | **87.90%** |
| ship | **91.50%** | 84.70% | 90.40% | 88.00% | 91.20% | 89.00% |
| truck | 87.50% | 90.00% | 89.90% | 90.20% | 87.80% | **93.90%** |

**ResNet20 without data augmentation**

looking to the accuracy plot of the model for different colorspace trained with ResNet without data augmentation we can see that all model have the same behavior they all overfitting we can see that by looking at Figure44 and Figure45 we can observe that starting at epoch 10 the training loss still going down but test loss is going up , this is obvious sign of overfitting.

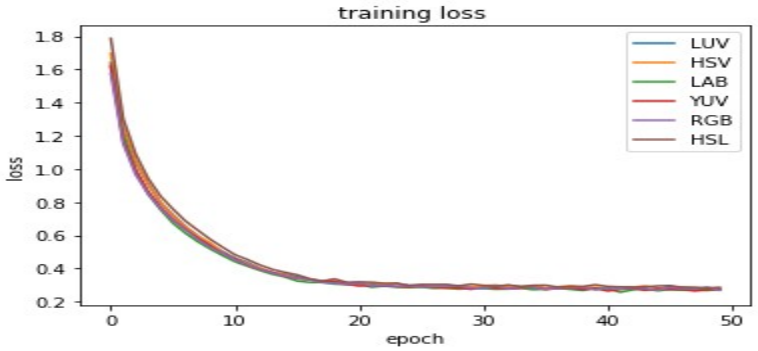



**Figure 44** ResNet20 without data augmentation training loss

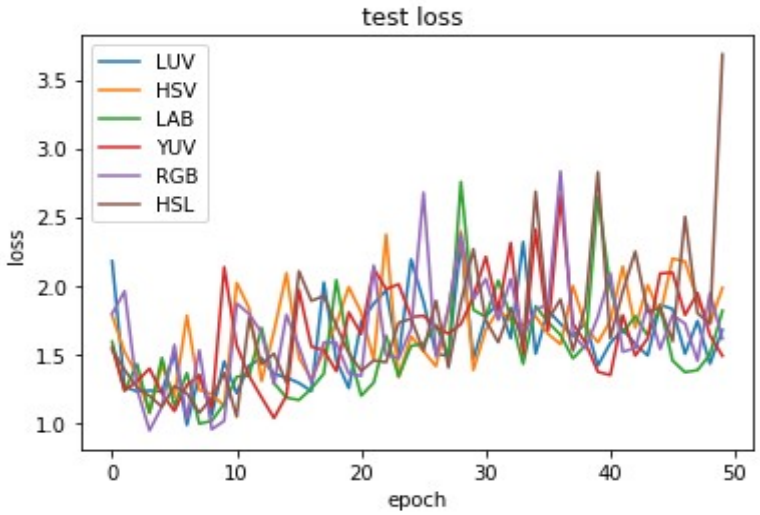



**Figure 45** ResNet20 without data augmentation test loss

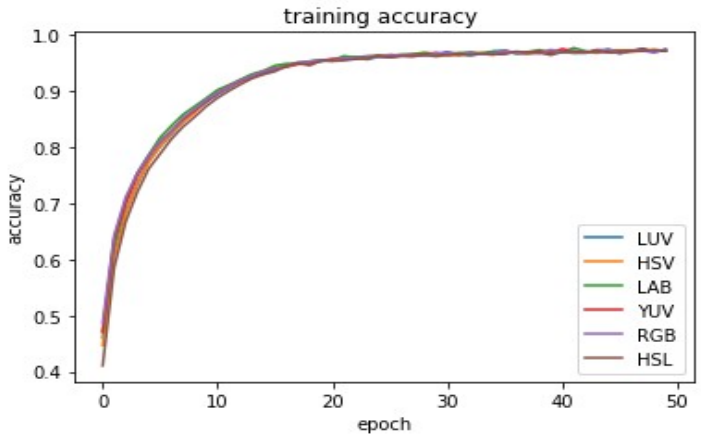



**Figure 46** ResNet20 without data augmentation training accuracy

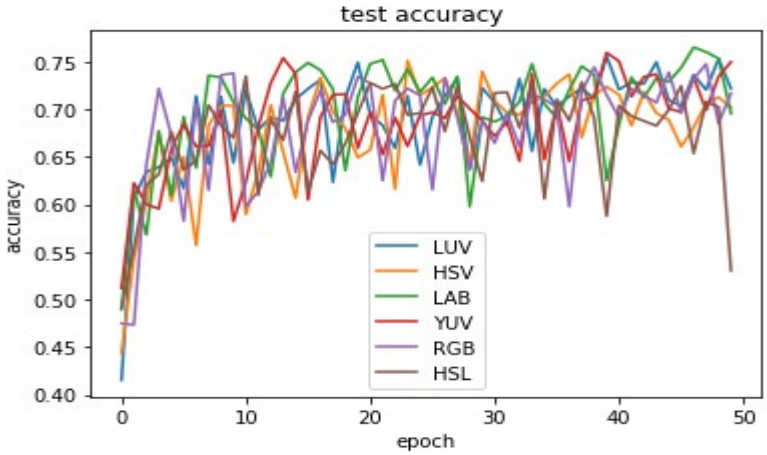



**Figure 47** ResNet20 without data augmentation test accuracy

**ResNet20 with data augmentation** we can observe by looking at Figure48 ,Figure49 , Figure50 and Figure51 that models are less overfitting still have a going up and down in test accuracy it look that the training process is not down in a smooth way in the other hand looking at Figure52 ,Figure53 , Figure54 and Figure55 CapsuleNet with data augmentation is doing well the test accuracy is going up in a smooth way .

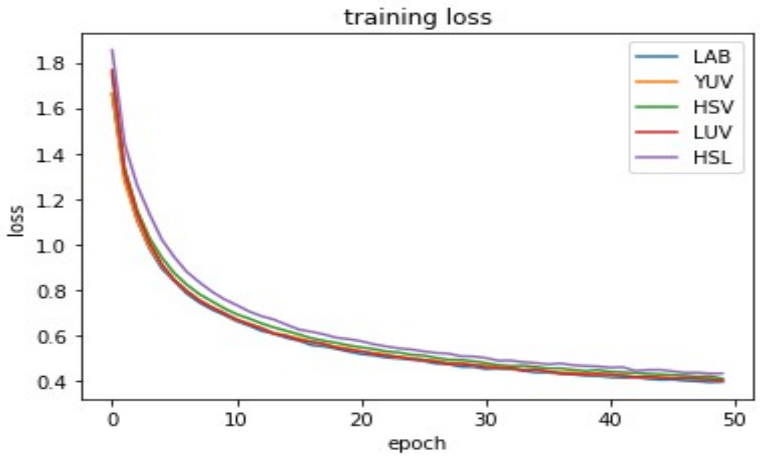



**Figure 48** ResNet20 with data augmentation training loss

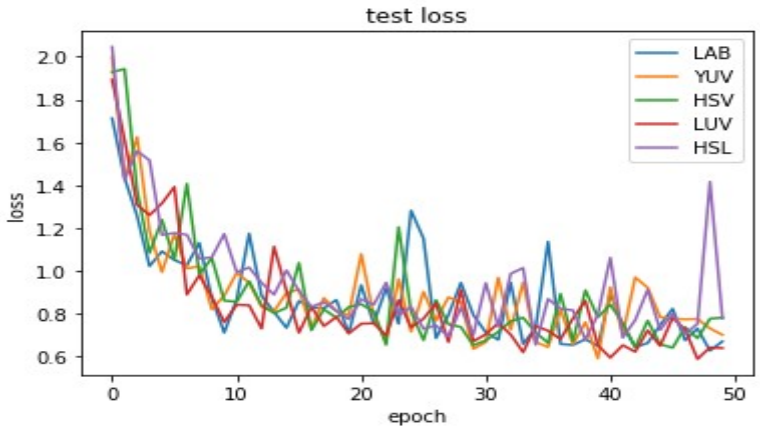



**Figure 49** ResNet20 with data augmentation test loss

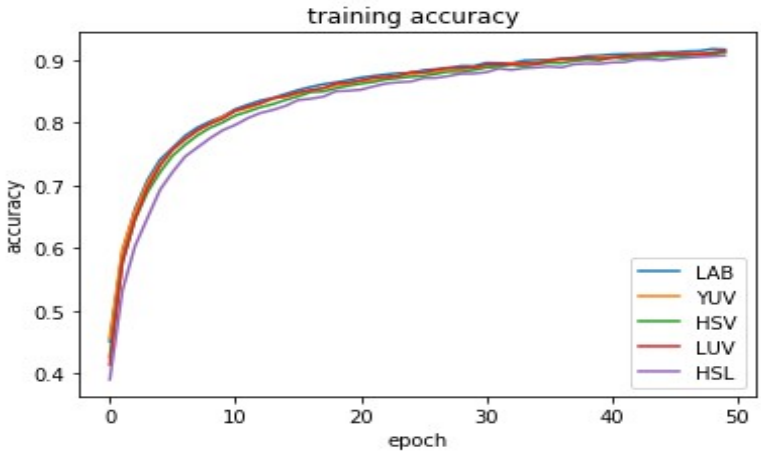



**Figure 50** ResNet20 with data augmentation training accuracy

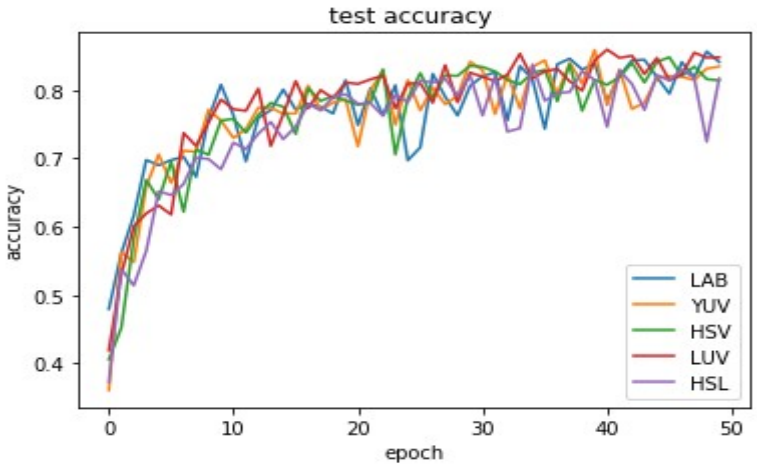



**Figure 51** ResNet20 with data augmentation test accuracy

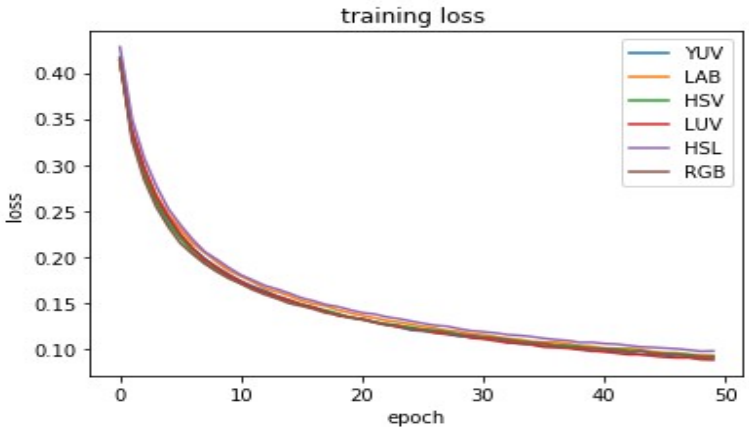



**Figure 52** CapsulNet with data augmentation training loss

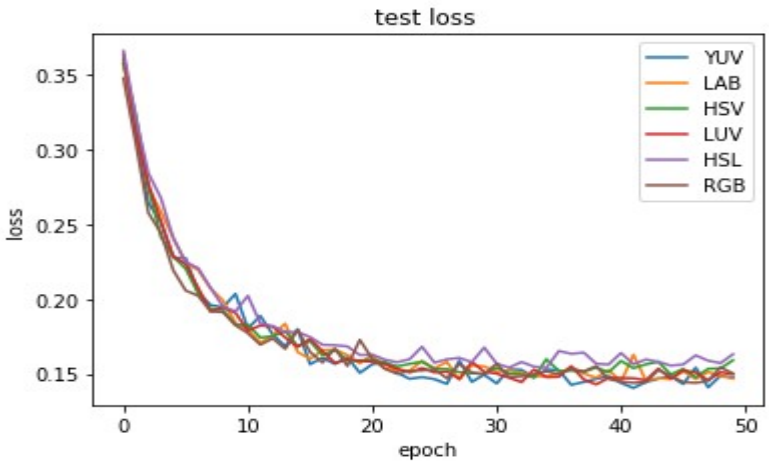



**Figure 53** CapsulNet with data augmentation test loss

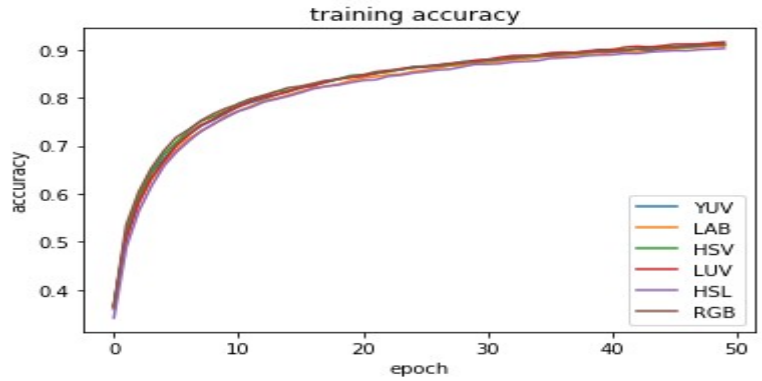



**Figure 54** CapsulNet with data augmentation training accuracy

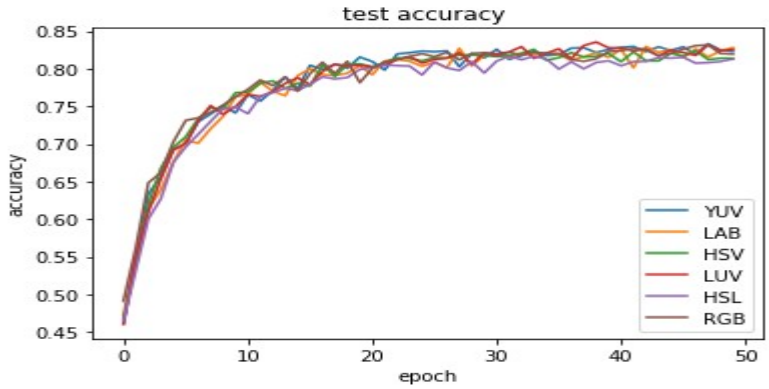



**Figure 55** CapsulNet with data augmentation test accuracy

# CONCLUSION

Converting dataset into different color space and trained each one of them in two different deep learning architecture model ResNet20 and CapsulNet have showing minor change in accuracy in ResNet20 and also in CapsulNet but even a small percentage may make the difference, in the other hand LUV is good alternative it show improvement about 0.92% comparison to RGB in ResNet20 and 0.39% in CapsulNet.

The future scope is instead of making images captured in sRGB colorspace and then convert it to other color space we could capture image directly in the desired color space and use an appropriate structure to store the data to prevent the lose of information in the convert process from one colorspace to another, but by capture image directly in the desired color space and use an appropriate structure to store the data , we may be improve the performance of deep learning model , but we also face the problem of the availability of labled data-set as the majority of data-set are in RGB .

So, we may see in the future a machine that has trained with data-set in RGB have ability to capture and label similar image in different color space in appropriate data structure and then retrained her self to get better model.

# Bibliography


[1] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition,” *arXiv e-prints*, p. arXiv:1512.03385, 2015.

[2] S. Sabour, N. Frosst, and G. E Hinton, “Dynamic Routing Between Capsules,” *arXiv e-prints*, p. arXiv:1710.09829, Oct. 2017.

[3] A. K. V. N. G. Hinton, “The CIFAR-10 dataset {https://www.cs.toronto.edu/~kriz/cifar.html}.” 2014.

[4] K. S. Reddy, U. Singh, and P. K. Uttam, “Effect of image colourspace on performance of convolution neural networks,” in *2017 2nd IEEE International Conference on Recent Trends in Electronics, Information Communication Technology (RTEICT)*, 2017, pp. 2001–2005.

[5] S. N. Gowda and C. Yuan, “ColorNet: Investigating the importance of color spaces for image classification,” *arXiv e-prints*, p. arXiv:1902.00267, 2019.

[6] “Keras: The Python Deep Learning library {https://keras.io}.” 2019.

[7] F. Chollet, *Deep learning with Python*. Shelter Island, NY: Manning Publications Co, 2018.

[8] T. Mitchell, *Machine Learning*. New York: McGraw-Hill, 1997.

[9] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, no. 7553, p. 436, 2015.

[10] N. Buduma, *Fundamentals of deep learning : designing next-generation machine intelligence algorithms*. Sebastopol, CA: O’Reilly Media, 2017.

[11] J. Patterson, *Deep learning : a practitioner’s approach*. Sebastopol, CA: O’Reilly, 2017.

[12] D. E. Rumelhart, G. E. Hinton, R. J. Williams, and others, “Learning representations by back-propagating errors,” *Cogn. Model.*, vol. 5, no. 3, p. 1, 1988.

[13] M. A. Wani, *Advances in deep learning*. Singapore: Springer, 2020.

[14] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[15] W. Burger, *Principles of digital image processing : core algorithms*. London: Springer, 2009.

[16] J. Schanda, *Colorimetry : understanding the CIE system*. Vienna, Austria Hoboken, N.J: CIE/Commission internationale de l’eclairage Wiley-Interscience, 2007.

[17] M. Safdar, G. Cui, Y. Jin Kim, and M. Luo, “Perceptually uniform color space for image signals including high dynamic range and wide gamut,” *Opt. Express*, vol. 25, p. 15131, 2017.

[18] “scikit-image {https://scikit-image.org/}.” 2019.

[19] “colorsys module {https://docs.python.org/2/library/colorsys.html}.” 2019.

[20] Mart\'\in~Abadi *et al.*, "{TensorFlow}: Large-Scale Machine Learning on Heterogeneous Systems." 2015.

[21] "Train a simple CNN-Capsule Network on the CIFAR10 {https://keras.io/examples/cifar10_cnn_capsule/}." 2019.

[22] "Trains a ResNet on the CIFAR10 dataset {https://keras.io/examples/cifar10_resnet}." 2019.