TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
PYTHON

Python is a significant level, translated scripting language created in the late 1980s by Guido van Rossum at the National Research Institute for Mathematics and Computer Science in the Netherlands.

Python 2.0 was released in 2000, and the 2.x renditions were the predominant releases until December 2008. Around then, the improvement group settled on the choice to release rendition 3.0, which contained a couple of generally little however huge changes that were not in reverse perfect with the 2.x forms. [2]

Python 2 and 3 are fundamentally the same as, and a few highlights of Python 3 have been backported to Python 2. In any case, all in all, they remain not exactly perfect.

The name Python, coincidentally, gets not from the snake, however from the British satire troupe Monty Python's Flying Circus, of which Guido was, apparently still is, a fan. It isn't unexpected to discover references to Monty Python representations and films dissipated all through the Python documentation.

1.1 Data types in Python

1.1.1 Python Numbers

Integers, floating-point numbers and complex numbers fall under Python numbers category. They are defined as int, float and complex class in Python.

1.1.2 Python List

The list is an ordered sequence of items. It is one of the most used data types in Python and is very flexible. All the items in a list do not need to be of the same type. Declaring a list is pretty straight forward. Items separated by commas are enclosed within brackets [4].

1.1.3 Python Tuple

The tuple is an arranged succession of things same as a rundown. The main contrast is that tuples are permanent. Tuples once made can't be adjusted. Tuples are utilized to compose ensure information and are normally quicker than list as it can't change progressively. It is

characterized inside enclosures () where things are isolated by commas.

1.1.4 Python Strings

A string is a sequence of Unicode characters. We can use single quotes or double quotes to represent strings. Multi-line strings can be denoted using triple quotes, ''' or """.

1.1.5 Python Set

Set is an unordered collection of unique items. Set is defined by values separated by a comma inside braces { }. Items in a set are not ordered.

1.1.6 Python Dictionary

Dictionary in Python is an unordered assortment of information values, used to store information values like a guide, which not at all like other Data Types that hold just single value as a component, Dictionary holds the key: value pair. Key-value is given in the dictionary to make it more upgraded. Each key-value pair in a Dictionary is isolated by a colon: while each key is isolated by a 'comma'.

1.1.6(a) Creating a Dictionary:

In Python, a Dictionary can be made by putting a succession of components inside wavy {} supports, isolated by 'comma'. Dictionary holds a couple of values, one being the Key and the other comparing pair component being its Key: value. Values in a dictionary can be of any data type and can be copied, though keys can't be rehashed and should be changeless. [2]

Dictionary can likewise be made by the implicit capacity dict(). A vacant dictionary can be made by simply putting to wavy braces {}.

In Python Dictionary, Addition of components should be possible in different manners. Each value, in turn, can be added to a Dictionary by characterizing value alongside the key, for example, Dict[Key] = 'Value'. Refreshing a current value in a Dictionary should be possible by utilizing the implicit update () strategy. Settled key values can likewise be added to a current Dictionary.

1.2 Defining a Function

We can define functions to provide the required functionality. Here are simple rules to define a function in Python.

- Function blocks begin with the keyword def followed by the function name and parentheses (( )).
- Any input parameters or arguments should be placed within these parentheses. We can also define parameters inside these parentheses.
- The first statement of a function can be optional - the documentation string of the function or docstring.
- The code block within every function starts with a colon (:) and is indented.
- The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return none.

1.3 Python Modules

If you quit from the Python translator and enter it once more, the definitions you have made (capacities and factors) are lost. Hence, if you need to compose a to some degree longer program, you are in an ideal situation utilizing a content tool to set up the contribution for the translator and running it with that record as a contribution. This is known as making content.

As your program gets longer, you might need to part it into a few records for simpler support. You may likewise need to utilize a helpful capacity that you've written in a few projects without replicating its definition into each program.

To help this, Python has an approach to place definitions in a document and use them in content or an intuitive example of the mediator. Such a document is known as a module; definitions from a module can be brought into different modules or into the primary module (the assortment of factors that you approach in a content executed at the top level and in adding machine mode).

A module is a document containing Python definitions and explanations. The document name is the module name with the postfix .py added. Inside a module, the module's name

(as a string) is accessible as the value of the worldwide variable __name__.

1.4 Python Classes

Classes give a method for packaging information and usefulness together. Making another class makes another sort of item, enabling new occasions of that type to be made. Each class occurrence can have ascribes connected to it for keeping up its state. Class occasions can likewise have strategies (characterized by its group) for altering its state.

Contrasted and other programming dialects, Python's class instrument includes classes with at least a new sentence structure and semantics. It is a blend of the class instruments found in C++ and Modula-3. Python classes give all the standard highlights of Object-Oriented Programming: the class legacy component permits different base classes, an inferred class can supersede any techniques for its base class or classes, and a strategy can call the technique for a base class with a similar name.

Items can contain self-assertive sums and sorts of information. As is valid for modules, classes participate in the dynamic idea of Python: they are made at runtime and can be adjusted further after creation.

In C++ phrasing, ordinarily class individuals (counting the information individuals) are open (except seeing underneath Private Variables), and all part capacities are virtual. As in Modula-3, there are no shorthands for referencing the item's individuals from its techniques: the strategy work is announced with an unequivocal first contention speaking to the article, which is given verifiably by the call.

As in Smalltalk, classes themselves are objects. This gives semantics to bringing in and renaming. Not at all like C++ and Modula-3, worked in types can be utilized as base classes for expansion by the client. Likewise, as in C++, generally inherent administrators with exceptional sentence structure (number-crunching administrators, subscribing and so forth.) can be re-imagined for class examples. [1]

1.5 MATPLOTLIB

Matplotlib is a Python 2D plotting library which produces distribution quality figures in an assortment of printed copy positions and intuitive situations crosswise over stages. Matplotlib can be utilized in Python contents, the Python and IPython shells, the Jupyter

note pad, web application servers, and four graphical UI toolboxes.

Matplotlib attempts to make simple things simple and hard things conceivable. You can create plots, histograms, control spectra, bar outlines, error charts, scatterplots, and so forth., with only a couple of lines of code. For models, see the example plots and thumbnail display.

For basic plotting, the pyplot module gives a MATLAB-like interface, especially when joined with IPython. For the power client, you have full control of line styles, textual style properties, tomahawks properties, and so forth, through an article arranged interface or utilizing a lot of capacities recognizable to MATLAB clients.

1.6 NUMPY

NumPy is a general-purpose array-processing package. It provides a high-performance multidimensional array object and tools for working with these arrays.

- It is the fundamental package for scientific computing with Python. It contains various features including these important ones:
- A powerful N-dimensional array object
- Sophisticated (broadcasting) functions
- Tools for integrating C/C++ and Fortran code
- Useful linear algebra, Fourier transform, and random number capabilities

Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data.

Arbitrary data-types can be defined using Numpy which allows NumPy to seamlessly and speedily integrate with a wide variety of databases. [3]

1.6.1 Arrays in NumPy: NumPy's main object is the homogeneous multidimensional array.

It is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers.

- In NumPy dimensions are called axes. The number of axes is rank.
- NumPy's array class is called ndarray. It is also known by the alias array.

1.6.2 Array creation: There are various ways to create arrays in NumPy.

For example, you can create an array from a regular Python list or tuple using the array function. The type of the resulting array is deduced from the type of the elements in the sequences.

Often, the elements of an array are originally unknown, but its size is known. Hence, NumPy offers several functions to create arrays with initial placeholder content. These minimize the necessity of growing arrays, an expensive operation. For example: np.zeros, np.ones, np.full, np.empty, etc.

To create sequences of numbers, NumPy provides a function analogous to a range that returns arrays instead of lists. [3]

- arrange: returns evenly spaced values within a given interval. step size is specified.
- linspace: returns evenly spaced values within a given interval. num no. of elements are returned.
- Reshaping array: We can use the reshape method to reshape an array. Consider an array with shape (a1, a2, a3, …, aN). We can reshape and convert it into another array with shape (b1, b2, b3, …, bM). The only required condition is a1 x a2 x a3 … x aN = b1 x b2 x b3 … x bM. (i.e original size of the array remains unchanged.)
- Flatten array: We can use the flatten method to get a copy of array collapsed into one dimension. It accepts order argument. The default value is 'C' (for row-major order). Use 'F' for column-major order.

1.6.3 Array Indexing: Knowing the basics of array indexing is important for analysing and manipulating the array object. NumPy offers many ways to do array indexing.

- Slicing: Just like lists in python, NumPy arrays can be sliced. As arrays can be multidimensional, you need to specify a slice for each dimension of the array.
- Integer array indexing: In this method, lists are passed for indexing for each dimension. One to one mapping of corresponding elements is done to construct a new arbitrary array.
- Boolean array indexing: This method is used when we want to pick elements from an array which satisfy some condition.

CHAPTER 2
ARTIFICIAL NEURAL NETWORKS

2.1 What is a Neural Network?

An artificial neural network (ANN) is an information processing paradigm that is driven by biological nervous system, such as brain, process information. The main element of this paradigm is the novel structure of information processing systems. It is composed of a large number of highly interconnected processing elements (neurons) working together to solve specific problems. ANNs, like people, learn by example. An ANN is configured through the learning process for a specific application, such as pattern recognition or data classification. Learning in biological systems involves adjustment to the synaptic connections that exist between neurons. This is also true of ANN.

2.2 Historical background

Neural system reenactments give off an impression of being an ongoing improvement. Notwithstanding, this field was built up before the appearance of PCs, and has made due in any event one significant mishap and a few periods.

Numerous significant advances have been helped by the utilization of economical PC copies. Following an underlying time of eagerness, the field endure a time of dissatisfaction and notoriety. During this period when financing and expert help was insignificant, significant advances were made by generally scarcely any scientists. These pioneers had the option to create persuading innovation which outperformed the restrictions recognized by Minsky and Papert. Minsky and Papert, distributed a book (in 1969) in which they summarized a general sentiment of disappointment (against neural systems) among analysts, and was accordingly acknowledged by most moving forward without any more investigation. As of now, the neural system field appreciates a resurgence of intrigue and a relating increment in financing.

The principal fake neuron was created in 1943 by the neurophysiologist Warren McCulloch and the scholar Walter Pits. In any case, the innovation accessible around then didn't enable them to do excessively..

2.3 Why use neural networks?

Neural systems, with their wonderful capacity to get importance from confused or uncertain

information, can be utilized to remove designs and identify patterns that are too unpredictable to ever be seen by either people or other PC strategies. This master would then be able to be utilized to give projections offered new circumstances of intrigue and response "imagine a scenario in which" questions..

2.3.1 Other advantages

1. Adaptive learning: An ability to learn how to do tasks based on the data given for training or initial experience.
2. Self-Organisation: An ANN can create its own organisation or representation of the information it receives during learning time.
3. Real Time Operation: ANN computations may be carried out in parallel, and special hardware devices are being designed and manufactured which take advantage of this capability.
4. Fault Tolerance via Redundant Information Coding: Partial destruction of a network leads to the corresponding degradation of performance. However, some network capabilities may be retained even with major network damage.

2.4 Neural networks versus conventional computers

Neural systems adopt an alternate strategy to critical thinking than that of regular PCs. Traditional PCs utilizes an algorithmic methodology for example the PC adheres to a lot of directions so as to tackle an issue. Except if the particular advances that the PC needs to pursue are realized the PC can't take care of the issue. That confines the critical thinking ability of regular PCs to issues that we as of now comprehend and realize how to understand. In any case, PCs would be quite a lot more valuable on the off chance that they could do things that we don't actually have the foggiest idea how to do.

Neural systems process data along these lines the human mind does. The system is made out of countless profoundly interconnected handling elements (neurons) working in parallel to take care of a particular issue. Neural systems learn by model. They can't be modified to play out a particular errand. The models must be chosen cautiously generally helpful time is squandered or much more terrible the system may be working inaccurately.

The drawback is that in light of the fact that the system discovers how to take care of the issue without anyone else; its activity can be flighty.

Then again, regular PCs utilize an intellectual way to deal with critical thinking; the manner in which the issue is to comprehended must be known and expressed in little unambiguous directions. These guidelines are then changed over to a significant level language program and afterward into machine code that the PC can comprehend. These machines are absolutely unsurprising; in the event that anything turns out badly is because of a product or equipment issue.

Neural systems and traditional algorithmic PCs are not in rivalry yet supplement one another. There are errands are progressively fit to an algorithmic methodology like math activities and undertakings that are increasingly fit to neural systems. Significantly progressively, an enormous number of undertakings, require frameworks that utilization a mix of the two methodologies (ordinarily a customary PC is utilized to oversee the neural system) so as to perform at most extreme productivity.

Much is as yet obscure about how the cerebrum trains itself to process data, so speculations proliferate. In the human cerebrum, an ordinary neuron gathers signals from others through a large group of fine structures called dendrites. The neuron conveys spikes of electrical movement through a long, slight stand known as an axon, which parts into a huge number of branches. Toward the finish of each branch, a structure called a neurotransmitter changes over the movement from the axon into electrical impacts that repress or energize action from the axon into electrical impacts that restrain or energize action in the associated neurons. At the point when a neuron gets excitatory info that is adequately huge contrasted and its inhibitory information, it sends a spike of electrical movement down its axon. Learning happens by changing the adequacy of the neurotransmitters with the goal that the impact of one neuron on another progressions.

2.5 Human and Artificial Neurons - investigating the similarities

2.5.1 How the Human Brain Learns?

Much is still unknown as to how the brain trains itself to process information, so theory abounds. In the human brain, a common neuron collects signals from others through a host of fine structures called dendrites. The neuron sends spikes of electrical activity through a long, thin stand known as an axon, which splits into thousands of branches. At the end of each branch, a structure called a synapse converts activity from the axon into electrical effects that inhibit or stimulate activity from the axon into electrical effects that inhibit or

stimulate activity in the connected neurons. When a neuron receives excitatory input that is sufficiently larger than its inhibitory input, it sends a spike of electrical activity down the axon. Learning occurs by changing the effectiveness of synapse so that one neuron has an effect on other changess.

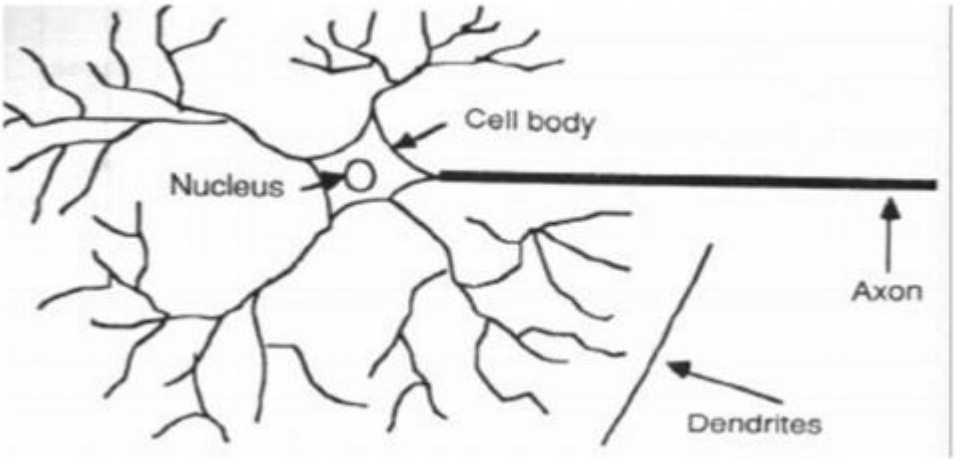

Fig 2.1 Components of Neuron [1]

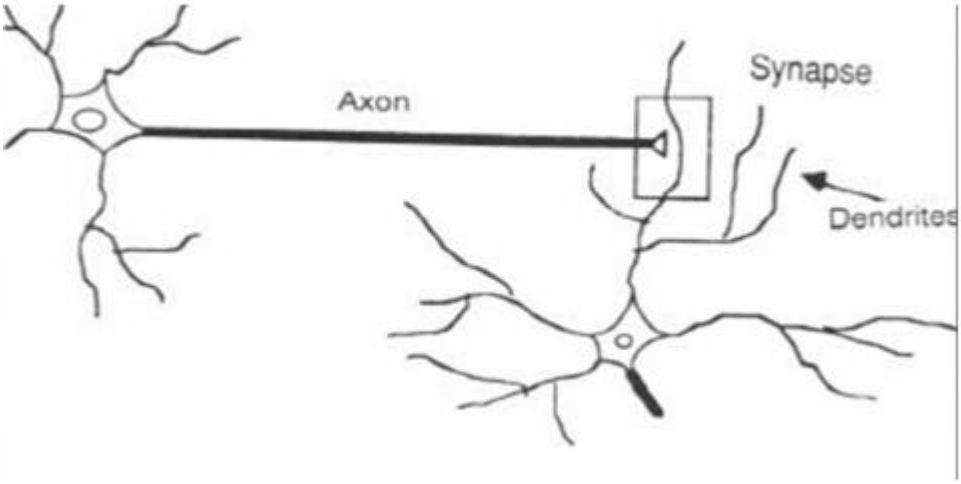

Fig 2.2 the Synapse [1]

2.5.2 From Human Neurons to Artificial Neurons

We first conduct these neural networks by trying to deduce the essential features of neurons and their interrelationships. Then we usually program a computer to emulate these features.

However, because our knowledge of neurons is incomplete and our computing power is

limited, our models are necessarily gross models of the actual network of neurons.

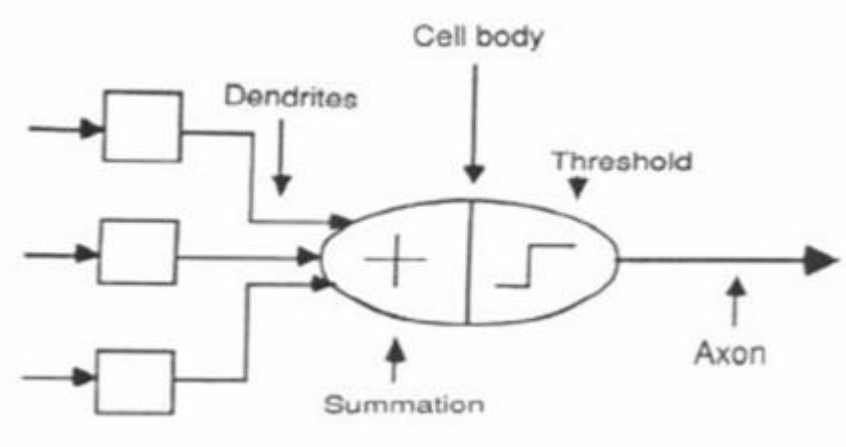

Fig 2.3 The neuron Model [2]

2.6. An Engineering Approach

2.6.1 A simple neuron

An artificial neuron is a device that has many inputs and one output. There are two modes of operation of a neuron; Training mode and usage mode. In training mode, the neuron can be trained to fire (or not) for particular input patterns. In usage mode, when a taught input pattern is detected on the input, its associated output current becomes the output. If the input pattern is not in the taught list of input patterns, the firing rule is used to determine whether to fire.

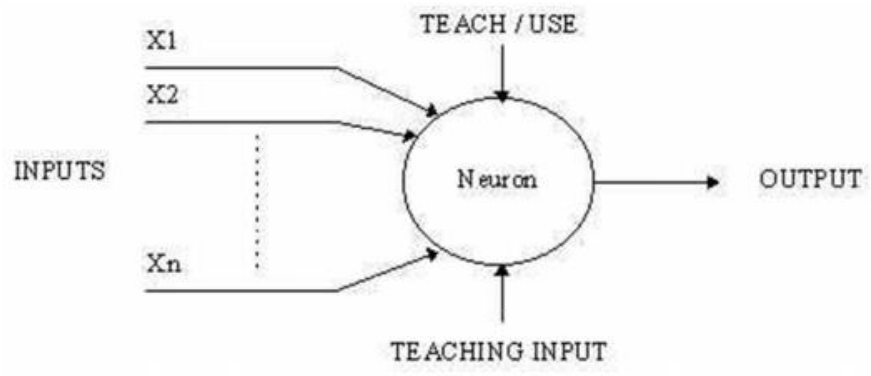

Fig 2.4 A Simple Neuron [2]

2.6.2 Firing rules

The terminating rule is a significant idea in neural systems and records for their high

adaptability. A terminating rule decides how one figures whether a neuron should fire for any information design. It identifies with all the info designs, not just the ones on which the hub was prepared.

A straightforward terminating rule can be executed by utilizing Hamming separation procedure. The standard goes as pursues:

Take an assortment of preparing designs for a hub, some of which cause it to fire (the 1-showed set of examples) and others which keep it from doing as such (the 0-instructed set). At that point the examples not in the assortment cause the hub to fire if, on correlation , they share more info components for all intents and purpose with the 'closest' design in the 1-showed set than with the 'closest' design in the 0-educated set. On the off chance that there is a tie, at that point the example stays in the vague state.

X1:		0	0	0	0	1	1	1	1
X2:		0	0	1	1	0	0	1	1
X3:		0	1	0	1	0	1	0	1
OUT:		0	0	0/1	0/1	0/1	1	0/1	1

Table 2.1 Truth table for firing [3]

The 'nearest' pattern is 000 which belongs in the 0-taught set. Thus the firing rule requires that the neuron should not fire when the input is 001. On the other hand, 011 is equally distant from two taught patterns that have different outputs and thus the output stays undefined (0/1).

By applying the firing in every column the following truth table is obtained;

X1:		0	0	0	0	1	1	1	1
X2:		0	0	1	1	0	0	1	1
X3:		0	1	0	1	0	1	0	1
OUT:		0	0	0	0/1	0/1	1	1	1

Table 2.2 Truth table after firing [3]

The difference between the two truth tables is called the generalization of the neuron. Therefore, the firing rule gives the neuron a sense of similarity and enables it to respond 'sensibly' to patterns not seen during training.

2.6.3 Pattern Recognition - an example

A significant utilization of neural systems is design acknowledgment. Example acknowledgment can be actualized by utilizing a feed-forward neural system that has been prepared in like manner.

During preparing, the system is prepared to connect yields with input designs. At the point when the system is utilized, it recognizes the info example and attempts to yield the related yield design. The intensity of neural systems becomes animated when an example that has no yield related with it, is given as an information. For this situation, the system gives the yield that relates to an encouraged information design that is least unique in relation to the given example.

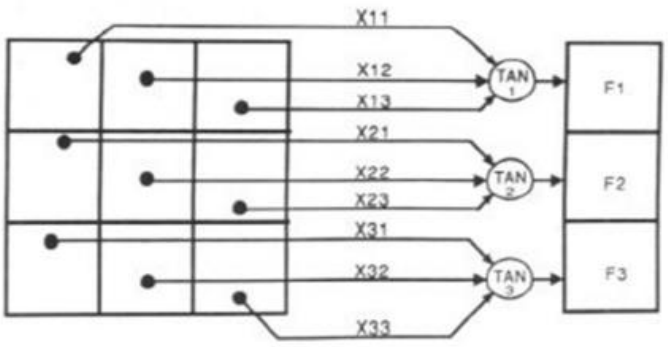

Fig 2.5 Pattern Recognition [2]

For example:

The network is trained to recognise the patterns T and H. The associated patterns are all black and all white respectively as shown below

Fig 2.6 Black and White associative pattern [2]

If we represent black squares with 0 and white squares with 1 then the truth tables for the 3 neurones after generalisation are;

X11:		0	0	0	0	1	1	1	1
X12:		0	0	1	1	0	0	1	1
X13:		0	1	0	1	0	1	0	1
OUT:		0	0	1	1	0	0	1	1

Table 2.3 Top Neuron [3]

X21:		0	0	0	0	1	1	1	1
X22:		0	0	1	1	0	0	1	1
X23:		0	1	0	1	0	1	0	1
OUT:		1	0/1	1	0/1	0/1	0	0/1	0

Table 2.4 Middle Neuron [3]

X21:		0	0	0	0	1	1	1	1
X22:		0	0	1	1	0	0	1	1
X23:		0	1	0	1	0	1	0	1
OUT:		1	0	1	1	0	0	1	0

Table 2.5 Bottom Neuron [3]

From the tables it can be seen the following associations can be extracted:

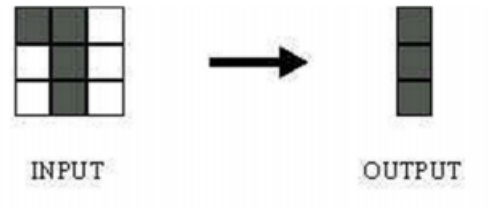

Fig 2.7 T pattern [2]

In this case, it is obvious that the output should be all blacks since the input pattern is almost the same as the 'T' pattern

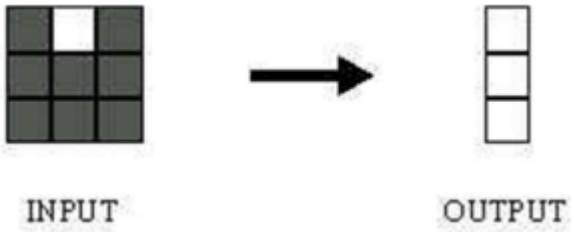

Fig 2.8 H pattern [2]

Here also, it is obvious that the output should be all whites since the input pattern is almost the same as the 'H' pattern.

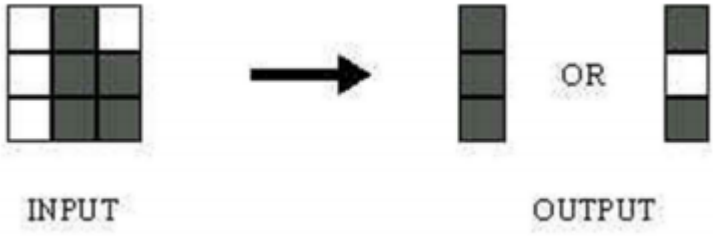

Fig 2.9 H pattern [2]

Here, the top row is 2 errors away from a T and 3 from an H. So the top output is black. The

middle row is 1 error away from both T and H so the output is random. The bottom row is 1 error away from T and 2 away from H. Therefore, the output is black. The total output of the network is still in favor of the T shape.

2.6.4 A more complicated neuron

An increasingly refined neuron is the McCulloch and Pitts model (MCP). The thing that matters is that the sources of info are 'weighted', the impact that each information has at basic leadership is reliant on the heaviness of the specific information. The heaviness of an information is a number which when increased with the information gives the weighted information. These weighted sources of info are then included and on the off chance that they surpass a pre-set edge esteem, the neuron fires. In some other case the neuron doesn't fire.

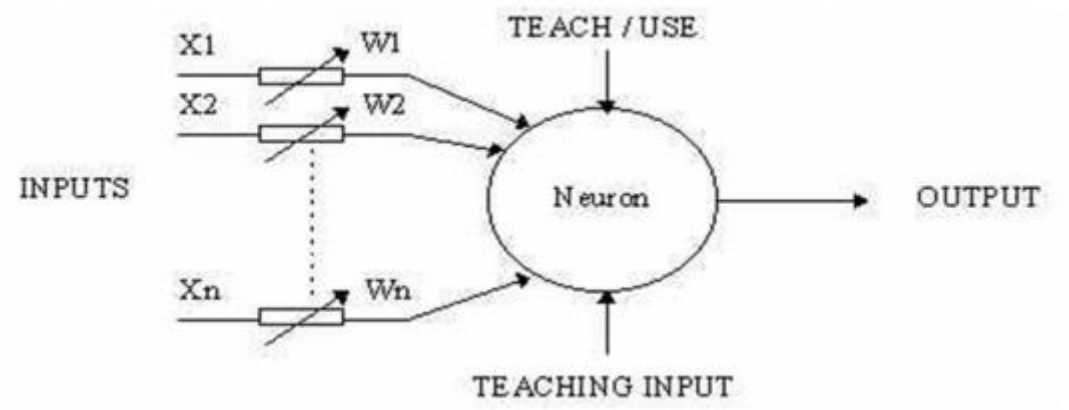

Fig 2.10 A MCP Neuron [2]

In mathematical terms, the neuron fires if and only if;

$$X1W1 + X2W2 + X3W3 + ... > T$$

The addition of input weights and of the threshold makes this neuron a very flexible and powerful one. The MCP neuron has the ability to adapt to a particular situation by changing its weights and/or threshold. Various algorithms exist that cause the neuron to 'adapt'; the most used ones are the Delta rule and the back error propagation. The former is used in feed forward networks and the latter in feedback networks.

2.7 Architecture of neural networks

2.7.1 Feed-forward networks

Feed-forward ANNs allow signals to travel one way only; from input to output. There is no feedback (loops) i.e. the output of any layer does not affect that same layer. Feed-forward ANNs tend to be straight forward networks that associate inputs with outputs. They are extensively used in pattern recognition. This type of organization is also referred to as bottom-up or top-down.

2.7.2 Feedback networks

Feedback networks can have signals travelling in both directions by introducing loops in the network. Feedback networks are very powerful and can get extremely complicated.

Feedback networks are dynamic; their 'state' is changing continuously until they reach an equilibrium point. They remain at the equilibrium point until the input changes and a new equilibrium needs to be found. Feedback architectures are also referred to as interactive or recurrent, although the latter term is often used to denote feedback connections in single layer organizations.

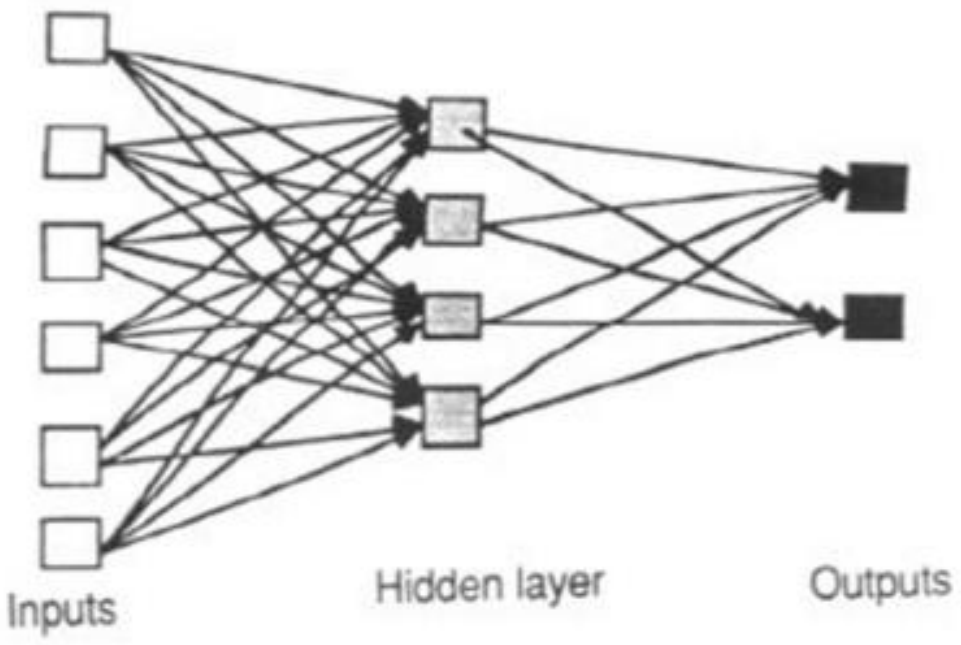

Fig 2.11 An example of simple feed forward neuron [2]

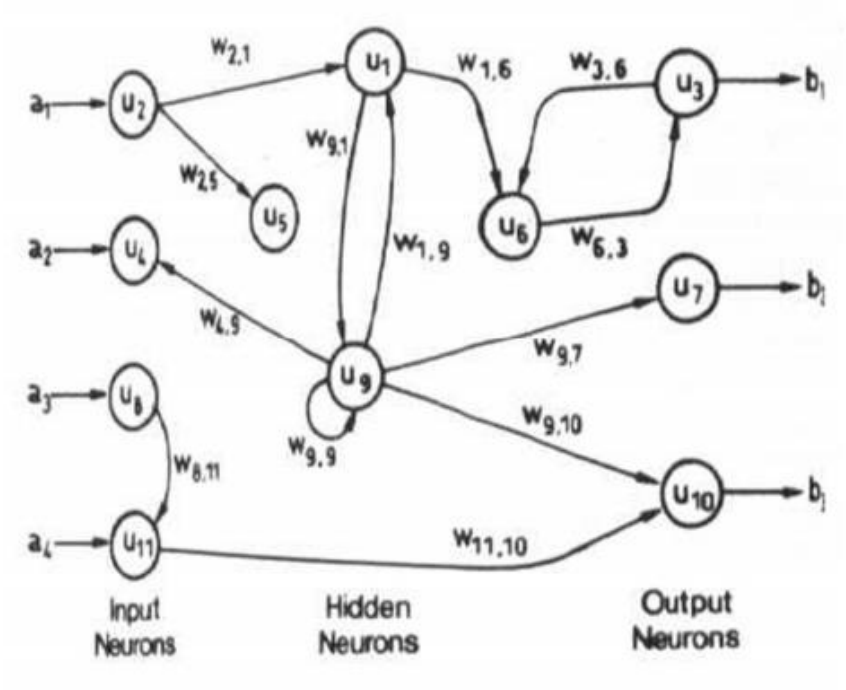

Fig 2.12 An example of complicated neuron [2]

2.7.3 Network layers

The commonest type of artificial neural network consists of three groups, or layers, of units:

A layer of "input" units is connected to a layer of "hidden" units, which is connected to a layer of "output" units.

1. The activity of the input units represents the raw information that is fed into the network.
2. The activity of each hidden unit is determined by the activities of the input units and the weights on the connections between the input and the hidden units.
3. The behavior of the output units depends on the activity of the hidden units and the weights between the hidden and output units.

This basic sort of system is fascinating on the grounds that the shrouded units are allowed to build their own portrayals of the info. The loads between the info and shrouded units decide when each concealed unit is dynamic, thus by changing these loads, a shrouded unit can pick what it speaks to.

We likewise recognize single-layer and multi-layer models. The single-layer association, where all units are associated with each other, comprises the most broad case and is of more

potential computational power than progressively organized multi-layer associations. In multi-layer systems, units are regularly numbered by layer, rather than following a worldwide numbering.

2.7.4 Perceptron

The most influential work on neural nets in the 60's went under the heading of 'Perceptrons' a term coined by Frank Rosenblatt. The perceptron turns out to be an MCP model (neuron with weighted inputs) with some additional, fixed, preprocessing. Units labeled A1, A2, Aj, Ap are called association units and their task is to extract specific, localized featured from the input images. Perceptrons mimic the basic idea behind the mammalian visual system. They were mainly used in pattern recognition even though their capabilities extended a lot more.

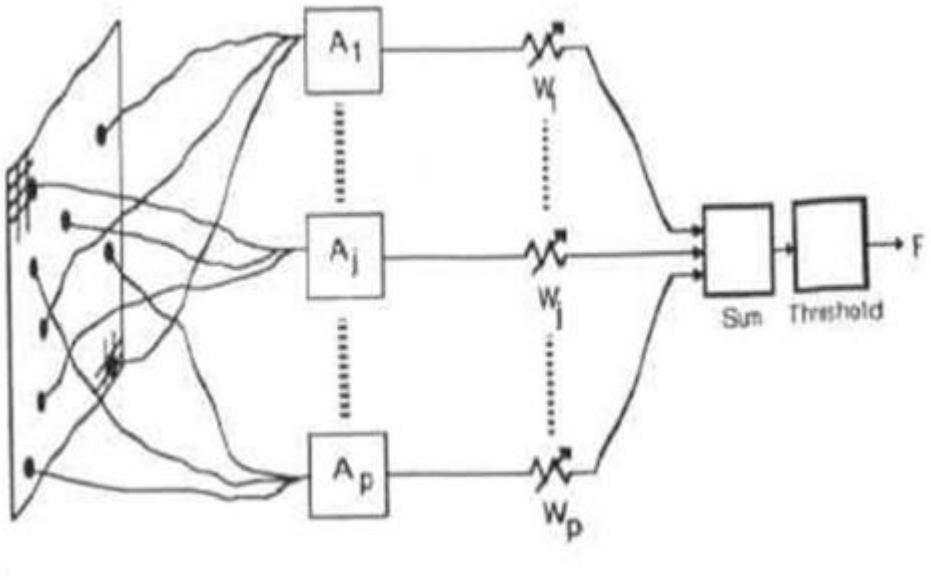

Fig 2.13 An example of a neural net [2]

2.7.5 The Learning Process

The memorization of patterns and the subsequent response of the network can be categorized into two general paradigms:

1. Associative mapping in which the network learns to produce a particular pattern on the set of input units whenever another particular pattern is applied on the set of input units. The associative mapping can generally be broken down into two mechanisms:

2. Auto-association: an input pattern is associated with itself and the states of input and output units coincide. This is used to provide pattern competition, ie to produce a pattern whenever a portion of it or a distorted pattern is presented. In the second case, the network actually stores pairs of patterns building an association between two sets of patterns.
3. hetero-association: is related to two recall mechanisms:
4. nearest-neighbor recall, where the output pattern produced corresponds to the input pattern stored, which is closest to the pattern presented, and
5. Interpolative recall, where the output pattern is a similarity dependent interpolation of the patterns stored corresponding to the pattern presented. Yet another paradigm, which is a variant associative mapping is classification, ie when there is a fixed set of categories into which the input patterns are to be classified.
6. Regularity detection in which units learn to respond to particular properties of the input patterns. Whereas in associative mapping the network stores the relationships among patterns, in regularity detection the response of each unit has a particular 'meaning'. This type of learning mechanism is essential for feature discovery and knowledge representation.

Every neural network possesses knowledge which is contained in the values of the connections weights. Modifying the knowledge stored in the network as a function of experience implies a learning rule for changing the values of the weights.

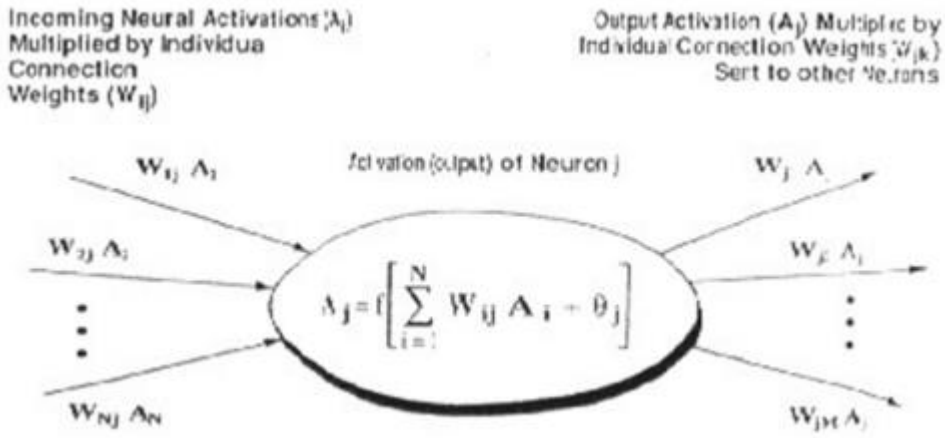

Fig 2.14 Activation of neural network [2]

Information is stored in the weight matrix W of a neural network. Learning is the determination of the weights. Following the way learning is performed, we can distinguish two major categories of neural networks:

1. Fixed networks in which the weights cannot be changed, i.e. dW/dt=0. In such networks, the weights are fixed a priori according to the problem to solve.
2. Adaptive networks which are able to change their weights, i.e. dW/dt not= 0.

All learning methods used for adaptive neural networks can be classified into two major categories:

Managed realizing which joins an outer instructor, so each yield unit is determined what its ideal reaction to include signals should be. During the learning procedure worldwide data might be required. Ideal models of managed learning incorporate blunder rectification learning, refinement learning and stochastic figuring out how to give some examples. A significant issue concerning managed learning is the issue of mistake union, ie the minimization of blunder between the ideal and figured unit esteems. The point is to decide a lot of loads which limits the mistake. One surely understood strategy, which is regular to many learning ideal models is the least mean square (LMS) union.

Unaided learning utilizes no outer educator and depends on just nearby data. It is likewise alluded to as self-association, as in it self-sorts out information displayed to the system and identifies their emanant aggregate properties. We state that a neural system learns disconnected if the learning stage and the activity stage are particular. A neural system learns on-line in the event that it learns and works simultaneously. Typically, regulated learning is performed disconnected, while unaided learning is performed on-line.

2.7.6 Transfer Function

The behavior of an ANN (Artificial Neural Network) depends on both the weights and the input-output function (transfer function) that is specified for the units. This function typically falls into one of three categories:

1. linear (or ramp)
2. threshold
3. sigmoid

For direct units, the yield action is corresponding to the all out weighted yield. For edge units, the yield is set at one of two levels, contingent upon whether the absolute info is more prominent than or not exactly some limit esteem.

For sigmoid units, the yield fluctuates constantly yet not directly as the information changes. Sigmoid units look somewhat like genuine neurons than do straight or edge units, however every one of the three must be viewed as unpleasant approximations.

To make a neural system that plays out some particular errand, we should pick how the units are associated with each other, and we should set the loads on the associations suitably. The associations decide if it is workable for one unit to impact another. The loads indicate the quality of the impact. We can instruct a three-layer system to play out a specific assignment by utilizing the accompanying method:

1. We present the network with training examples, which consist of a pattern of activities for the input units together with the desired pattern of activities for the output units.
2. We determine how closely the actual output of the network matches the desired output.
3. We change the weight of each connection so that the network produces a better approximation of the desired output.

2.7.7 An Example to illustrate the above teaching procedure:

Assume that we want a network to recognize hand-written digits. We might use an array of, say, 256 sensors, each recording the presence or absence of ink in a small area of a single digit. The network would therefore need 256 input units (one for each sensor), 10 output units (one for each kind of digit) and a number of hidden units.

For each kind of digit recorded by the sensors, the network should produce high activity in the appropriate output unit and low activity in the other output units.

To prepare the system, we present a picture of a digit and look at the genuine action of the 10 yield units with the ideal action. We at that point figure the mistake, which is characterized as the square of the distinction between the real and the ideal exercises. Next we change the heaviness of every association in order to lessen the blunder. We rehash this preparation procedure for a wide range of pictures of each various pictures of every sort of digit until the

system arranges each picture effectively.

To actualize this methodology, we have to figure the blunder subordinate for the weight (EW) so as to change the weight by a sum that is corresponding to the rate at which the mistake changes as the weight is changed. One approach to ascertain the EW is to bother a weight somewhat and see how the blunder changes. However, that strategy is wasteful in light of the fact that it requires a different annoyance for every one of the numerous loads.

2.7.8 The Back-Propagation Algorithm

In order to train a neural network to perform some task, we must adjust the weights of each unit in such a way that the error between the desired output and the actual output is reduced.

This process requires that the neural network compute the error derivative of the weights (EW). In other words, it must calculate how the error changes as each weight is increased or decreased slightly. The back propagation algorithm is the most widely used method for determining the EW.

The back-spread calculation is most effortless to comprehend if every one of the units in the system are straight. The calculation registers every EW by first figuring the EA, the rate at which the mistake changes as the action level of a unit is changed. For yield units, the EA is basically the contrast between the real and the ideal yield. To process the EA for a shrouded unit in the layer just before the yield layer, we initially recognize every one of the loads between that concealed unit and the yield units to which it is associated. We then increase those loads by the EAs of those yield units and include the items. This aggregate equivalent the EA for the picked concealed unit. In the wake of ascertaining every one of the EAs in the shrouded layer just before the yield layer, we can figure in like design the EAs for different layers, moving from layer to layer toward a path inverse to the manner in which exercises proliferate through the system. This is the thing that gives back proliferation its name. When the EA has been figured for a unit, it is straight forward to register the EW for every approaching association of the unit. The EW is the result of the EA and the action through the approaching association.

Note that for non-direct units, (see Appendix C) the back-spread calculation incorporates an additional progression. Before back-proliferating, the EA must be changed over into the EI,

the rate at which the mistake changes as the absolute info got by a unit is changed.

2.8 Applications of neural networks

Since neural networks are best at identifying patterns or trends in data, they are well suited for prediction or forecasting needs including:

1. sales forecasting
2. industrial process control
3. customer research
4. data validation
5. risk management
6. target marketing

ANN are also used in the following specific paradigms: recognition of speakers in communications; diagnosis of hepatitis; recovery of telecommunications from faulty software; interpretation of multi meaning Chinese words; undersea mine detection; texture analysis; three-dimensional object recognition; hand-written word recognition; and facial recognition.

2.8.1 Neural networks in medicine

Artificial Neural Networks (ANN) are as of now a 'hot' investigate territory in medication and it is accepted that they will get broad application to biomedical frameworks in the following hardly any years. Right now, the exploration is for the most part on displaying portions of the human body and perceiving ailments from different sweeps (for example cardiograms, CAT checks, ultrasonic outputs, and so forth.).

Neural systems are perfect in perceiving ailments utilizing checks since there is no compelling reason to give a particular calculation on the best way to recognize the ailment. Neural systems learn by model so the subtleties of how to perceive the illness are not required. What is required is a lot of models that are illustrative of the considerable number of varieties of the infection. The amount of models isn't as significant as the 'amount'. The models should be chosen cautiously if the framework is to perform dependably and proficiently.

1. **Modeling and Diagnosing the Cardiovascular System**

Neural Networks are utilized tentatively to display the human cardiovascular framework. Conclusion can be accomplished by building a model of the cardiovascular arrangement of an individual and contrasting it and the constant physiological estimations taken from the patient. On the off chance that this routine is completed normally, potential hurtful ailments can be recognized at a beginning time and in this way make the procedure of battling the sickness a lot simpler.

A model of a person's cardiovascular framework must copy the relationship among physiological factors (i.e., pulse, systolic and diastolic blood weights, and breathing rate) at various physical movement levels. On the off chance that a model is adjusted to an individual, at that point it turns into a model of the physical state of that person. The test system should have the option to adjust to the highlights of any person without the supervision of a specialist. This requires a neural system.

Another explanation that legitimizes the utilization of ANN innovation is the capacity of ANNs to give sensor combination which is the joining of qualities from a few unique sensors. Sensor combination empowers the ANNs to learn complex connections among the individual sensor esteems, which would somehow or another be lost if the qualities were separately dissected. In medicinal demonstrating and conclusion, this suggests despite the fact that every sensor in a set might be delicate just to a particular physiological variable; ANNs are equipped for recognizing complex ailments by melding the information from the individual biomedical sensors.

2. **Electronic noses**

ANNs are used experimentally to implement electronic noses. Electronic noses have several potential applications in telemedicine. Telemedicine is the practice of medicine over long distances via a communication link. The electronic nose would identify odors in the remote surgical environment. These identified odors would then be electronically transmitted to another site where an door generation system would recreate them. Because the sense of smell can be an important sense to the surgeon, telesmell would enhance telepresent surgery.

3. **Instant Physician**

An application developed in the mid-1980s called the "instant physician" trained an auto associative memory neural network to store a large number of medical records, each of

which includes information on symptoms, diagnosis, and treatment for a particular case. After training, the net can be presented with input consisting of a set of symptoms; it will then find the full stored pattern that represents the "best" diagnosis and treatment.

2.8.2 Neural Networks in business

Business is a diverted field with several general areas of specialization such as accounting or financial analysis. Almost any neural network application would fit into one business area or financial analysis.

There is some potential for using neural networks for business purposes, including resource allocation and scheduling. There is also a strong potential for using neural networks for database mining, that is, searching for patterns implicit within the explicitly stored information in databases. Most of the funded work in this area is classified as proprietary.

Thus, it is not possible to report on the full extent of the work going on. Most work is applying neural networks, such as the Hopfield-Tank network for optimization and scheduling.

1. Marketing

There is a marketing application which has been integrated with a neural network system. The Airline Marketing Tactician (a trademark abbreviated as AMT) is a computer system made of various intelligent technologies including expert systems. A feed forward neural network is integrated with the AMT and was trained using back-propagation to assist the marketing control of airline seat allocations. The adaptive neural approach was amenable to rule expression. Additionally, the application's environment changed rapidly and constantly, which required a continuously adaptive solution. The system is used to monitor and recommend booking advice for each departure. Such information has a direct impact on the profitability of an airline and can provide a technological advantage for users of the system.

While it is significant that neural networks have been applied to this problem, it is also important to see that this intelligent technology can be integrated with expert systems and other approaches to make a functional system. Neural networks were used to discover the influence of undefined interactions by the various variables. While these interactions were not defined, they were used by the neural system to develop useful conclusions. It is also noteworthy to see that neural networks can influence the bottom line.

2. Credit Evaluation

The HNC Company, founded by Robert Hecht-Nielsen, has developed several neural network applications. One of them is the Credit Scoring system which increases the profitability of the existing model up to 27%. The HNC neural systems were also applied to mortgage screening. A neural network automated mortgage insurance underwriting system was developed by the Nestor Company. This system was trained with 5048 applications of which 2597 were certified. The data related to property and borrower qualifications. In a conservative mode the system agreed on the underwriters on 97% of the cases. In the liberal model the system agreed 84% of the cases. This is system run on an Apollo DN3000 and used 250K memory while processing a case file in approximately 1 sec

CHAPTER 3
CONVOLUTIONAL NEURAL NETWORKS

Artificial Intelligence has been witnessing a monumental growth in bridging the gap between the capabilities of humans and machines. Researchers and enthusiasts alike, work on numerous aspects of the field to make amazing things happen. One of many such areas is the domain of Computer Vision.

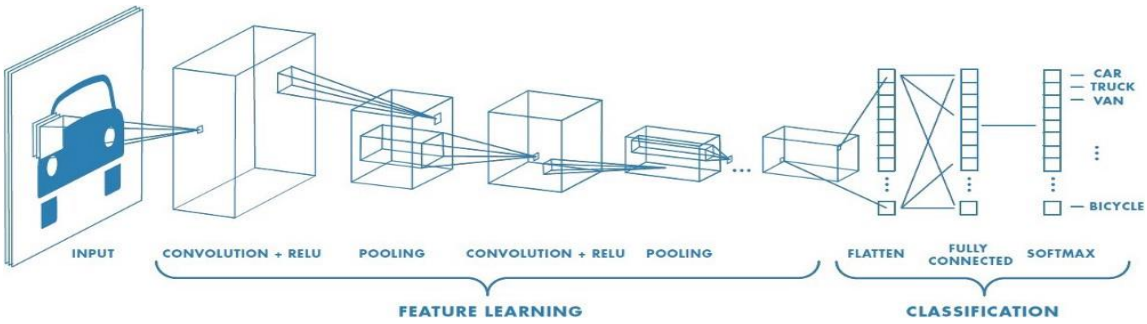

Fig 3.1 CNN Standard Architecture [3]

The agenda for this field is to enable machines to view the world as humans do, perceive it in a similar manner and even use the knowledge for a multitude of tasks such as Image & Video recognition, Image Analysis & Classification, Media Recreation, Recommendation Systems, Natural Language Processing, etc. The advancements in Computer Vision with Deep Learning has been constructed and perfected with time, primarily over one algorithm — a **Convolutional Neural Network**.

3.1 Introduction

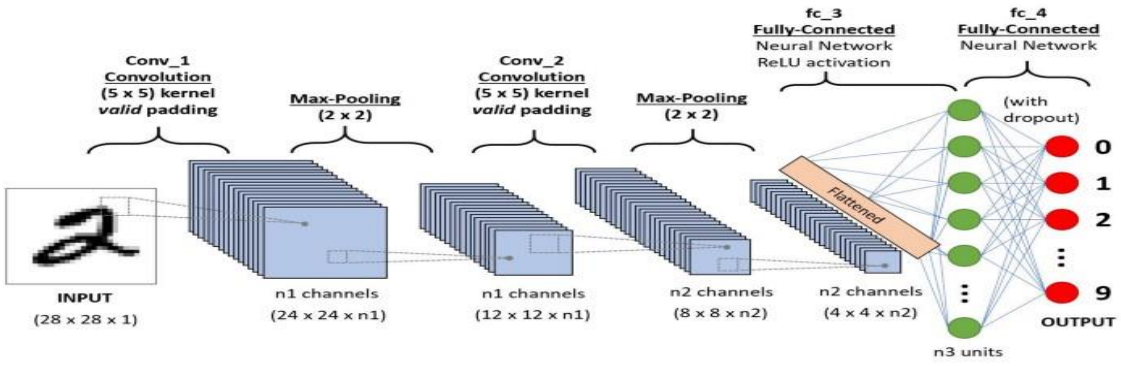

Fig 3.2 CNN Layer Structure [1]

A **Convolutional Neural Network (ConvNet/CNN)** is a Deep Learning algorithm which can take in an input image, assign importance (learnable weights and biases) to various aspects/objects in the image and be able to differentiate one from the other. The pre-processing required in a ConvNet is much lower as compared to other classification algorithms. While in primitive methods filters are hand-engineered, with enough training, ConvNets can learn these filters/characteristics.

The architecture of a ConvNet is analogous to that of the connectivity pattern of Neurons in the Human Brain and was inspired by the organization of the Visual Cortex. Individual neurons respond to stimuli only in a restricted region of the visual field known as the Receptive Field. A collection of such fields overlaps to cover the entire visual area.

3.2 Why ConvNets over Feed-Forward Neural Nets?

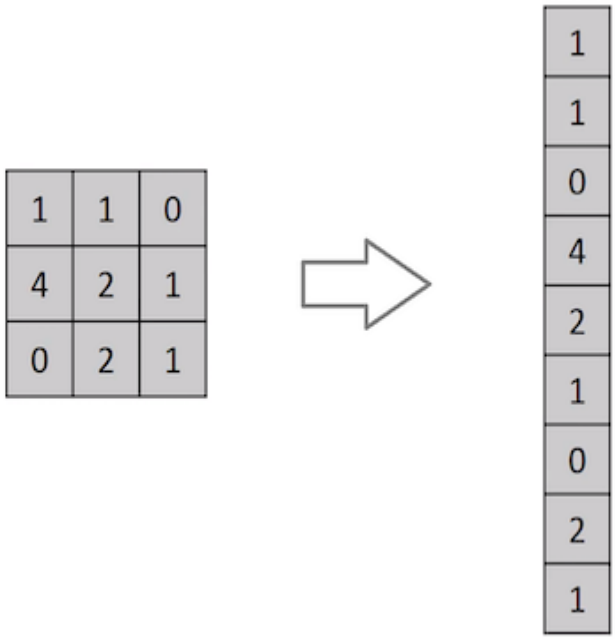

Fig 3.3 Flattening of 3x3 matric to 9x1 [3]

In cases of extremely basic binary images, the method might show an average precision score while performing prediction of classes but would have little to no accuracy when it comes to complex images having pixel dependencies throughout.

A ConvNet can **successfully capture the Spatial and Temporal dependencies** in an image

through the application of relevant filters. The architecture performs a better fitting to the image dataset due to the reduction in the number of parameters involved and reusability of weights. In other words, the network can be trained to understand the sophistication of the image better.

3.3 Input Image

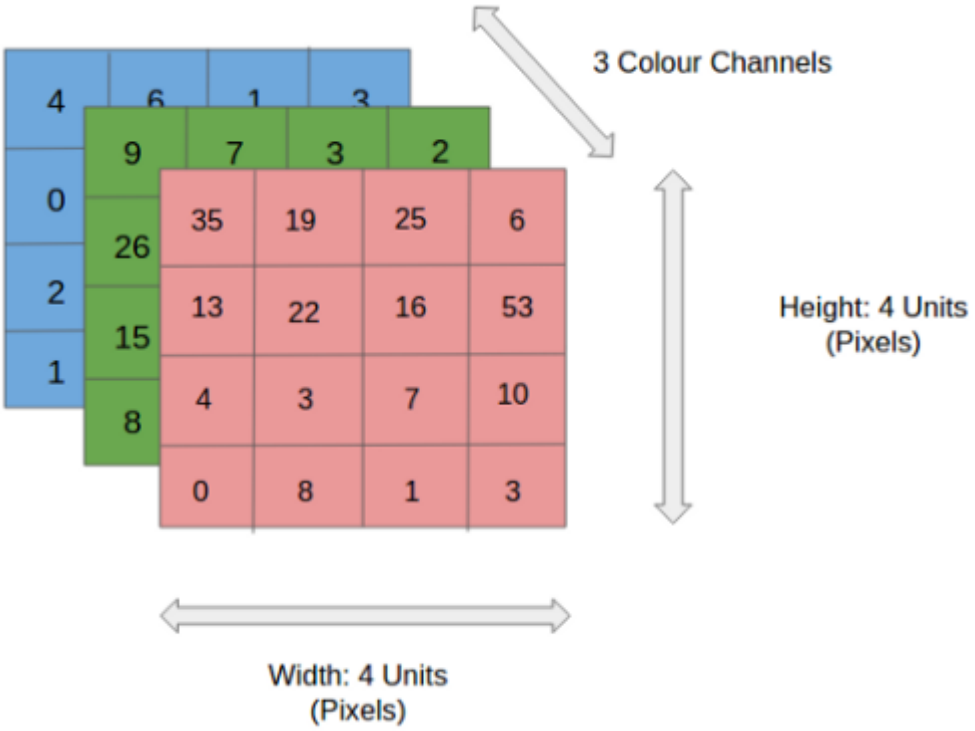

Fig 3.4 4x4x3 RBGB Image [4]

In the figure, we have an RGB image which has been separated by its three colour planes — Red, Green, and Blue. There are several such colour spaces in which images exist — Grayscale, RGB, HSV, CMYK, etc.

The role of the ConvNet is to reduce the images into a form which is easier to process, without losing features which are critical for getting a good prediction. This is important when we are to design an architecture which is not only good at learning features but also is scalable to massive datasets.

3.4 Convolution Layer — The Kernel

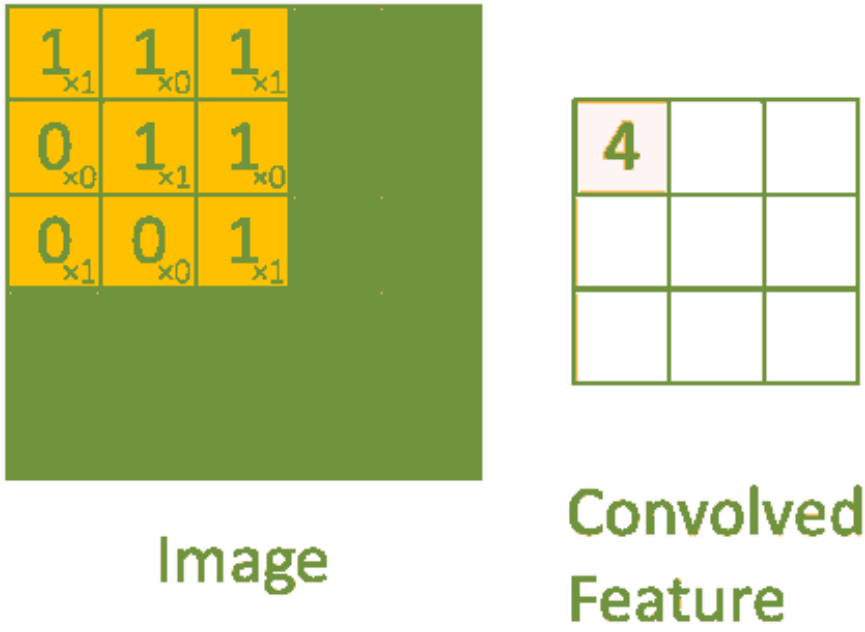

Fig 3.5 Convoluting a 5x5x1 image with a 3x3x1 kernel to get a 3x3x1 convolved feature, Image Dimensions = 5 (Height) x 5 (Breadth) x 1 (Number of channels, eg. RGB) [4]

In the above demonstration, the green section resembles our **5x5x1 input image, I**. The element involved in carrying out the convolution operation in the first part of a Convolutional Layer is called the **Kernel/Filter, K**, represented in the colour yellow. We have selected **K as a 3x3x1 matrix.**

Kernel/Filter, K = 1 0 1
0 1 0
1 0 1

The Kernel shifts 9 times because of **Stride Length = 1 (Non-Strided)**, every time performing a **matrix multiplication operation between K and the portion P of the image** over which the kernel is hovering.

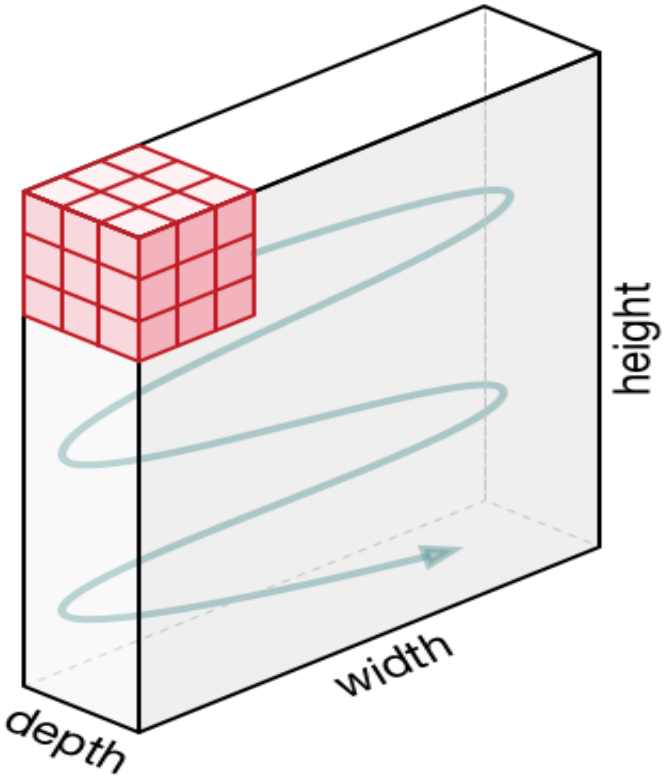

Fig 3.6 Movement of the Kernel [2]

The filter moves to the right with a certain Stride Value till it parses the complete width. Moving on, it hops down to the beginning (left) of the image with the same Stride Value and repeats the process until the entire image is traversed.

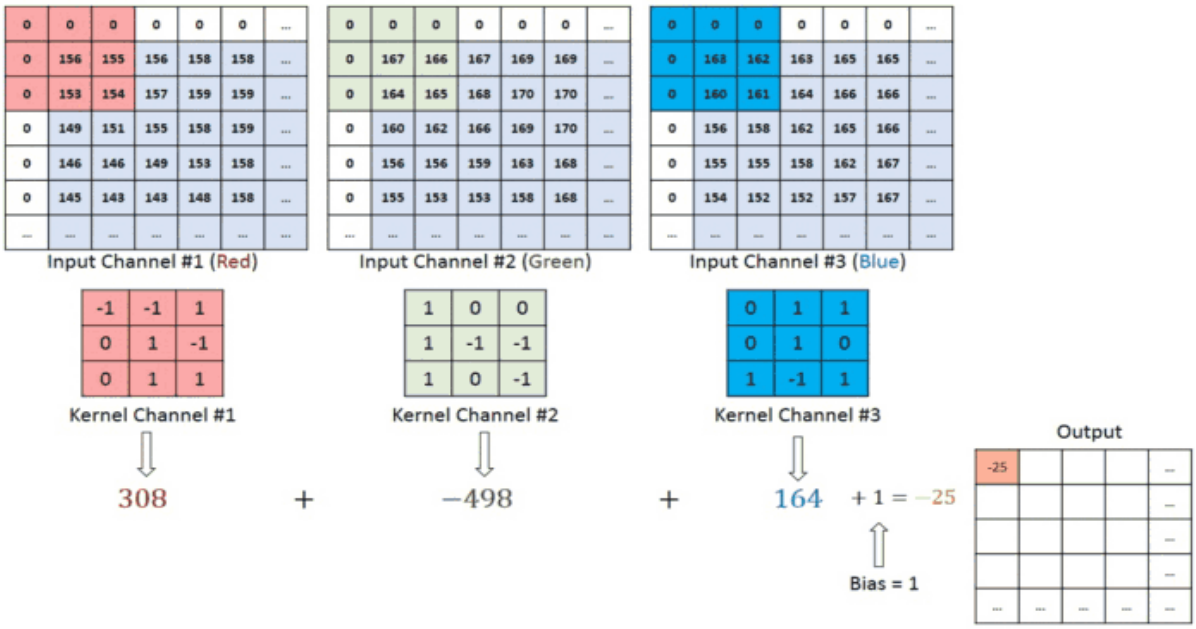

Fig 3.7 Convolution operation on a MxNx3 image matrix with a 3x3x3 Kernel [1]

In the case of images with multiple channels (e.g. RGB), the Kernel has the same depth as that of the input image. Matrix Multiplication is performed between Kn and In stack ([K1, I1]; [K2, I2]; [K3, I3]) and all the results are summed with the bias to give us a squashed one-depth channel Convoluted Feature Output.

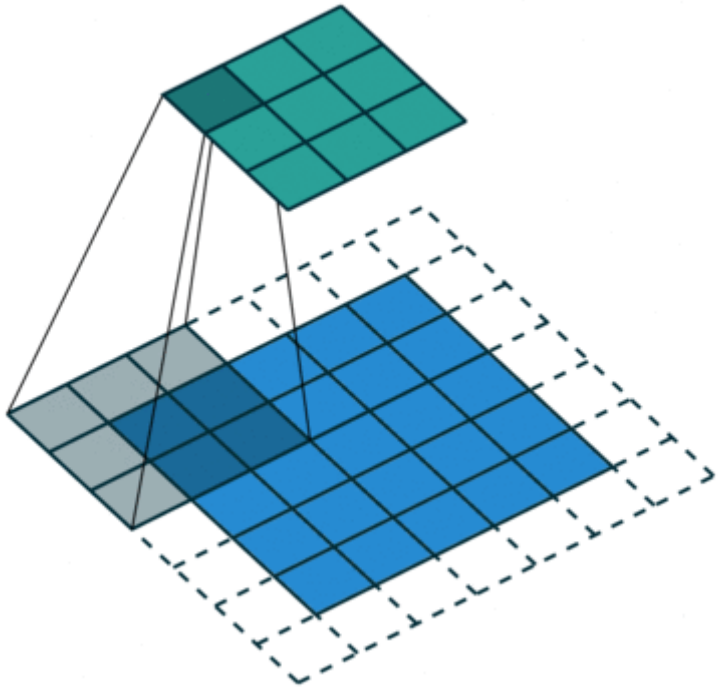

Fig 3.8 Convolution Operation with Stride Length = 2 [1]

The objective of the Convolution Operation is to **extract the high-level features** such as edges, from the input image. ConvNets need not be limited to only one Convolutional Layer. Conventionally, the first ConvLayer is responsible for capturing the Low-Level features such as edges, colour, gradient orientation, etc. With added layers, the architecture adapts to the High-Level features as well, giving us a network, which has the wholesome understanding of images in the dataset, similar to how we would.

There are two types of results to the operation — one in which the convolved feature is reduced in dimensionality as compared to the input, and the other in which the dimensionality is either increased or remains the same. This is done by applying **Valid Padding** in case of the former, or **Same Padding** in the case of the latter.

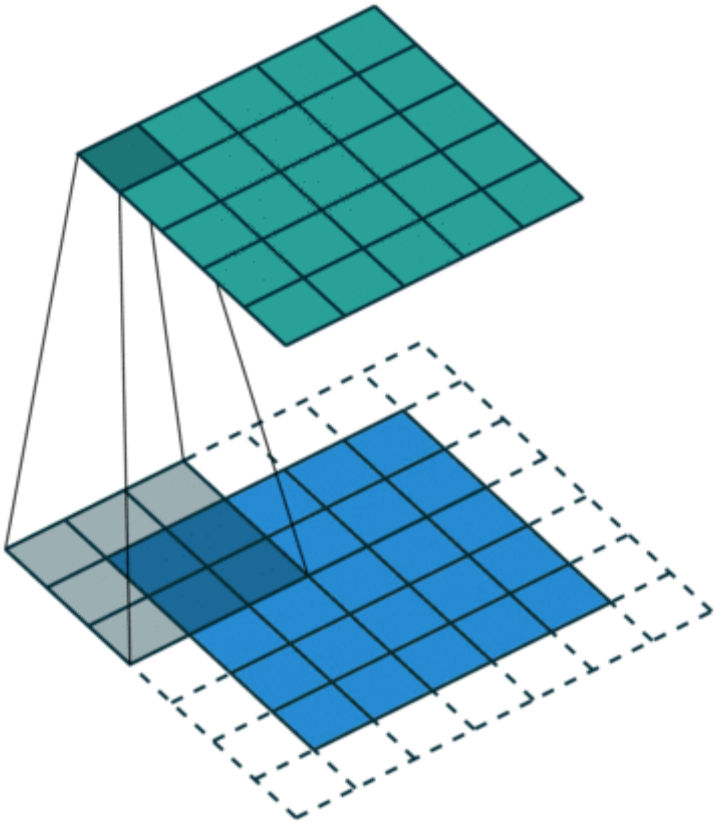

Fig 3.9 SAME padding: 5x5x1 image is padded with 0s to create a 6x6x1 image

When we augment the 5x5x1 image into a 6x6x1 image and then apply the 3x3x1 kernel over it, we find that the convolved matrix turns out to be of dimensions 5x5x1. Hence the name — **Same Padding**.

On the other hand, if we perform the same operation without padding, we are presented with a matrix which has dimensions of the Kernel (3x3x1) itself — **Valid Padding**.

After the celebrated victory of AlexNet at the LSVRC2012 classification contest, deep Residual Network was arguably the most ground-breaking work in the computer vision/deep learning community in the last few years. ResNet makes it possible to train up to hundreds or even thousands of layers and still achieves compelling performance.

Taking advantage of its powerful representational ability, the performance of many computer vision applications other than image classification have been boosted, such as object detection and face recognition.

3.5 Deep Residual Networks (ResNet)

According to the universal approximation theorem, given enough capacity, we know that a feedforward network with a single layer is sufficient to represent any function. However, the layer might be massive and the network is prone to overfitting the data. Therefore, there is a common trend in the research community that our network architecture needs to go deeper.

Since AlexNet, the state-of-the-art CNN architecture is going deeper and deeper. While AlexNet had only 5 convolutional layers, the VGG network [3] and GoogleNet (also codenamed Inception_v1) [4] had 19 and 22 layers respectively.

However, increasing network depth does not work by simply stacking layers together. Deep networks are hard to train because of the notorious vanishing gradient problem — as the gradient is back-propagated to earlier layers, repeated multiplication may make the gradient infinitively small. As a result, as the network goes deeper, its performance gets saturated or even starts degrading rapidly.

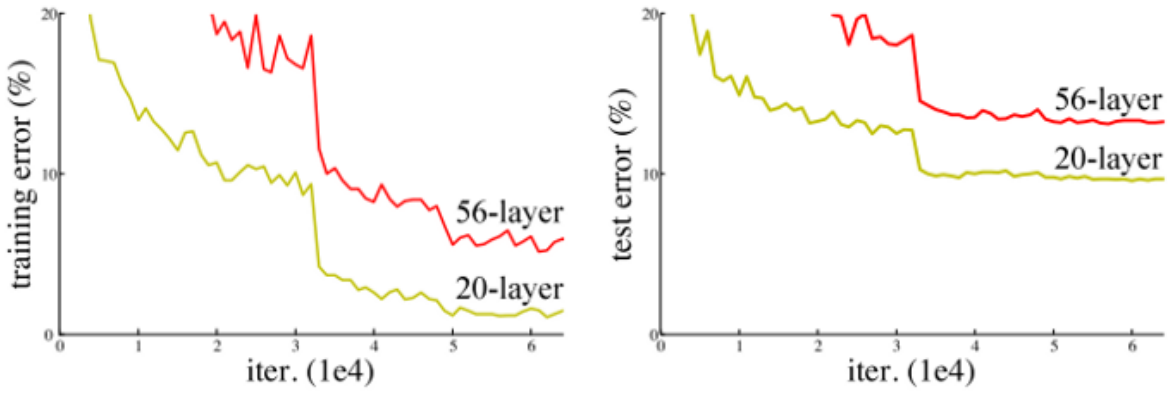

Fig 3.10 Increasing network depth leads to worse performance

Before ResNet, there had been several ways to deal the vanishing gradient issue, for instance, adds an auxiliary loss in a middle layer as extra supervision, but none seemed to really tackle the problem once and for all.

The core idea of ResNet is introducing a so-called “identity shortcut connection” that skips one or more layers, as shown in the following figure:

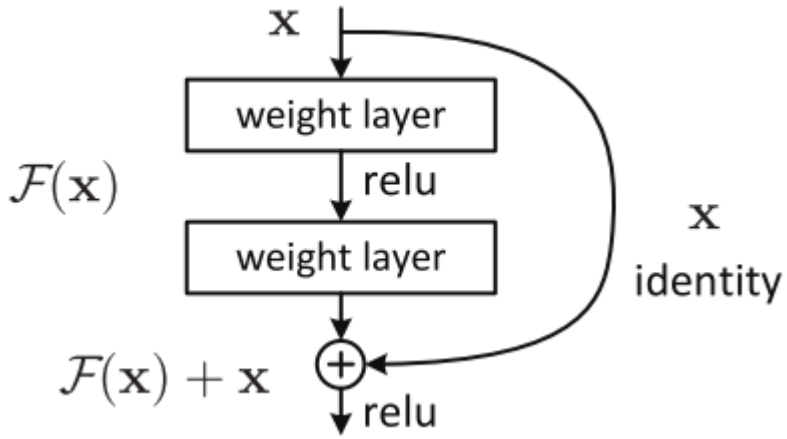

Fig 3.11 A residual block

The authors of [2] argue that stacking layers shouldn't degrade the network performance, because we could simply stack identity mappings (layer that doesn't do anything) upon the current network, and the resulting architecture would perform the same.

This indicates that the deeper model should not produce a training error higher than its shallower counterparts. They hypothesize that letting the stacked layers fit a residual mapping is easier than letting them directly fit the desired underlaying mapping. And the residual block above explicitly allows it to do precisely that.

As a matter of fact, ResNet was not the first to make use of shortcut connections, Highway Network introduced gated shortcut connections. These parameterized gates control how much information is allowed to flow across the shortcut. Similar idea can be found in the Long-Term Short Memory (LSTM) cell, in which there is a parameterized forget gate that controls how much information will flow to the next time step. Therefore, ResNet can be thought of as a special case of Highway Network.

However, experiments show that Highway Network performs no better than ResNet, which is kind of strange because the solution space of Highway Network contains ResNet, therefore it should perform at least as good as ResNet.

This suggests that it is more important to keep these "gradient highways" clear than to go for larger solution space.

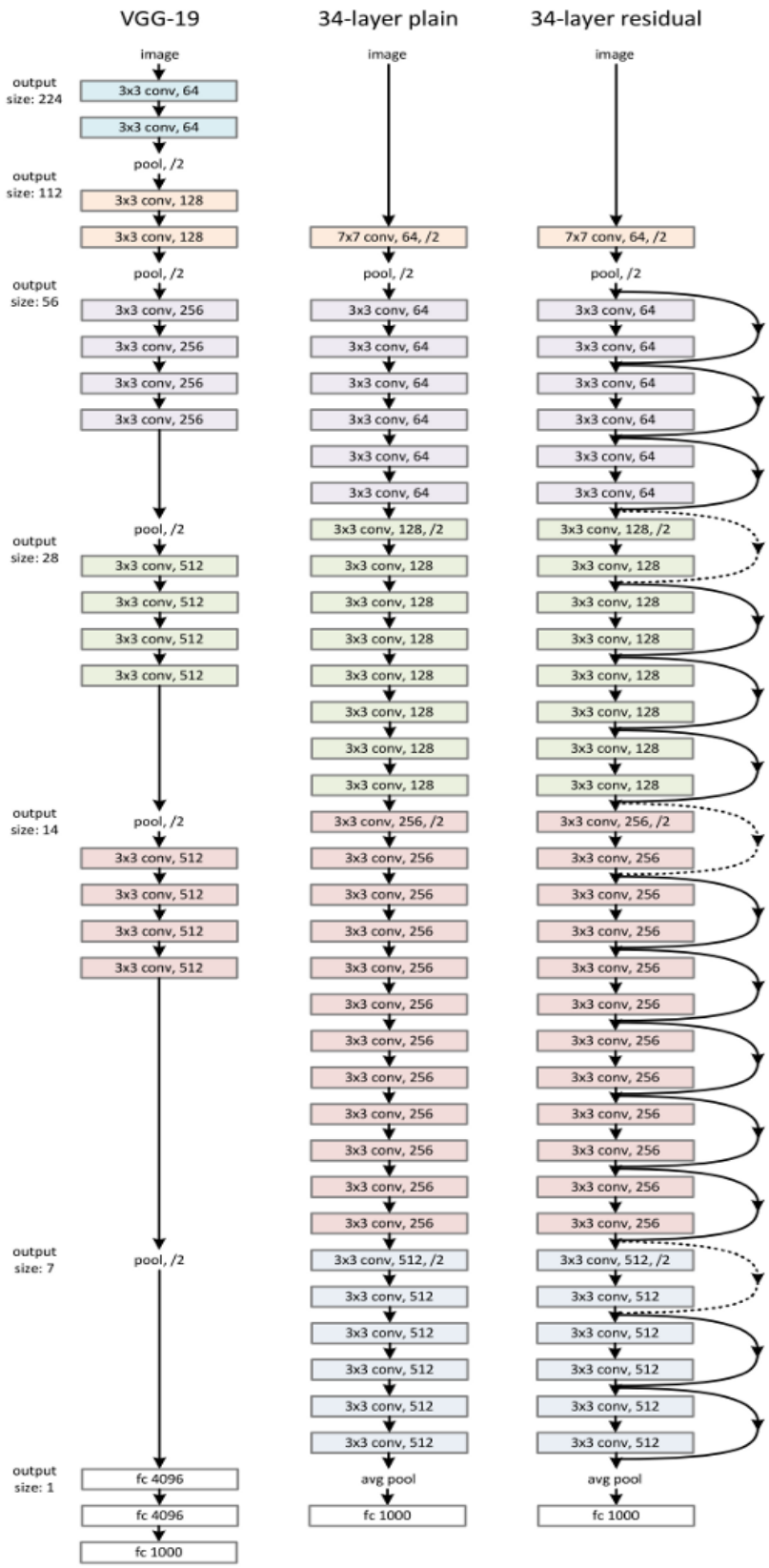

Fig 3.12 The ResNet architecture [5]

Following this intuition, the authors of [2] refined the residual block and proposed a pre-activation variant of residual block, in which the gradients can flow through the shortcut

connections to any other earlier layer unimpededly. In fact, using the original residual block in [2], training a 1202-layer ResNet resulted in worse performance than its 110-layer counterpart.

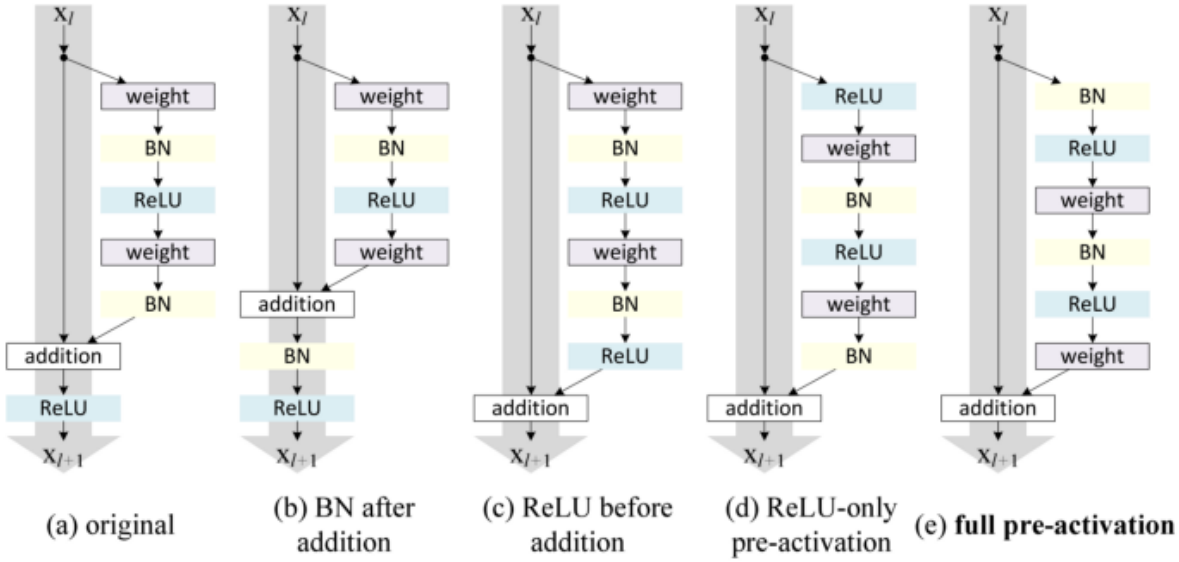

Fig 3.13 Variants of residual blocks [5]

It was demonstrated with experiments that they can now train a 1001-layer deep ResNet to outperform its shallower counterparts. Because of its compelling results, ResNet quickly became one of the most popular architectures in various computer vision tasks.

3.6 Recent Variants and Interpretations of ResNet

As ResNet gains more and more popularity in the research community, its architecture is getting studied heavily. In this section, I will first introduce several new architectures based on ResNet, then introduce a paper that provides an interpretation of treating ResNet as an ensemble of many smaller networks.

3.6.1 ResNeXt

Xie et al. [8] proposed a variant of ResNet that is codenamed ResNeXt with the following building block:

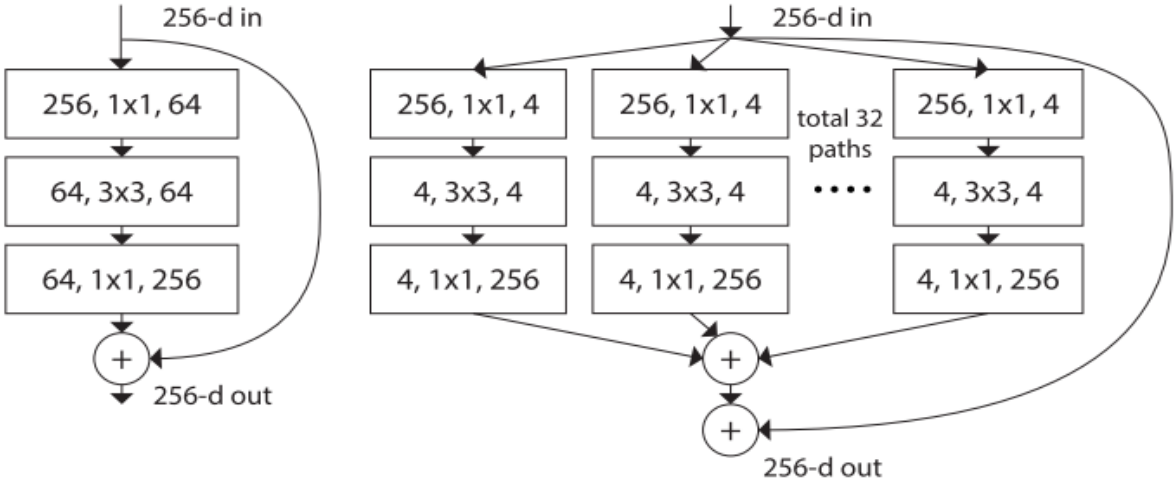

Fig 3.14 left: a building block of resnet, right: a building block of ResNeXt with cardinality = 32

This may look familiar as it is very similar to the Inception module of, they both follow the split-transform-merge paradigm, except in this variant, the outputs of different paths are merged by adding them together, while in they are depth-concatenated. Another difference is that in, each path is different (1x1, 3x3 and 5x5 convolution) from each other, while in this architecture, all paths share the same topology.

A hyper-parameter called cardinality was introduced — the number of independent paths, to provide a new way of adjusting the model capacity. Experiments show that accuracy can be gained more efficiently by increasing the cardinality than by going deeper or wider. It was stated that compared to Inception, this novel architecture is easier to adapt to new datasets/tasks, as it has a simple paradigm and only one hyper-parameter to be adjusted, while Inception has many hyper-parameters (like the kernel size of the convolutional layer of each path) to tune.

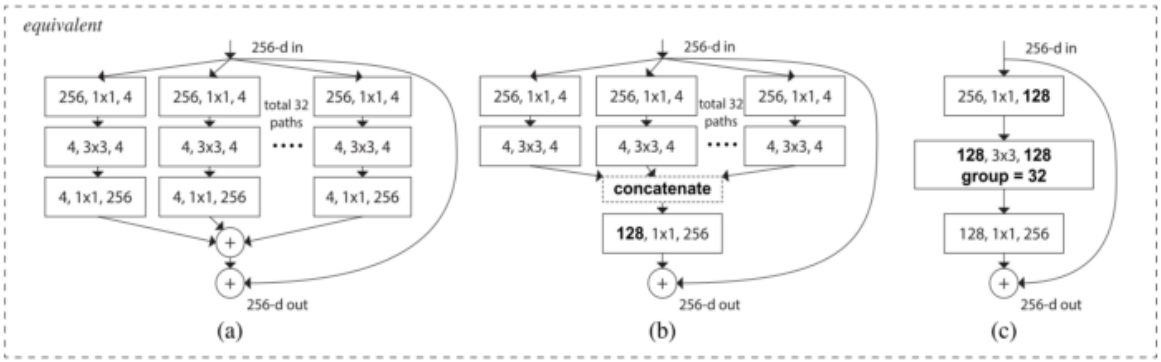

Fig 3.15 Grouped Model [5]

In practice, the “split-transform-merge” is usually done by pointwise grouped convolutional

layer, which divides its input into groups of feature maps and perform normal convolution respectively, their outputs are depth-concatenated and then fed to a 1x1 convolutional layer.

3.6.2 Densely Connected CNN

Huang et al. [9] proposed a novel architecture called DenseNet that further exploits the effects of shortcut connections — it connects all layers directly with each other. In this novel architecture, the input of each layer consists of the feature maps of all earlier layer, and its output is passed to each subsequent layer. The feature maps are aggregated with depth-concatenation.

Other than tackling the vanishing gradients problem, it was argued that this architecture also encourages feature reuse, making the network highly parameter-efficient. One simple interpretation of this is that, the output of the identity mapping was added to the next block, which might impede information flow if the feature maps of two layers have very different distributions. Therefore, concatenating feature maps can preserve them all and increase the variance of the outputs, encouraging feature reuse.

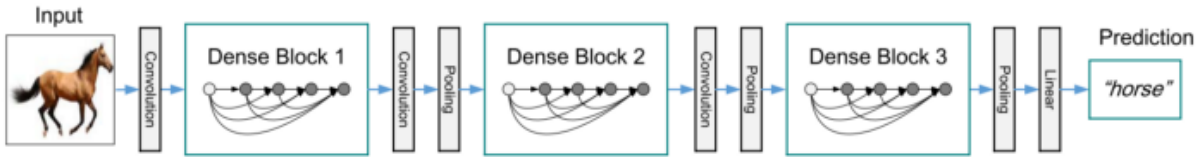

Fig 3.16 Prediction Model [6]

Following this paradigm, we know that the *l_th* layer will have *k * (l-1) + k_0* input feature maps, where *k_0* is the number of channels in the input image.

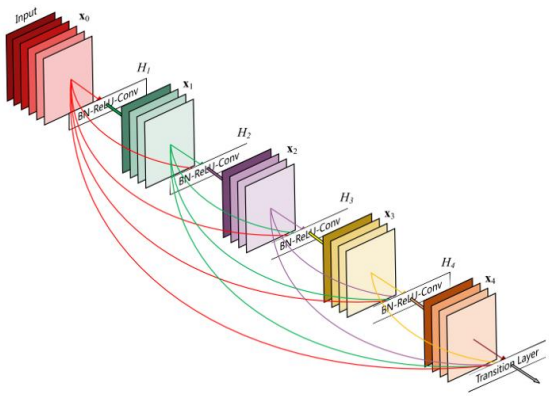

Fig 3.17 Growth Stages [6]

A hyper-parameter called growth rate (k) is used to prevent the network from growing too wide, they also used a 1x1 convolutional bottleneck layer to reduce the number of feature maps before the expensive 3x3 convolution. The overall architecture is shown in the below table:

Layers	Output Size	DenseNet-121($k = 32$)	DenseNet-169($k = 32$)	DenseNet-201($k = 32$)	DenseNet-161($k = 48$)
Convolution	112 × 112	7 × 7 conv, stride 2			
Pooling	56 × 56	3 × 3 max pool, stride 2			
Dense Block (1)	56 × 56	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6	[1 × 1 conv, 3 × 3 conv] × 6
Transition Layer (1)	56 × 56	1 × 1 conv			
	28 × 28	2 × 2 average pool, stride 2			
Dense Block (2)	28 × 28	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12	[1 × 1 conv, 3 × 3 conv] × 12
Transition Layer (2)	28 × 28	1 × 1 conv			
	14 × 14	2 × 2 average pool, stride 2			
Dense Block (3)	14 × 14	[1 × 1 conv, 3 × 3 conv] × 24	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 48	[1 × 1 conv, 3 × 3 conv] × 36
Transition Layer (3)	14 × 14	1 × 1 conv			
	7 × 7	2 × 2 average pool, stride 2			
Dense Block (4)	7 × 7	[1 × 1 conv, 3 × 3 conv] × 16	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 32	[1 × 1 conv, 3 × 3 conv] × 24
Classification Layer	1 × 1	7 × 7 global average pool			
		1000D fully-connected, softmax			

Fig 3.18 DenseNet architectures for ImageNet [7]

3.7 Deep Network with Stochastic Depth

Although ResNet has proven powerful in many applications, one major drawback is that deeper network usually requires weeks for training, making it practically infeasible in real-world applications. To tackle this issue, a counter-intuitive method of randomly dropping layers during training and using the full network in testing.

The residual block was used as the network's building block, therefore, during training, when a particular residual block is enabling, its input flows through both the identity shortcut and the weight layers, otherwise the input only flows only through the identity shortcut. In training time, each layer has a "survival probability" and is randomly dropped. In testing time, all blocks are kept active and re-calibrated according to its survival probability during training.

Formally, let H_l be the output of the l_th residual block, f_l be the mapping defined by the l_th block's weighted mapping, b_l be a Bernoulli random variable that be only 1 or 0 (indicating whether a block is active), during training:

$$H_l = ReLU(b_l * f_l(H_{l-1}) + id(H_{l-1})).$$

When *b_l* = 1, this block becomes a normal residual block, and when *b_l* = 0, the above formula becomes:

$$H_l = ReLU(id(H_{l-1})).$$

Since we know that *H_(l-1)* is the output of a ReLU, which is already non-negative, the above equation reduces to a identity layer that only passes the input through to the next layer:

$$H_l = id(H_{l-1}).$$

Let *p_l* be the survival probability of layer *l* during training, during test time, we have:

$$H_l = ReLU(p_l * f_l(H_{l-1}) + id(H_{l-1})).$$

A linear decay rule was applied to the survival probability of each layer, they argue that since earlier layers extract low-level features that will be used by later ones, they should not be dropped too frequently, the resulting rule therefore becomes:

$$p_l = 1 - \frac{l}{L}(1 - p_L).$$

Where *L* denotes the total number of blocks, thus *p_L* is the survival probability of the last residual block and is fixed to 0.5 throughout experiments. Also note that in this setting, the input is treated as the first layer (*l = 0*) and thus never dropped.

The overall framework over stochastic depth training is demonstrated in the figure below.

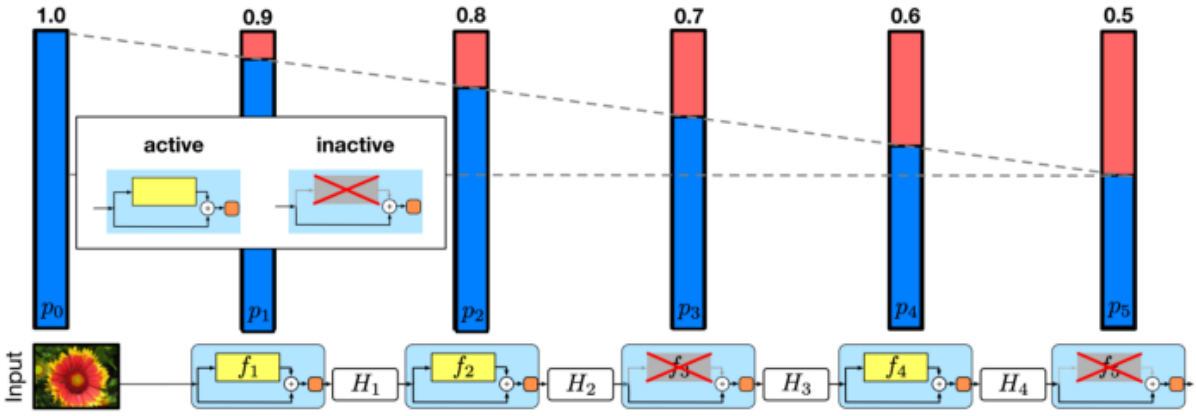

Fig 3.19 During training, each layer has a probability of being disabled [8]

Like Dropout, training a deep network with stochastic depth can be viewed as training an ensemble of many smaller ResNets. The difference is that this method randomly drops an entire layer while Dropout only drops part of the hidden units in one layer during training.

Experiments show that training a 110-layer ResNet with stochastic depth results in better performance than training a constant-depth 110-layer ResNet, while reduces the training time dramatically. This suggests that some of the layers (paths) in ResNet might be redundant.

3.6.3 ResNet as an Ensemble of Smaller Networks

[10] proposed a counter-intuitive way of training a very deep network by randomly dropping its layers during training and using the full network in testing time. Veit et al. [14] had an even more counter-intuitive finding: we can actually drop some of the layers of a trained ResNet and still have comparable performance. This makes the ResNet architecture even more interesting, also dropped layers of a VGG network and degraded its performance dramatically.

An unraveled view of ResNet makes things clearer. After we unroll the network architecture, it is quite clear that a ResNet architecture with *i* residual blocks has 2 ** i different paths (because each residual block provides two independent paths).

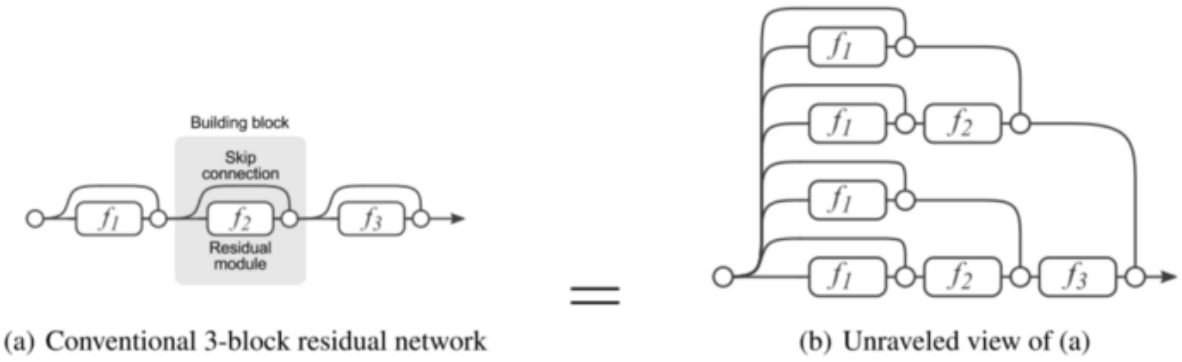

Fig 3.20 Resnet Block View [8]

Given the above finding, it is quite clear why removing a couple of layers in a ResNet architecture doesn't compromise its performance too much — the architecture has many independent effective paths and the majority of them remain intact after we remove a couple of layers. On the contrary, the VGG network has only one effective path, so removing a single layer compromises this one the only path.

Experiments were conducted to show that the collection of paths in ResNet have ensemble-like behaviour. They do so by deleting different number of layers at test time and see if the performance of the network smoothly correlates with the number of deleted layers. The results suggest that the network indeed behaves like ensemble, as shown in the below figure:

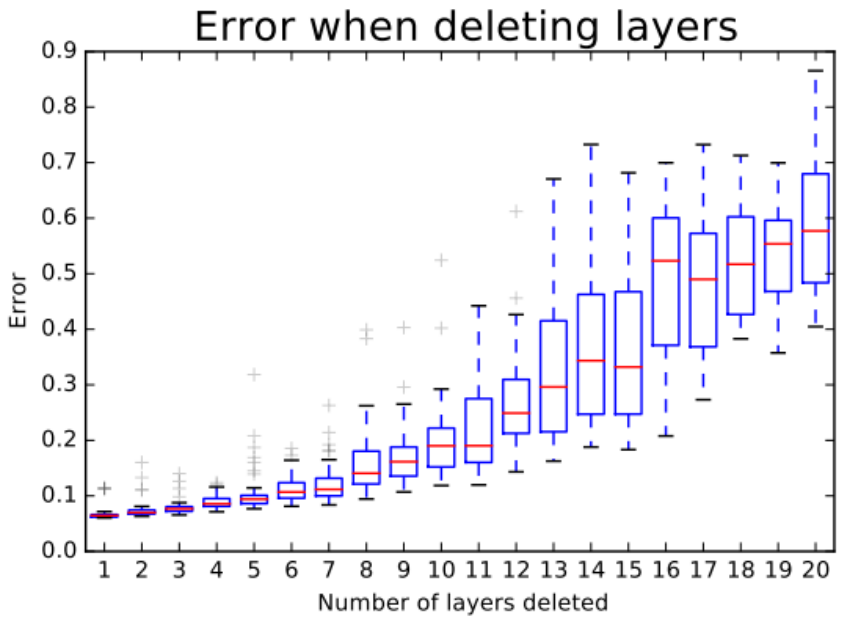

Fig 3.21 error increases smoothly as the number of deleted layers increases [9]

Finally, the characteristics of the paths in ResNet were looked into: It is apparent that the distribution of all possible path lengths follows a Binomial distribution, as shown in (a) of the blow figure. Most paths go through 19 to 35 residual blocks.

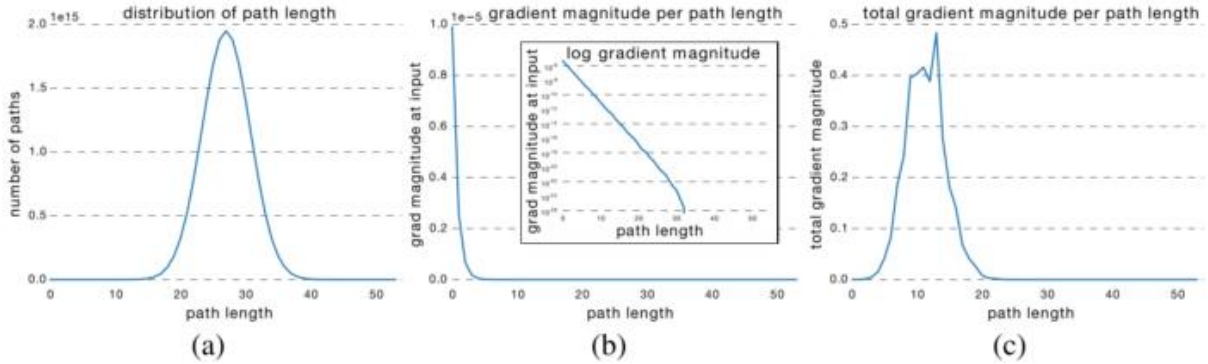

Fig 3.22 Path Length Graphs [10]

To investigate the relationship between path length and the magnitude of the gradients flowing through it. To get the magnitude of gradients in the path of length *k*, a batch of data was first fed to the network and then, and randomly sample *k* residual blocks. When back propagating the gradients, they propagated through the weight layer only for the sampled residual blocks. (b) shows that the magnitude of gradients decreases rapidly as the path becomes longer.

We can now multiply the frequency of each path length with its expected magnitude of gradients to have a feel of how much paths of each length contribute to training, as in (c). Surprisingly, most contributions come from paths of length 9 to 18, but they constitute only a tiny portion of the total paths, as in (a). This is a very interesting finding, as it suggests that ResNet did not solve the vanishing gradients problem for very long paths, and that ResNet enables training very deep network by shortening its effective paths.

Chapter 4
Generative Adversarial Network

4. 1 Generative Adversarial Network

Generative Adversarial Networks (GANs for short) have had a huge success since they were introduced in 2014 by Ian J. Goodfellow and co-authors in the article Generative Adversarial Nets.

Suppose that we are interested in generating black and white square images of dogs with a size of n by n pixels. We can reshape each data as a N=nxn dimensional vector (by stacking columns on top of each others) such that an image of dog can then be represented by a vector. However, it doesn't mean that all vectors represent a dog once shaped back to a square! So, we can say that the N dimensional vectors that effectively give something that look like a dog are distributed according to a very specific probability distribution over the entire N dimensional vector space (some points of that space are very likely to represent dogs whereas it is highly unlikely for some others). In the same spirit, there exists, over this N dimensional vector space, probability distributions for images of cats, birds and so on.

Then, the problem of generating a new image of dog is equivalent to the problem of generating a new vector following the "dog probability distribution" over the N dimensional vector space. So we are, in fact, facing a problem of generating a random variable with respect to a specific probability distribution.

At this point, we can mention two important things. First the "dog probability distribution" we mentioned is a very complex distribution over a very large space. Second, even if we can assume the existence of such underlying distribution (there actually exists images that looks like dog and others that doesn't) we obviously don't know how to express explicitly this distribution. Both previous points make the process of generating random variables from this distribution pretty difficult. Let's then try to tackle these two problems in the following.

Our first problem when trying to generate our new image of dog is that the "dog probability distribution" over the N dimensional vector space is a very complex one and we don't know how to directly generate complex random variables. However, as we know pretty well how to generate N uncorrelated uniform random variables, we could make use of the transform method. To do so, we need to express our N dimensional random variable as the result of a very complex function applied to a simple N dimensional random variable!

Here, we can emphasise the fact that finding the transform function is not as straightforward as just taking the closed-form inverse of the Cumulative Distribution Function (that we obviously don't know) as we have done when describing the inverse transform method. The transform function can't be explicitly expressed and, then, we have to learn it from data.
As most of the time in these cases, very complex function naturally implies neural network modelling. Then, the idea is to model the transform function by a neural network that takes as input a simple N dimensional uniform random variable and that returns as output another N dimensional random variable that should follow, after training, the the right "dog probability distribution". Once the architecture of the network has been designed, we still need to train it. In the next two sections, we will discuss two ways to train these generative networks, including the idea of adversarial training behind GANs!

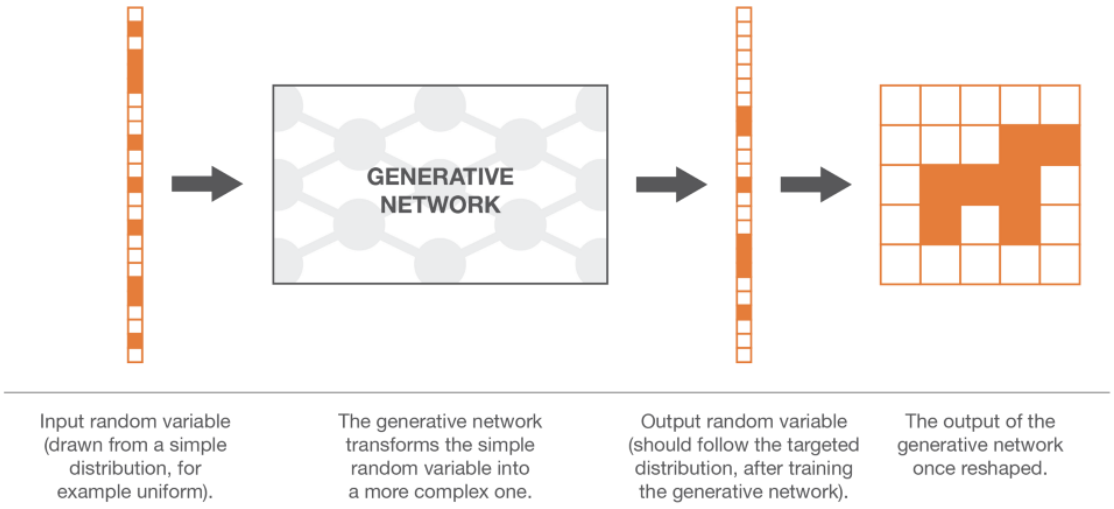

Fig 4.1 GAN structure

4.2 Generative vs. Discriminative Algorithms

To understand GANs, you should know how generative algorithms work, and for that, contrasting them with discriminative algorithms is instructive. Discriminative algorithms try to classify input data; that is, given the features of an instance of data, they predict a label or category to which that data belongs.
For example, given all the words in an email (the data instance), a discriminative algorithm could predict whether the message is spam or not_spam. spam is one of the labels, and the bag of words gathered from the email are the features that constitute the input data. When this problem is expressed mathematically, the label is called y and the features are called x.

The formulation p(y|x) is used to mean “the probability of y given x”, which in this case would translate to “the probability that an email is spam given the words it contains.”
So discriminative algorithms map features to labels. They are concerned solely with that correlation. One way to think about generative algorithms is that they do the opposite. Instead of predicting a label given certain features, they attempt to predict features given a certain label.
The question a generative algorithm tries to answer is: Assuming this email is spam, how likely are these features? While discriminative models care about the relation between y and x, generative models care about “how you get x.” They allow you to capture p(x|y), the probability of x given y, or the probability of features given a label or category. (That said, generative algorithms can also be used as classifiers. It just so happens that they can do more than categorize input data.)

Another way to think about it is to distinguish discriminative from generative like this:

- Discriminative models learn the boundary between classes
- Generative models model the distribution of individual classes

One neural network, called the *generator*, generates new data instances, while the other, the *discriminator*, evaluates them for authenticity; i.e. the discriminator decides whether each instance of data that it reviews belongs to the actual training dataset or not.
Let’s say we’re trying to do something more banal than mimic the Mona Lisa. We’re going to generate hand-written numerals like those found in the MNIST dataset, which is taken from the real world. The goal of the discriminator, when shown an instance from the true MNIST dataset, is to recognize those that are authentic.
Meanwhile, the generator is creating new, synthetic images that it passes to the discriminator. It does so in the hopes that they, too, will be deemed authentic, even though they are fake. The goal of the generator is to generate passable hand-written digits: to lie without being caught. The goal of the discriminator is to identify images coming from the generator as fake.

Here are the steps a GAN takes:

- The generator takes in random numbers and returns an image.
- This generated image is fed into the discriminator alongside a stream of images taken from the actual, ground-truth dataset.
- The discriminator takes in both real and fake images and returns probabilities, a number

between 0 and 1, with 1 representing a prediction of authenticity and 0 representing fake.

So, you have a double feedback loop:

- The discriminator is in a feedback loop with the ground truth of the images, which we know.
- The generator is in a feedback loop with the discriminator.

You can think of a GAN as the opposition of a counterfeiter and a cop in a game of cat and mouse, where the counterfeiter is learning to pass false notes, and the cop is learning to detect them. Both are dynamic; i.e. the cop is in training, too (to extend the analogy, maybe the central bank is flagging bills that slipped through), and each side comes to learn the other's methods in a constant escalation.

For MNIST, the discriminator network is a standard convolutional network that can categorize the images fed to it, a binomial classifier labeling images as real or fake. The generator is an inverse convolutional network, in a sense: While a standard convolutional classifier takes an image and downsamples it to produce a probability, the generator takes a vector of random noise and upsamples it to an image. The first throws away data through downsampling techniques like maxpooling, and the second generates new data.

Both nets are trying to optimize a different and opposing objective function, or loss function, in a zero-zum game. This is essentially an actor-critic model. As the discriminator changes its behavior, so does the generator, and vice versa. Their losses push against each other.

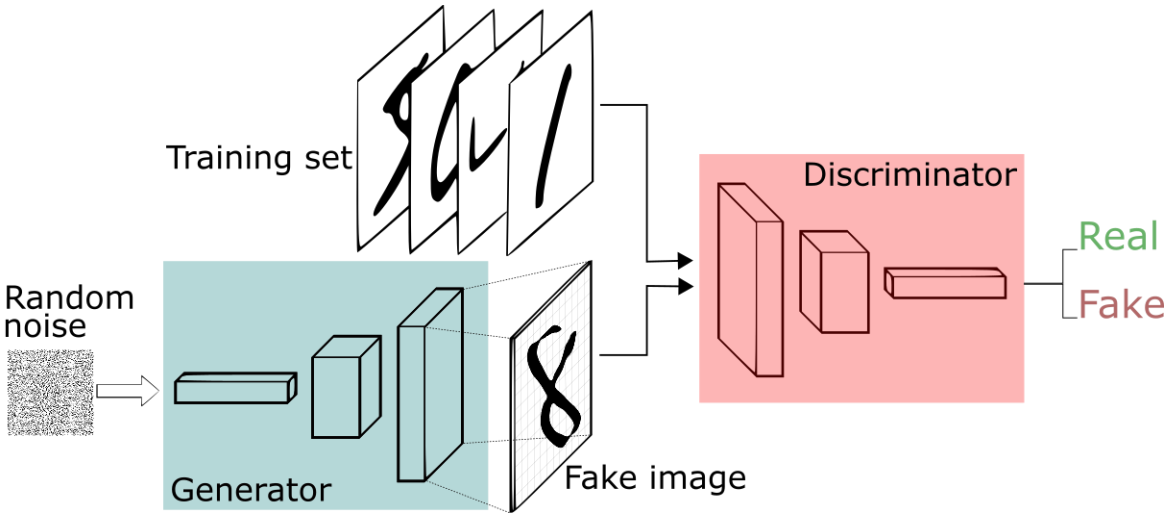

Fig 4.2 Functioning of a GAN

4.3 Cycle GAN

CycleGAN is a Generative Adversarial Network (GAN) that uses *two*generators and *two* discriminators

We call one generator *G*, and have it converted images from the *X* domain to the *Y* domain. The other generator is called *F*, and converts images from *Y* to *X*.

$$G : X \rightarrow Y$$
$$F : Y \rightarrow X$$

Both G and F are generators that take an image from one domain and translate it to another. G maps from X to Y, whereas F goes in the opposite direction, mapping Y to X.

Each generator has a corresponding discriminator, which attempts to tell apart its synthesized images from real ones.

$$D_y : \text{Distinguishes y from G(x)}$$
$$D_x : \text{Distinguishes x from F(y)}$$

One discriminator provides adversarial training for G, and the other does the same for F.

4.3.1 The Objective Function

There are two components to the CycleGAN objective function, an *adversarial loss* and a *cycle consistency loss*. Both are essential to getting good results.

If you are familiar with GANs, the adversarial loss should come as no surprise. Both generators are attempting to "fool" their corresponding discriminator into being less able to distinguish their generated images from the real versions.

$$Loss_{adv}(G, D_y, X) = \frac{1}{m}\sum_{i=1}^{m}(1 - D_y(G(x_i)))^2$$

$$Loss_{adv}(F, D_x, Y) = \frac{1}{m}\sum_{i=1}^{m}(1 - D_x(F(y_i)))^2$$

However, the adversarial loss alone is not sufficient to produce good images, as it leaves the model *under-constrained*. It enforces that the generated output be of the appropriate domain but does *not* enforce that the input and output are recognizably the same. For example, a generator that output an image *y* that was an excellent example of that domain, but looked

nothing like *x*, would do well by the standard of the adversarial loss, despite not giving us what we really want.

The *cycle consistency loss* addresses this issue. It relies on the expectation that if you convert an image to the other domain and back again, by successively feeding it through both generators, you should get back something similar to what you put in. It enforces that $F(G(x)) \approx x$ and $G(F(y)) \approx y$.

$$Loss_{cyc}(G, F, X, Y) = \frac{1}{m}\sum_{i=1}^{m}[F(G(x_i)) - x_i] + [G(F(y_i)) - y_i]$$

We can create the full objective function by putting these loss terms together, and weighting the cycle consistency loss by a hyperparameter λ. We suggest setting λ = 10.

$$Loss_{full} = Loss_{adv} + \lambda Loss_{cyc}$$

4.4 Cycle GAN architecture

The CycleGAN works without paired examples of transformation from source to target domain. Recent methods such as Pix2Pix depend on the availability of training examples where the same data is available in both domains. The power of CycleGAN lies in being able to learn such transformations without one-to-one mapping between training data in source and target domains. The need for a paired image in the target domain is eliminated by making a two-step transformation of source domain image - first by trying to map it to target domain and then back to the original image. Mapping the image to target domain is done using a generator network and the quality of this generated image is improved by pitching the generator against a discriminator (as described below)

We have a generator network and discriminator network playing against each other. The generator tries to produce samples from the desired distribution and the discriminator tries to predict if the sample is from the actual distribution or produced by the generator. The generator and discriminator are trained jointly. The effect this has is that eventually the generator learns to approximate the underlying distribution completely and the discriminator is left guessing randomly.

The above adversarial method of training has a problem though. Quoting the authors of the

original paper:

Adversarial training can, in theory, learn mappings GG and FF that produce outputs identically distributed as target domains YY and XX respectively. However, with large enough capacity, a network can map the same set of input images to any random permutation of images in the target domain, where any of the learned mappings can induce an output distribution that matches the target distribution. Thus, an adversarial loss alone cannot guarantee that the learned function can map an individual input xixi to a desired output yiyi.

To regularize the model, the authors introduce the constraint of cycle-consistency - if we transform from source distribution to target and then back again to source distribution, we should get samples from our source distribution.

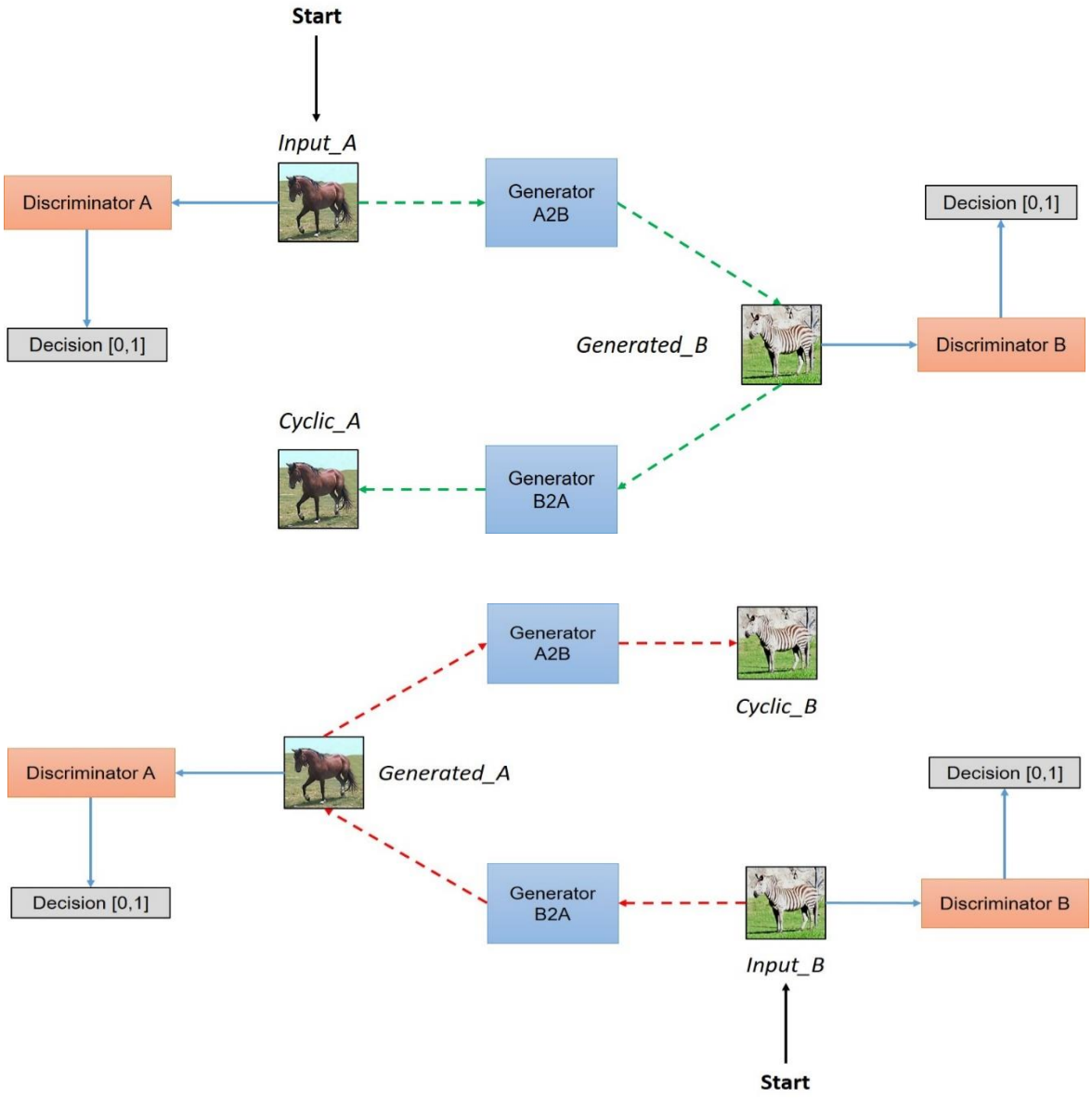

Fig 4.3 Simplified view of CycleGAN architecture

In a paired dataset, every image, say imgAimgA, is manually mapped to some image, say imgBimgB, in target domain, such that they share various features. Features that can be used to map an image (imgA/imgB)(imgA/imgB) to its correspondingly mapped counterpart (imgB/imgA)(imgB/imgA). Basically, pairing is done to make input and output share some common features. This mapping defines meaningful transformation of an image from one damain to another domain. So, when we have paired dataset, generator must take an input, say inputAinputA, from domain DADA and map this image to an output image, say genBgenB, which must be close to its mapped counterpart. But we don't have this luxury in unpaired dataset, there is no pre-defined meaningful transformation that we can learn, so, we will create it. We need to make sure that there is some meaningful relation between input image and generated image. So, authors tried to enforce this by saying that Generator will map input image (inputA)(inputA) from domain DADA to some image in target domain DBDB, but to make sure that there is meaningful relation between these images, they must share some feature, features that can be used to map this output image back to input image, so there must be another generator that must be able to map back this output image back to original input. So, you can see this condition defining a meaningful mapping between inputAinputA and genBgenB.

In a nutshell, the model works by taking an input image from domain DADA which is fed to our first generator GeneratorA→BGeneratorA→B whose job is to transform a given image from domain DADA to an image in target domain DBDB. This new generated image is then fed to another generator GeneratorB→AGeneratorB→A which converts it back into an image, CyclicACyclicA, from our original domain DADA (think of autoencoders, except that our latent space is DtDt). And as we discussed in above paragraph, this output image must be close to original input image to define a meaningful mapping that is absent in unpaired dataset.

As you can see in above figure, two inputs are fed into each discriminator(one is original image corresponding to that domain and other is the generated image via a generator) and the job of discriminator is to distinguish between them, so that discriminator is able to defy the adversary (in this case generator) and reject images generated by it. While the generator would like to make sure that these images get accepted by the discriminator, so it will try to generate images which are very close to original images in Class DBDB. (In fact, the generator and discriminator are actually playing a game whose Nash equilibrium is achieved when the generator's distribution becomes same as the desired distribution)

4.4.1 Generator Architecture

Each CycleGAN generator has three sections: an *encoder*, a *transformer*, and a *decoder*. The input image is fed directly into the encoder, which shrinks the representation size while increasing the number of channels. The encoder is composed of three convolution layers. The resulting activation is then passed to the transformer, a series of six residual blocks. It is then expanded again by the decoder, which uses two transpose convolutions to enlarge the representation size, and one output layer to produce the final image in RGB.

You can see the details in the figure below. Please note that each layer is followed by an instance normalization and a ReLU layer, but these have been omitted for simplicity.

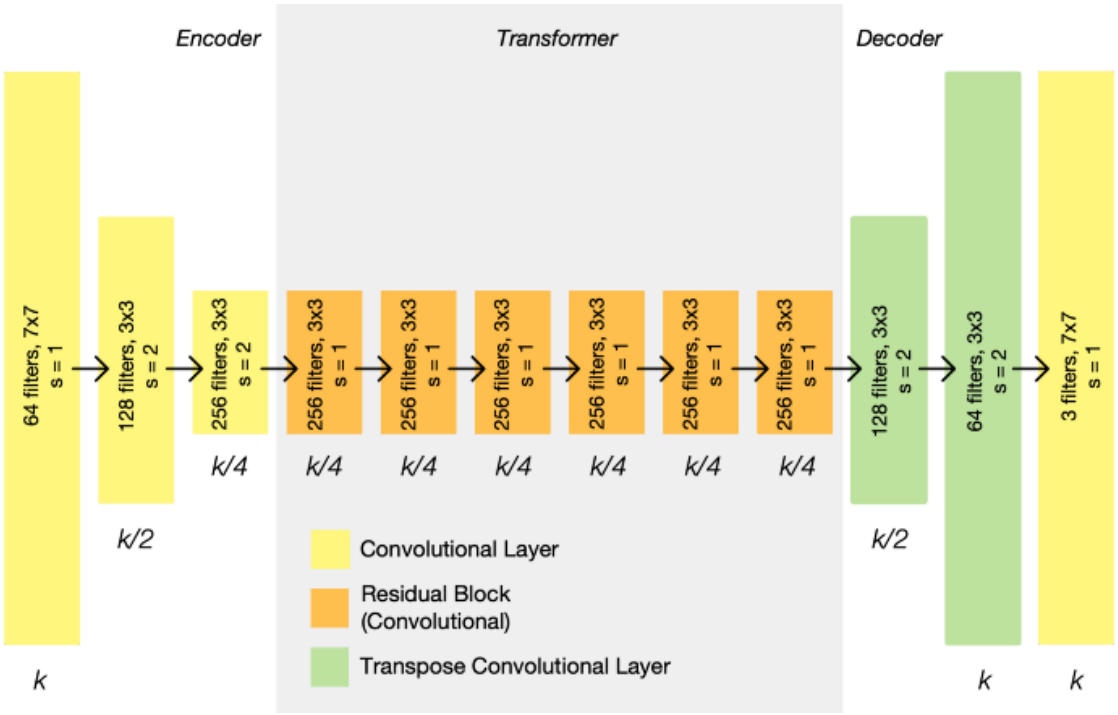

Fig 4.4 An architecture for a CycleGAN generator.

An architecture for a CycleGAN generator. As you can see above, the representation size shrinks in the encoder phase, stays constant in the transformer phase, and expands again in the decoder phase. The representation size that each layer outputs is listed below it, in terms of the input image size, k. On each layer is listed the number of filters, the size of those filters, and the stride. Each layer is followed by an instance normalization and ReLU activation.

Since the generators' architecture is fully convolutional, they can handle arbitrarily large input once trained.

4.4.2 Discriminator Architecture

The discriminators are PatchGANs, fully convolutional neural networks that look at a "patch" of the input image and output the probability of the patch being "real". This is both more computationally efficient than trying to look at the entire input image and is also more effective — it allows the discriminator to focus on more surface-level features, like texture, which is often the sort of thing being changed in an image translation task.

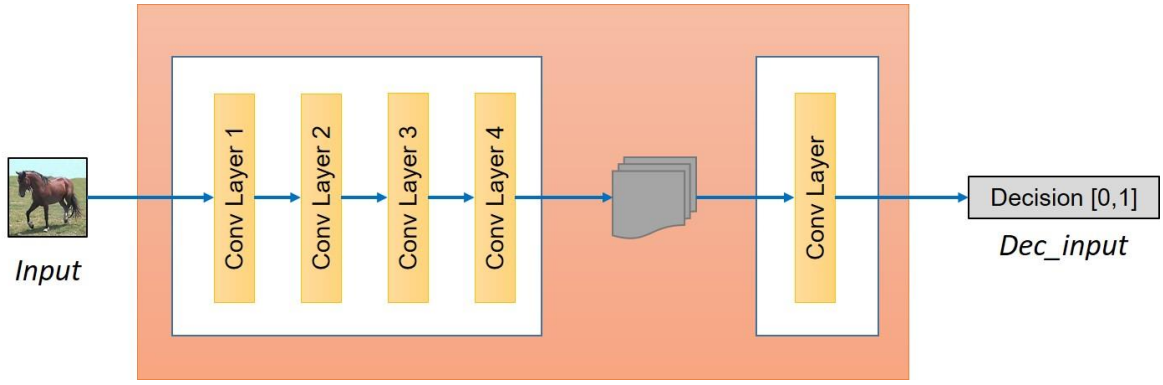

Fig 4.5 An architecture for a CycleGAN discriminator

CHAPTER 5
LIBRARIES USED

5.1 FastAI

The fastai library simplifies training fast and accurate neural nets using modern best practices. It's based on research in to deep learning best practices undertaken at fast.ai, including "out of the box" support for vision, text, tabular, and collab (collaborative filtering) models.

5.1.1 fastai vision

The vision module of the fastai library contains all the necessary functions to define a Dataset and train a model for computer vision tasks. It contains four different submodules to reach that goal:

- vision.image contains the basic definition of an Image object and all the functions that are used behind the scenes to apply transformations to such an object.
- vision.transform contains all the transforms we can use for data augmentation.
- vision.data contains the definition of ImageDataBunch as well as the utility function to easily build a DataBunch for Computer Vision problems.
- vision.learner lets you build and fine-tune models with a pretrained CNN backbone or train a randomly initialized model from scratch.

Each of the four-module links above includes a quick overview and examples of the functionality of that module, as well as complete API documentation. Below, we'll provide a walk-thru of end to end computer vision model training with the most commonly used functionality.

5.1.2 The data block API

The data block API lets you customize the creation of a DataBunch by isolating the underlying parts of that process in separate blocks.
Each of these may be addressed with a specific block designed for your unique setup. Your inputs might be in a folder, a csv file, or a dataframe. You may want to split them randomly, by certain indices or depending on the folder they are in. You can have your labels in your csv file or your dataframe, but it may come from folders or a specific function of the input.

You may choose to add data augmentation or not. A test set is optional too. Finally you have to set the arguments to put the data together in a DataBunch (batch size, collate function...)
The data block API is called as such because you can mix and match each one of those blocks with the others, allowing for a total flexibility to create your customized DataBunch for training, validation and testing. The factory methods of the various DataBunch are great for beginners but you can't always make your data fit in the tracks they require.

5.1.3 Training

The fastai library structures its training process around the Learner class, whose object binds together a PyTorch model, a dataset, an optimizer, and a loss function; the entire learner object then will allow us to launch training.
basic_train defines this Learner class, along with the wrapper around the PyTorch optimizer that the library uses. It defines the basic training loop that is used each time you call the fit method (or one of its variants) in fastai. This training loop is very bare-bones and has very few lines of codes; you can customize it by supplying an optional Callback argument to the fit method.
callback defines the Callback class and the CallbackHandler class that is responsible for the communication between the training loop and the Callback's methods. The CallbackHandler maintains a state dictionary able to provide each Callback object all the information of the training loop it belongs to, putting any imaginable tweaks of the training loop within your reach.
callbacks implements each predefined Callback class of the fastai library in a separate module. Some modules deal with scheduling the hyperparameters, like callbacks.one_cycle, callbacks.lr_finder and callback.general_sched. Others allow special kinds of training like callbacks.fp16 (mixed precision) and callbacks.rnn. The Recorder and callbacks.hooks are useful to save some internal data generated in the training loop.

CHAPTER 6
PROJECT DESCRIPTION

6.1 Dateset

The dataset used is the Yosemite summer2winter dataset which contains images of Yosemite National Park in both summer and winter. We have a total of 2744 images

6.2 Data Preprocessing

Below we enumerate the steps followed to prepare our data for the study.

6.2.1 Resizing of Images

In order to train our model effectively under hardware constraints of GPU memory and memory, we had to scale down our images to a uniform, smaller size of 64 x 64 pixels before we could use the images to train out model. This was done because pre-trained models tend to perform better on square images of similar sizes.

6.2.2. Date Augmentation

On application of the NN models, certain basic transformations were applied to the input dataset. These transformations include rotation, flipping, slight change in lighting and zooming. In our model, these transformations are applied to images with a probability of 75%. Augmentations help to prevent out models from being overfitted.

6.3 Implementation

We implemented CycleGAN using fastai . We used Resnet Blocks in our generator, and we created two resnet generators and two discriminators.

The main loss used to train the generators. It has three parts:

- the classic GAN loss: they must make the critics believe their images are real
- identity loss: if they are given an image from the set they are trying to imitate, they should return the same thing
- cycle loss: if an image from A goes through the generator that imitates B then through the generator that imitates A, it should be the same as the initial image. Same for B and switching the generators

6.3.1 Structure of Discriminator

Sequential(

(0): Conv2d(3, 64, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))

(1): LeakyReLU(negative_slope=0.2, inplace)

(2): Conv2d(64, 128, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))

(3): InstanceNorm2d(128, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)

(4): LeakyReLU(negative_slope=0.2, inplace)

(5): Conv2d(128, 256, kernel_size=(4, 4), stride=(2, 2), padding=(1, 1))

(6): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)

(7): LeakyReLU(negative_slope=0.2, inplace)

(8): Conv2d(256, 512, kernel_size=(4, 4), stride=(1, 1), padding=(1, 1))

(9): InstanceNorm2d(512, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)

(10): LeakyReLU(negative_slope=0.2, inplace)

(11): Conv2d(512, 1, kernel_size=(4, 4), stride=(1, 1), padding=(1, 1))

)

6.3.2 Structure of Generator

Sequential(

(0): ReflectionPad2d((3, 3, 3, 3))

(1): Conv2d(3, 64, kernel_size=(7, 7), stride=(1, 1))

(2): InstanceNorm2d(64, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)

(3): ReLU(inplace)

(4): Conv2d(64, 128, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))

(5): InstanceNorm2d(128, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)

(6): ReLU(inplace)

(7): Conv2d(128, 256, kernel_size=(3, 3), stride=(2, 2), padding=(1, 1))

(8): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False, track_running_stats=False)

(9): ReLU(inplace)

```
  (10): ResnetBlock(
    (conv_block): Sequential(
      (0): ReflectionPad2d((1, 1, 1, 1))
      (1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
      (2): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
      (3): ReLU(inplace)
      (4): ReflectionPad2d((1, 1, 1, 1))
      (5): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
      (6): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
    )
  )
  (11): ResnetBlock(
    (conv_block): Sequential(
      (0): ReflectionPad2d((1, 1, 1, 1))
      (1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
      (2): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
      (3): ReLU(inplace)
      (4): ReflectionPad2d((1, 1, 1, 1))
      (5): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
      (6): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
    )
  )
  (12): ResnetBlock(
    (conv_block): Sequential(
      (0): ReflectionPad2d((1, 1, 1, 1))
      (1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
      (2): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
      (3): ReLU(inplace)
      (4): ReflectionPad2d((1, 1, 1, 1))
```

```
        (5): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (6): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
      )
    )
    (13): ResnetBlock(
      (conv_block): Sequential(
        (0): ReflectionPad2d((1, 1, 1, 1))
        (1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (2): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
        (3): ReLU(inplace)
        (4): ReflectionPad2d((1, 1, 1, 1))
        (5): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (6): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
      ) )
    (14): ResnetBlock(
      (conv_block): Sequential(
        (0): ReflectionPad2d((1, 1, 1, 1))
        (1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (2): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
        (3): ReLU(inplace)
        (4): ReflectionPad2d((1, 1, 1, 1))
        (5): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (6): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
track_running_stats=False)
      ) )
    (15): ResnetBlock(
      (conv_block): Sequential(
        (0): ReflectionPad2d((1, 1, 1, 1))
        (1): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1))
        (2): InstanceNorm2d(256, eps=1e-05, momentum=0.1, affine=False,
```

6.6 Conclusion

We were able to see that our cycleGAN model was effectively able to convert summer images to winter and vice versa. Thus, cycleGAN can be used for converting one kind of object to another without pairing them, thus it’s much easier to construct a dataset for.

References:

Chapter 1

1. https://docs.python.org/3/
2. https://www.programiz.com/python-programming
3. https://www.datacamp.com/community/tutorials/data-structures-python
4. https://medium.com/the-renaissance-developer/python-101-the-basics-441136fb7cc3

Chapter 2

1. https://en.wikipedia.org/wiki/Artificial_neural_network#components_neurons/
2. https://medium.com/architecture-of-artificial-neural-networks/
3. https://en.wikipedia.org/wiki/truth-tables-in-neural-networks/524125/
4. Yung-Yao; Ming-Han "Design and Implementation of Artificial Networks".
5. Bethge, Matthias; Gatys, Leon A. "A Neural Algorithm of Artistic Style"
6. McCulloch, Warren; Walter Pitts (1943). "A Logical Calculus of Ideas Immanent in Nervous Activity". Bulletin of Mathematical Biophysics.
7. Hebb, Donald (1949). The Organization of Behavior. New York: Wiley.
8. Farley, B.G.; W.A. Clark (1954). "Simulation of Self-Organizing Systems by Digital Computer". IRE Transactions on Information Theory.
9. Rosenblatt, F. (1958). "The Perceptron: A Probabilistic Model For Information Storage And Organization In The Brain". Psychological Review.

Chapter 3

1. https://datafloq.com/read/machine-learning-explained-understanding-learning/4478
2. https://cdn.datafloq.com/cms/2018/01/23/supervised-learning.png
3. https://en.wikipedia.org/wiki/Supervised_learning
4. https://miro.medium.com/max/341/0*4YosVQ8oGBg6ZAWv

5. https://miro.medium.com/max/1200/0*ZaEKARNxNgB7-H3F

6. https://medium.com/machine-learning-for-humans/supervised-learning-740383a2feab

7. Ian J. Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, Yoshua Bengio "Generative Adversarial Networks", in NIPS 2014.

8. Alec Radford, Luke Metz and Soumith Chintala "Unsupervised Representation Learning with Deep Convolutional Generative Adversarial Networks", in ICLR 2016.

9. Jun-Yan Zhu, Philipp Krähenbühl, Eli Shechtman, and Alexei A. Efros. "Generative Visual Manipulation on the Natural Image Manifold", in ECCV 2016.

10. Phillip Isola, Jun-Yan Zhu, Tinghui Zhou, and Alexei A. Efros. "Image-to-Image Translation with Conditional Adversarial Networks", in CVPR 2017.

11. https://arxiv.org/pdf/1703.10593.pdf

Chapter 4

1. https://link.springer.com/article/10.1007/s13244-018-0639-9

2. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

3. https://adeshpande3.github.io/A-Beginner%27s-Guide-To-Understanding-Convolutional-Neural-Networks/

4. https://www.researchgate.net/publication/319253577_Understanding_of_a_Convolutional_Neural_Network

5. Carlos E. Perez. "A Pattern Language for Deep Learning".

6. "Regularization of Neural Networks using DropConnect | ICML 2013 | JMLR W&CP". jmlr.org. 2013-02-13. pp. 1058–1066. Retrieved 2015-12-17.

7. Zeiler, Matthew D.; Fergus, Rob (2013-01-15). "Deep Convolutional Neural Networks". arXiv:1301.3557 [cs.LG].

Chapter 5

1. Docs.fast.ai

2. PyTorch Open Documentation