

Master's Thesis

Implementation of an Approach for 3D Vehicle Detection in Monocular Traffic Surveillance Videos

Abhinav Mishra

January 09, 2021

Technische Universität Dresden
Institute for Transport Planning and Road Traffic
Chair Of Integrated Transport Planning And Traffic Engineering

First Reviewer: Prof.Dr.-Ing. Regine Gerike
Second Reviewer: Dipl.-Ing. Martin Bärwolff



Master's Thesis

Mr Abhinav Mishra

TASK

Topic: **"Implementation of an Approach for 3D Vehicle Detection in Monocular Traffic Surveillance Videos"**

INITIAL SITUATION

In recent years, considerable progress has been made in the field of computer vision, partly due to more powerful computer technology. This enables the recognition and tracking of vehicles, cyclists and pedestrians in monocular video sequences. Most of the approaches used do not detect the rather complex shape of an object, but the minimal surrounding rectangle (the so-called Bounding Box, "BB"). In most cases only the centre of the two dimensional BB (2D-BB) is considered for further traffic analysis. With this method, after the conversion of the detected image coordinates into world coordinates for photos and videos recorded from the air, there are only slight deviations between the centre of the actual ground plane of the traffic participants and the centre of the BB. However, recording road traffic from the air is time-limited, elaborate and expensive. Cameras installed locally at a height of approx. 3-6 metres (e. g. traffic surveillance cameras) can be used to record traffic videos permanently and cost-effectively. The flatter the camera's angle of view (in relation to the road surface), the greater the deviation between the centre of the BB (in world coordinates) and the centre of the road user's ground plane (up to 10 metres). This deviation depends, among other things, on the poses of the detected objects and their distance from the camera. This means that, for example, speeds of road users can only be calculated incorrectly and the definition of gates and areas for traffic counting is made more difficult. In-depth traffic analyses based on the exact locations, dimensions and poses of as well as distances between road users are not possible on the basis of the BB method.

There exist a number of alternative approaches to automatically detect information about the dimensions and poses of objects in monocular camera images (beyond 2D-BB). These approaches range from methods that directly output a 3D-BB (e.g. on the basis of known 3D vehicle models) to methods that detect so-called key features (e.g. taillight, mirror, wheel) as the first output, which are then connected to a wireframe model. The Key Feature method is also assumed to produce more stable

detection results in the case of partial occlusion of the detected objects. For some of the alternative approaches the information necessary for an own implementation (code, training data for neural network) is freely available.

OBJECTIVE AND WORK STEPS

The aim of this thesis is the implementation and evaluation of an approach for the automated detection of the dimensions and poses of vehicles (3D detection) in images from oblique monocular cameras. For this purpose, corresponding approaches are to be researched. At least one suitable approach is to be implemented and applied to camera images already available at the supervising chair. If possible, the implementation should be connectable to the system currently developed at the supervising chair. The achieved accuracy and the required resources of the approach should be evaluated. Problems in the implementation and application of an approach should be documented and solved if possible. In case of unsolvable problems, other approaches should be used in consultation with the supervisors. If no approach can be implemented or applied, the work should result in a comprehensive documentation of the problems and the attempts to solve them for different approaches.

In detail, the following work steps are to be carried out:

- International literature research on methods and corresponding data sets for automated detection of the dimensions and poses of vehicles in oblique recorded, monocular camera images (3D-BB methods and key feature/wireframe methods); documentation of code and data accessibility; ranking according to suitability for implementation
- Implementation of at least one suitable approach, commenting on own source code for subsequent use by programmers (e.g. in accordance with the PEP 8 Style Guide for Python Code); documentation in the form of short descriptions of all functions for potential users; documentation of problems that have occurred and attempts made to solve them
- Application and evaluation of at least one implemented approach to camera images available at the supervising professorship; assessment of the achieved accuracy and the required resources of the approach; preparation of a manual for potential users

RESULTS

As results of this work, commented source code, documentation and evaluation according to the steps described above are expected.

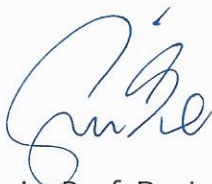
The Master's thesis must be submitted to the responsible examination office in due time in the form of two typewritten and bound copies as well as in digital text form on a suitable data carrier (typically on CD or DVD-ROM, a USB memory is not accepted). The examination regulations of the study programme and the corresponding guidelines for the submission of the Master's thesis must be followed.

Responsible university teacher:

Univ.-Prof. Dr.-Ing. Regine Gerike

Supervisor:

Dipl.-Ing. Martin Bärwolff

A handwritten signature in blue ink, appearing to read 'Gerike', with a stylized flourish above it.

Univ.-Prof. Dr.-Ing. R. Gerike

Declaration

I hereby declare that I have created this work myself and have not used any aids other than those specified.

Dresden, January 09, 2021

Abhinav Mishra

Acknowledgement

This thesis has seen two Corona virus induced lockdowns and as a result was conceived in many places and in many parts depending on what spaces were accessible. It journeyed through multiple library spaces (available through online reservations only), through spaces in regional trains connecting Leipzig and Dresden, through the intermittently accessible (owing to lockdowns) space at the supervising chair and finally came back to its space of origin, my home in Leipzig. My journey, like this thesis, also grazed through many spaces, with each space transforming me into a more informed individual. Writing this thesis enabled me to visualize mathematics and empowered me to observe the reality it creates. With this heightened sense of observation, I also observed the support of every individual who has lend a helping hand through this journey. Amongst all those individuals, I would first like to thank my thesis supervisor Martin Bärwolff for setting the foundations of this thesis and to share the same zeal of seeing things in action like me. Apart from all his support and odd hour discussions, I specifically thank Martin for giving me the space to be. I would also like to thank Prof. Regine Gerike who approved and supervised this thesis and shared our zeal and, to Prof. Alexander Schill for accepting this thesis in the DSE curriculum.

I will be forever indebted to both my parents and parents-in-law in India for their unconditional support throughout my Master's journey. I thank all my friends in DSE for the fun times. And finally, I thank my partner and light, Garima, firstly for her inputs on my writing and thereby preventing any academic crimes that I might have had committed in the process and secondly, for being the being she is.

Abstract

Recent advancements in the field of Computer Vision are a by-product of breakthroughs in the domain of Artificial Intelligence. Object detection in monocular images is now realized by an amalgamation of Computer Vision and Deep Learning. While most approaches detect objects as a mere two dimensional (2D) bounding box, there are a few that exploit rather traditional representation of the 3D object. Such approaches detect an object either as a 3D bounding box or exploit its shape primitives using active shape models which results in a wireframe-like detection. Such a wireframe detection is represented as combinations of detected keypoints (or landmarks) of the desired object. Apart from a faithful retrieval of the object's true shape, wireframe based approaches are relatively robust in handling occlusions. The central task of this thesis was to find such an approach and to implement it with the goal of its performance evaluation. The object of interest is the vehicle class (cars, mini vans, trucks etc.) and the evaluation data is monocular traffic surveillance videos collected by the supervising chair. A wireframe type detection can aid several facets of traffic analysis by improved (compared to 2D bounding box) estimation of the detected object's ground plane. The thesis encompasses the process of implementation of the chosen approach called Occlusion-Net [40], including its design details and a qualitative evaluation on traffic surveillance videos. The implementation reproduces most of the published results across several occlusion categories except the truncated car category. Occlusion-Net's erratic detections are mostly caused by incorrect detection of the initial region of interest. It employs three instances of Graph Neural Networks for occlusion reasoning and localization. The thesis also provides a didactic introduction to the field of Machine and Deep Learning including intuitions of mathematical concepts required to understand the two disciplines and the implemented approach.

Contents

1	Introduction	1
2	Technical Background	7
2.1	AI, Machine Learning and Deep Learning	7
2.1.1	But what is AI ?	7
2.1.2	Representational composition by Deep Learning	10
2.2	Essential Mathematics for ML	14
2.2.1	Linear Algebra	15
2.2.2	Probability and Statistics	25
2.2.3	Calculus	34
2.3	Mathematical Introduction to ML	39
2.3.1	Ingredients of a Machine Learning Problem	39
2.3.2	The Perceptron	40
2.3.3	Feature Transformation	46
2.3.4	Logistic Regression	48
2.3.5	Artificial Neural Networks: ANN	53
2.3.6	Convolutional Neural Network: CNN	61
2.3.7	Graph Neural Networks	68
2.4	Specific Topics in Computer Vision	72
2.5	Previous work	76
3	Design of Implemented Approach	81
3.1	Training Dataset	81

3.2	Keypoint Detection : MaskRCNN	83
3.3	Occluded Edge Prediction : 2D-KGNN Encoder	84
3.4	Occluded Keypoint Localization : 2D-KGNN Decoder	86
3.5	3D Shape Estimation: 3D-KGNN Encoder	88
4	Implementation	93
4.1	Open-Source Tools and Libraries	93
4.1.1	Code Packaging: NVIDIA-Docker	94
4.1.2	Data Processing Libraries	94
4.1.3	Libraries for Neural Networks	95
4.1.4	Computer Vision Library	95
4.2	Dataset Acquisition and Training	96
4.2.1	Acquiring Dataset	96
4.2.2	Training Occlusion-Net	96
4.3	Refactoring	97
4.3.1	Error in Docker File	97
4.3.2	Image Directories as Input	97
4.3.3	Frame Extraction in Parallel.	98
4.3.4	Video as Input	100
4.4	Functional changes	100
4.4.1	Keypoints In Output	100
4.4.2	Mismatched BB and Keypoints	101
4.4.3	Incorrect Class Labels	101
4.4.4	Bounding Box Overlay	101
5	Evaluation	103
5.1	Qualitative Evaluation	103
5.1.1	Evaluation Across Occlusion Categories	103
5.1.2	Performance on Moderate and Heavy Vehicles:	105

5.2	Verification of Failure Analysis	106
5.2.1	Truncated Cars	107
5.2.2	Overlapping Cars	108
5.3	Analysis of Missing Frames	109
5.4	Test Performance	110
6	Conclusion	113
7	Future Work	117
	Bibliography	119

1 Introduction

Advances in modern day computing led to revival of a previously computationally-inhibited discipline of Artificial Intelligence. Graphics Processing Units or GPUs are the current state of the art and are optimized to perform vectorized arithmetic operations which are a key computation step in the field. The two most important facets of Artificial Intelligence (or AI) are Machine Learning and Deep Learning. Deep Learning is often introduced as Machine Learning with neural networks and is represented as a sub-field of Machine Learning. Despite being a sub-field, Deep Learning has gained significant traction due to its contributions in the field of Computer Vision. Computer vision attempts to assign semantic meaning to a given image. One way to develop a semantic understanding of an image is by identifying objects and their locations in the image. Such a task is categorized as the problem of object detection. Given an image which has a cat and a dog in it, an object detection algorithm outputs two labels 'cat' and 'dog' along with their corresponding location in the image. The most basic object detection algorithms output these locations as coordinates of a rectangular bounding box drawn around the object. This box is also known as the 2D bounding box. But there are advanced object detection algorithms that output detections adhering to the object's true shape constrained by its true 3D geometry. Such a detection is desired as it provides more room for semantic reasoning of a given image. Object detection finds numerous applications in self driving cars and surveillance video analysis.

This thesis explores object detection in the context of traffic surveillance videos. These surveillance videos are captured by the supervising chair of 'Integrated

Transport Planning And Traffic Engineering’ using pole-mount cameras. An accurate detection of vehicles in these surveillance videos can facilitate automated traffic analysis of the recorded location. Detections of vehicle objects can be used to approximate their speed, distance to other vehicles and orientation. These characteristics can then be fed to algorithms which develop understanding about traffic flow, speed and density. Such algorithms rely on accurate detection of the ground plane of the detected object because the centre of the ground plane is used to approximate object’s location with respect to other objects in a given scene. Distance between two objects can be estimated by comparing the distances between the centres of their ground plane. Such distances when calculated across several frames of a video can be used to build approximations about vehicles’ speed and their changing relative distances.

The thesis is titled ‘Implementation of an Approach for 3D Vehicle Detection in Monocular Traffic Surveillance Videos’. It is of importance to deconstruct the title in order to set the right expectation for both readers and evaluators. The first phrase ‘Implementation of an approach’ signifies that the thesis implements an existing approach and not proposes one. The selection of the implemented approach is a result of contemporary literature research in the field of deep learning based computer vision methods. Amongst many possible object detection techniques, the research was focused on the ones whose detections are a faithful representation of an object’s true shape. Since an image/video is a two dimensional representation of a 3D world, detecting a vehicle object’s true shape can be expressed as ‘3D Vehicle Detection’, thus making the second phrase in the title. The third phrase ‘Monocular Traffic Surveillance Videos’ indicates that the traffic surveillance videos are captured using a single camera which results in a monocular (analogous to looking from one eye) image/video of the surveilled scene.

The chosen approach is called Occlusion-Net [40]. Occlusion-Net detects vehicle objects using 3D wireframe models. Wireframe models are a popular way to repre-

sent rigid bodies such as cars. They derive their motivation from the seminal work of Active Shape Models [4] that represent a given object's shape as a statistical model. Any variation across several categories of the same object class is then captured by varying different dimensions of this statistical model. The term 'Occlusion' (hidden parts of an object) is a signifier of explicit occlusion handling in Occlusion-Net. Occlusion-Net uses two dimensional bounding box type detections from an established 2D object detector known as MaskRCNN [19]. MaskRCNN also detects the location of predefined keypoints (front-right wheel, back-left wheel, front-right light etc.) of car object. From the 2D bounding box and locations of these keypoints it is difficult to reason about 3D geometry and Occlusion status (occluded or not) of each keypoint. Therefore, Occlusion-Net employs three separate Graph Neural Networks (GNN). The first two GNNs identify and localize the occluded keypoints, the third one enforces a 3D symmetry on the 2D detection using wireframe models.

Chapter 2 attempts to build intuitions and concepts required for understanding the essence of Machine and Deep Learning and is highly recommended for the curious reader. Section 2.1.1 begins with explaining the fundamental differences between AI and other computer programming related fields. It establishes a notion of 'intelligence' and explains that why some machines (or programs they run) seem intelligent in the first look, but are in reality just a speedy execution of pre-defined hard coded rules. Section 2.1.2 provides a high level overview of how Deep Learning is realized and about its ability to work with raw data. This section also introduces the required vocabulary for further discussions.

The section on essential mathematics (2.2) attempts at establishing a correlation between the different disciplines of mathematics and Machine Learning. It discusses contributions from Linear Algebra (2.2.1), Probability & Statistics (2.2.2) and Calculus (2.2.3). Apart from providing a quick refresher of the fundamental concepts, this section also lists direct contributions from these disciplines

such as Principal Component Analysis (2.2.1), representation of features as vectors (2.2.1), Maximum Likelihood Estimation((2.2.2) and optimization techniques like gradient descent (2.2.3). Section 2.3 re-introduces Machine Learning and Deep Learning using the mathematical intuitions built in the previous sections. Historical learning algorithms like Perceptron 2.3.2 and Logistic Regression (2.3.4) are first presented individually and later as basis for modern day neural networks. The section 2.3.3 captures the essence of the term 'feature engineering' and presents a visual interpretation of the same. Section 2.3.5 describes the inner working and formulations of neural networks which are extended to section 2.3.6 in order to explain Convolutional Neural Networks which are specifically designed neural networks for solving image related task. Central to this work is the use of Graph Neural Network (2.3.7) which are a result of combining graph theory's algorithm of message passing and neural networks. Section 2.4 introduces the field of Computer Vision and familiarizes the reader with required geometrical intuitions and vocabulary surrounding different coordinate systems used in digital image formation. The chapter ends with a discussion on other approaches for object detection in section 2.5.

Chapter 3 gives an overview of the training dataset (section 3.1) and design choices (sections 3.2, 3.3, 3.4 and 3.5) made by the authors of Occlusion-Net.

Chapter 4 explains the code implementation of this approach along with the details of open-source libraries(4.1) for data processing (4.1.2) and Deep Learning (4.1.3). Details surrounding acquisition of training data(4.2.1)and trained weights of neural networks(4.2.2) along with usage of training script(4.2.2) are explained in section 4.2. Sections 4.4 and 4.3 detail both functional and non-functional code changes respectively. These changes were made to adapt the publicly available code for evaluation.

Chapter 5 presents the results of qualitative evaluation of the implemented approach on traffic surveillance videos. It starts with comparing the published re-

sults in original publications with the results reproduced during this thesis (section 5.1.1). Section 5.1.2 specifically evaluates Occlusion-Net’s performance on moderate and heavy vehicles. The authors of Occlusion-Net also published a failure analysis report [41], results of which are verified in section 5.2. During evaluation, it was observed that Occlusion-Net skips some frames (does not process them), section 5.3 is dedicated to discuss the reasons leading to skipping of frames followed by an analysis of skipped frames.

Chapter 6 summarizes the thesis and chapter 7 provides suggestions for possible future work on Occlusion-Net with respect to its inclusion in an object detection pipeline.

2 Technical Background

This chapter attempts at equipping the reader with the rudiments of Machine and Deep Learning along with introducing disciplines of mathematics that cross-cut this multidisciplinary field. The presented intuitions are required to set grounds on which the design and evaluation of the implemented approach are explained in later chapters. This chapter threads through these concepts systematically and ends with an introduction to computer vision where these concepts are applied, followed by an overview of both preceding and contemporary approaches to the evaluated approach.

2.1 AI, Machine Learning and Deep Learning

Before the discussion transitions into the intricacies of Machine and Deep Learning, it is essential to view them as united under the umbrella of Artificial Intelligence. The following section presents this unison and provides a framework of reasoning that can be applied to question intelligence of a given system that is claimed to be 'artificially intelligent'. The overall intuition and several anecdotal evidences are captured from the introduction chapter of Goodfellow et al.'s book titled Deep Learning [16].

2.1.1 But what is AI ?

Artificial Intelligence or AI could quite literally be understood as an intelligence that by its true nature is artificial, i.e. the presence of intelligence where it is

not supposed to be and therefore, must be 'artificially placed'. When the first programmable computer was conceived, computers were seen as machines that can perform fast computations and offer impeccable results every time. They were better than humans at problems that could be described using defined steps which are often mathematical in nature. For such problems, humans were definitely inferior to the computer's accuracy and speed.

But what about the intuitive intelligence of human beings? What about the virtue of vision? Much work has been done in understanding of the human visual cortex which is the part of the brain that processes visual information. Despite developing considerable understanding of the visual perception process, a larger part of it remains a mystery. Now, given the complexity of the visual process one can imagine the difficulty or rather the impossibility of breaking down the process in computer-understandable steps that are often mathematical in nature! The modern day artificial intelligence attempts to bridge this gap and tries to make computers more human-like by making them more intuitive.

Humans learn to distinguish colors and differentiate between different objects through the virtue of experience and constant learning. In order for computers to mimic humans, they must be provided with similar, if not the same types of experiences and learning. Humans intuitively use simple primitives for ex. shape, color and words etc. to derive complex concepts like dimensions (2D-3D), illuminance (bright or dark) and language respectively. Similarly, for a computer, complex concepts must be modelled as compositions of numerous simpler concepts. The computer then must be taught how to derive a complex concept using these simpler or low-level concepts.

A lot of early successes of the then perceived AI took place in closed rooms without needing the so called worldly knowledge and experience. Amongst the first successes of a computer beating a human at a thinking game was IBM's Deep Blue. Deep Blue was a chess playing computer that defeated the world champion. De-

spite the remarkable success of the computer program, one must critically analyze the 'artificially intelligent' part of it. Modelling a winning chess strategy is surely a non-trivial task but the game of chess has defined rules that can be summarized in few lines of code. The computer thus knows before hand, the rules of the game and then implements the remarkable winning strategy. But is there a learning experience here? Can one say that the computer learned through its experiences and became better over time? The right answer is 'No', this is because the computer was acting upon a formally described set of rules and therefore, will be superior to the fellow humans, including the world champions. But the same computer will fail at what the humans do almost automatically with utmost perfection, like recognizing a person by their voice, or to identify the boundary between a black cat sitting on a black couch in an image. This is what Goodfellow et al. in their book 'Deep Learning' [16] conclude as *"Ironically, abstract and formal tasks that are among the most difficult mental undertakings for a human being are among the easiest for a computer. Computers have long been able to defeat even the best human chess player, but are only recently matching some of the abilities of average human beings to recognize objects or speech."*

A human's knowledge of intuition and the intuition behind that knowledge is rather difficult to formalize. Formalizing this knowledge in order for it to be understood by a computer is one of the most prevalent challenges of AI. Some attempts were made to hard-code the world information but did not succeed. This indicates that our AI system must be able to generate subjective inferences by using raw data from the real world. This ability of inferring using non-explicitly specified rules is called **Machine Learning**. Raw data here refers to the indirectly-related features or simple concepts that define the object/situation on which the inference is being done. For e.g. an algorithm called Logistic Regression can be used to predict whether a Caesarean delivery should be recommended or not (excerpt from [16]). One of the raw feature fed to this algorithm is the presence/absence of

a uterine scar, to indicate if there was a previous delivery. Raw features like this do not directly relate to the output decision but provide their individual contributions in deciding the output. The final decision is then computed by considering contributions from many individual raw features. It is important to note that the success of the prediction algorithm is heavily dependent on the type of raw features that are being fed to it and how well these features are represented.

If we try to visualize these simple features as forming a base layer, and the layer above it is represented by more-complex features derived from these simple feature of the base layer and so on, then, by stacking many layers and thereby extracting more complex features, in the end we reach the object/situation we are trying to model. Thus, each layer transforms the features it extracted from the previous layer and creates a new representation of the features. This results in multiple layers stacked on one another creating a notion of **depth**, hence this approach of AI is called **Deep Learning**. Deep learning is a sub class of Machine Learning which in turn is a sub class of AI.

2.1.2 Representational composition by Deep Learning

The last section discussed how a Machine Learning algorithm is expected to use a representation of the raw data and learn to **map** this representation to a desired output. Output here could mean any desired inference that is expected from the model, like the presence of an object in an image or whether to recommend Cesarean delivery or not. But how are such representations realized? And what is a meaningful representation?

Meaning-fullness of a representation: The representation of the raw data is directly used by the model to generate the output inference. Therefore, the representation must be such that its value affects the outputs with some magnitude. A representation would consist of a few to many independent features that ideally should represent variation in data. These features are famously called 'Factors

Of Variation' (Chapter 1, Deep Learning [16]) and should be independent to each other. Independence here signifies two qualities; first, that each factor of variation should represent different orthogonal dimensions of the data and second, that it can change independently without affecting values of other factors. A branch of AI called **Representation Learning** (or feature learning) attempts at learning these factors of variation from the raw data. For e.g. such factors for a speech recognition algorithm could be the speaker's age, her accent and her vocal tract's size etc. But, these 'Factors of variations' are hard to comprehend from raw data. A common methodology of representation learning is to use **Autoencoders** for learning these factors of variation. An autoencoder consist of an **encoder** and a **decoder**. The encoder takes raw data as an input and derives a representation from it, the decoder takes this representation and converts it back to the original format (Chapter 1, Deep Learning [16]). The autoencoder tries to preserve as much information as it can about the raw data but at the same time attempts to learn interesting features about the data. But, identifying these factors from raw data is a non-trivial task and often can only be done by experts in the field. Thus one could argue that since representation learning attempts to learn complex representations which are difficult to identify just by looking at raw data, it does not solve the problem of Machine Learning but adds another level of indirection to it. But what if there was something more to look at than just RAW data? What if these complex representations can be learned from many, relatively easier to identify, and **simpler** representations?

Deep Learning solves this central problem of representation learning by introducing composability of representations i.e. complex representations can be represented as combinations of simpler ones. The previous section elaborated on why an AI algorithm needs to get a similar (if not same) human experience in order to derive intelligent and subjective inferences for a particular problem. Humans learn to differentiate between cats and dogs by sufficiently seeing many examples

of both the categories. The internals of how this differentiation is learned by a human brain is still a mystery, but one can at least provide a similar range of examples of each category to an AI algorithm in order for it to learn to differentiate between the two. The following section introduces the required vocabulary for future discussion about AI and deep learning.

An AI model (or algorithm) needs sufficient amount of data (for experience) to solve a given problem. This ample amount of data, this dataset, is known as the **Training Dataset**, and the process of teaching is referred to as **Training**. Like all other teachings, this one too has to be correctional in nature i.e. one should be able to correct the model's errors by empirically checking its outputs. The steps which are followed to train a model are collectively known as the **Learning Algorithm** and an empirical measure of a learning algorithm's incorrectness is given by the **Loss Function**. In order to develop the notion of correctness, one needs to tell the algorithm what 'correct' is by feeding it as input, a sufficiently large training dataset. Each instance of this dataset (a **training example**) is a pair of **features** (x) and an **output label** (y). The output label represents the desired correct answer and thus establishes the notion of correctness. The final goal of training algorithm is to calculate the correct output label for an input feature which was not present in the training dataset (and thus its output label is unknown). For e.g. If one is to build a learning algorithm that can classify an image as a cat or non-cat, then, a single training example's feature would be an image of a cat with an output label 'cat'. In order to train this learning algorithm, one can feed this image to the learning algorithm and measure the degree of incorrectness in its output using the output label (along with the loss function) of the fed training example. This incorrectness/loss is measured using the loss function. One could then tell the learning algorithm about its incorrectness and make it try again by taking this incorrectness into account. Thus, the algorithm learns from its mistakes and tries to minimize its incorrectness/loss using the output labels provided. This process

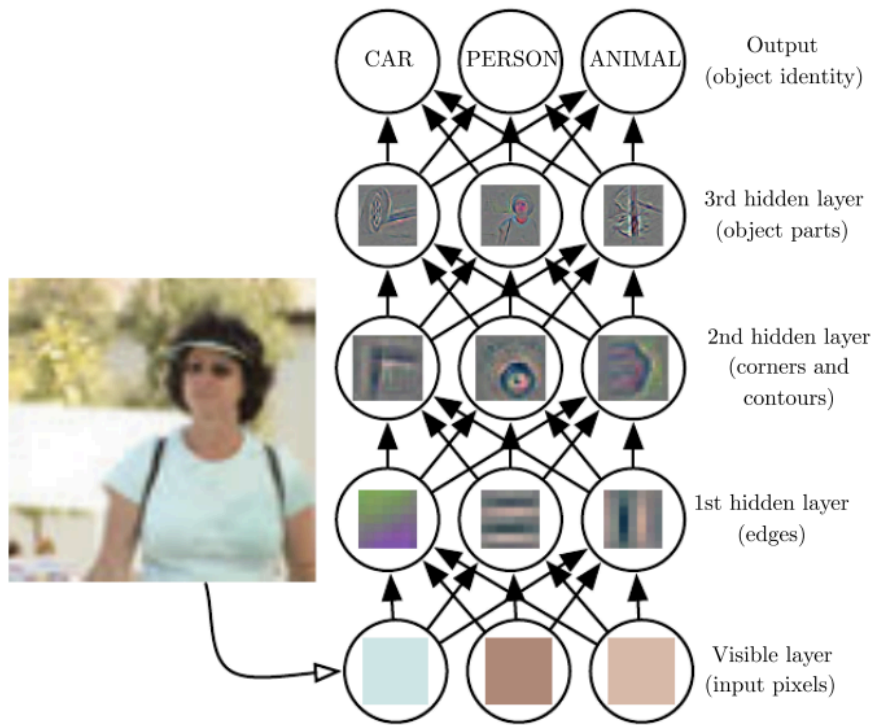


Figure 2.1: [16] Illustration of a deep learning model. It is difficult for a computer to understand the meaning of raw sensory input data, such as this image represented as a collection of pixel values. The function mapping from a set of pixels to an object identity is very complicated. Learning or evaluating this mapping seems insurmountable if tackled directly. Deep learning resolves this difficulty by breaking the desired complicated mapping into a series of nested simple mappings, each described by a different layer of the model. The input is presented at the visible layer, so named because it contains the variables that we are able to observe. Then a series of hidden layers extracts increasingly abstract features from the image. These layers are called “hidden” because their values are not given in the data; instead the model must determine which concepts are useful for explaining the relationships in the observed data. The images here are visualizations of the kind of feature represented by each hidden unit. Given the pixels, the first layer can easily identify edges, by comparing the brightness of neighboring pixels. Given the first hidden layer’s description of the edges, the second hidden layer can easily search for corners and extended contours, which are recognizable as collections of edges. Given the second hidden layer’s description of the image in terms of corners and contours, the third hidden layer can detect entire parts of specific objects, by finding specific collections of contours and corners. Finally, this description of the image in terms of the object parts it contains can be used to recognize the objects present in the image. Source: Figure 1.2 from Deep Learning [16]

is repeated until the algorithm learns a mapping from training example to its output label. This type of approach where one provides supervision to correct the algorithm's output, falls under the **Supervised Learning** category of Machine Learning algorithms.

Deep learning provides **representational composeability** i.e. it allows one to derive complex representations from simpler ones. This is done in layers with increasing complexities with initial layers representing simpler representations. Such a layered architecture is shown in 2.1. The figure shows how a deep learning algorithm attempts to learn **mappings** from images to labels. These layered architecture or networks are called Artificial Neural Networks (or ANN). The 'neural' here signifies the similarity of neural networks to the structure of inter-connected biological neurons. Researchers claim their initial motivation of neural networks from the miracle of the mammalian neural systems and its perceived functioning. It is however a known fact that biological neural networks are much more complex and more unexplored than the most complex of the existing ANN architectures. Although, an inspirational correlations between the two exists, but any further attempts at explaining ANN in this thesis will not use this scant correlation.

2.2 Essential Mathematics for ML

AI has been known to solve several problems using high performance computers. However, it must be known that the pillars of modern day AI or any of its sub categories consist of several classical disciplines from mathematics. Most significant from those are that of **Linear algebra**, **Probability & Statistics**, and **Calculus**. The intention here is not to provide an exhaustive course on each of them but to establish a general intuition about why any AI enthusiast must indulge in these disciplines and what are their respective contributions in a Machine/Deep Learning model.

2.2.1 Linear Algebra

Most of the concepts presented in this chapter are adapted from Ian Goodfellow's Deep Learning (chapter 2, [15]). The intuitions around visualization of vectors in a space are inspired by Grant Sanderson's online video series titled 'Essence of Linear Algebra' (3Blue1Brown, [44]).

One of the most important and relevant contribution from Linear Algebra to AI is the construct of Scalars, Vectors, Matrices and Tensors. Scalars represent single valued real numbers. A vector is an array of numbers which represent the vector's magnitude in each dimension. For e.g. a vector of size $n \times 1$ represents the tip of an arrow (a point) in an n-dimensional coordinate system and this arrow has its other end on the origin of this coordinate system i.e. it originates from the origin. Vector \mathbf{x} has dimensions 3×1 and hence is representing a tip of an arrow in a 3 dimensional space.

$$\mathbf{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} \quad (2.1)$$

Points are represented differently using parenthesis, these parenthesis is what tells the reader that this point should not be visualized as a vector's tip but as a standalone point in space. A matrix is represented by a 2 dimensional array which has rows and columns. A grid of arrays with m rows and n columns is called a $m \times n$ dimensional matrix. \mathbf{Y} represents a 3×3 matrix called a square matrix.

$$Y = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} \quad (2.2)$$

Intuitively, a $m \times n$ matrix can be imagined as a collection of n column vectors of size/length m each, stacked adjacent to each other. Tensors are higher dimensional

matrices and the algebra of matrices can be generalized to tensors. The future discussions will be mostly about matrices and vectors, and will include tensors when required.

Vectors in their essence are a language to describe space. Scalars (a number) when multiplied with vectors (an arrow) either stretches them or shrinks them depending on whether the chosen scalar is greater or less than 1. For a 3 dimensional coordinate system the direction of x, y and z axis is given by vectors of unit magnitude in each direction represented by \hat{i} , \hat{j} and \hat{k} respectively. Thus, another way to interpret the 3D space vector in equation 2.1 is that its tip can be reached by scaling the \hat{i} unit vector by a scalar x_1 , the \hat{j} unit vector by x_2 and the \hat{k} vector by x_3 . The scalars x_1 , x_2 and x_3 tells how much one needs to 'walk' in each dimension. With this idea a vector can be represented as follows:

$$x = (x_1)\hat{i} + (x_2)\hat{j} + (x_3)\hat{k} \quad (2.3)$$

This addition of two or more vectors after multiplying them with a scalar results in a new vector (x) which is a **linear combination** of these vectors.

Using all possible combinations of these vectors with all possible scalar values, one can represent any vector in the 3 dimensional space. This set of all possible vectors is known as the **span** of the unit vectors. For imaginative convenience, think of an arrow in space when a single vector is being discussed, but when an intuition about all possible vectors is to be made, it is better to think all of them as points in space (without a tail going to origin). This way one can imagine that the **span** of the combination of these three unit vectors in a 3D space is actually every possible point. The unit vectors are more formally known as the **basis vectors**. It is important to think about the span of one single vector which by definition is all possible vectors one can create by scaling this single vector. One can imagine that a single vector (an arrow) when multiplied by a number (a scalar) may only increase (if number > 1) or decrease (if number < 1) in its length, and therefore

a combination of all possible vectors would turn out to be a single line extendable in both directions. The span of a single vector is therefore a **straight line**.

The confident inference that one can reach any point in the 3D space by different combination of the three basis vector (\hat{i} , \hat{j} and \hat{k}) relies on the assumption that these vectors are independent to each other. This independence can be thought of as the fact that any of the basis vectors **cannot** be expressed as linear combinations of other basis vectors. This lack of a linear combination makes them **linearly independent** to each other.

When dealing with a large dataset of many instances of unknown vectors, it becomes important to know how many independent dimensions the data has. These dimensions are proportional to the total number of linearly independent vectors one can find. These dimensions actually represent features of the data that are independent and represent individual concepts that are canonical and thus cannot be derived from other concepts. Such independent were introduced as **factors of variation** in section 2.1.2.

A Matrix from a **computer science perspective** is a data structure to store numbers in order to perform faster mathematical operations on these numbers. What matters, is the cost (in terms of processing time) of accessing an element in a matrix or multiplying two matrices together. But the fact that matrices and Linear Algebra predates the first computer by centuries suggests that matrices are more than just objects for faster computations.

A Matrix from a **mathematical perspective** is a type of **linear transformation**. In other words, a $m \times n$ matrix can be used to transform a vector of length 'n' to a vector of length 'm'. This transformation is realized by matrix multiplication which states that on multiplying a $m \times n$ matrix by another $n \times 1$ vector, the resulting vector is of dimension $m \times 1$. Thus, it is concluded that a vector in an n-dimensional space can be transformed to an m-dimensional space using a $m \times n$ matrix. Thus, a linear transformation represents a **mapping** from

one vector space or **feature space** to another. This idea is discussed with the following intuitive example.

Matrix-Vector Multiplication: Supposing a 2×2 matrix **A** (equation 2.4) is to be multiplied with a 2×1 vector **v** (equation 2.5).

$$A = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \quad (2.4)$$

$$\mathbf{v} = \begin{bmatrix} 2 \\ 5 \end{bmatrix} \quad (2.5)$$

Referring to previous discussions on visualizing a vector, a 2×1 vector represents a tip of a vector or a point in a 2-dimensional space. By convention, the upper element represents the coordinate on x-axis and the lower element represents the coordinate on y-axis. As explained above, \hat{i} and \hat{j} are the two basis vectors that can be used to reach any point in this space by changing the value of scalars multiplied by them. So one can write the vector in equation 2.5 as a linear combination of its basis vectors as shown below.

$$\mathbf{v} = (2)\hat{i} + (5)\hat{j} \quad (2.6)$$

The matrix **A** tells what happens to vector **v** when it is multiplied by the matrix, i.e. where does vector **v** land. A geometrical intuition for linear transformation is that once a linear transformation is applied to any given space of vectors, the resulting space preserves the property that all lines must remain lines and that the origin must remain fixed. Since all lines remain lines and the origin is fixed, any linear transformation will at most reverse the direction of a vector, or may be stretch it (elongate or shrink), or may be move its tip to a new point. Whatever the case is, the transformed vector can still be represented as a combination of some basis vectors which represent independent dimensions in this **new** linearly

transformed space. The transformed vector v_t can be represented as a combination of these **transformed** basis vectors as follows:

$$v_t = (2)\text{transformed_}\hat{i} + (5)\text{transformed_}\hat{j} \quad (2.7)$$

Therefore, If one could somehow know how much the basis vector will be scaled, stretched or reversed in this new linear space, one could use this information to transform any vector from the old space to the new one. This transformation is what a matrix represents. The first column (0,1) can be thought of as representing the coordinates to which the old \hat{i} basis vector will move to, similarly the second column (-1,0) represents the new coordinates of \hat{j} basis vector. These new coordinates for \hat{i} and \hat{j} are represented by $\text{transformed_}\hat{i}$ and $\text{transformed_}\hat{j}$ respectively. Thus, the new position (v_t) of vector \mathbf{v} (in equation 2.5) can be calculated by multiplying it first with the $\text{transformed_}\hat{i}$ coordinates and then with that of $\text{transformed_}\hat{j}$ as:

$$v_t = (2) \begin{bmatrix} 0 \\ 1 \end{bmatrix} + (5) \begin{bmatrix} -1 \\ 0 \end{bmatrix} = \begin{bmatrix} 2 \times 0 \\ 2 \times 1 \end{bmatrix} + \begin{bmatrix} 5 \times -1 \\ 5 \times 0 \end{bmatrix} = \begin{bmatrix} 0 \\ 2 \end{bmatrix} + \begin{bmatrix} -5 \\ 0 \end{bmatrix} = \begin{bmatrix} -5 \\ 2 \end{bmatrix} \quad (2.8)$$

The steps of matrix multiplication are not covered in detail, however equation 2.8 can be observed to capture the dynamics of matrix-vector multiplication.

The result in equation 2.8 shows that the tip of the original vector has been **transformed** to point (-5,2). The same result can be reached by multiplying matrix **A** by a vector \mathbf{v} in a relatively more common form.

$$\mathbf{v_t} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 2 \\ 5 \end{bmatrix} = \begin{bmatrix} (0 \times 2) + (-1 \times 5) \\ (1 \times 2) + (0 \times 5) \end{bmatrix} = \begin{bmatrix} -5 \\ 2 \end{bmatrix} \quad (2.9)$$

One can read 2.9 as **linear transformation** of a 2×1 vector by a 2×2 matrix.

Matrix-Matrix Multiplication: By Extending the intuition for matrix-vector multiplication, one can reason about what a matrix-matrix multiplication

would represent. Since a matrix represents a linear transformation, its multiplication with another matrix can be thought of as applying another linear transformation on top of the previous one. Thus, the resultant matrix of a matrix product of **matrix A** and **B** when multiplied with any vector, will be equivalent to multiplying that vector first by matrix B followed by multiplying the previous result with matrix A. Hence matrix products can compose many linear transformations together and can represent the resulting n-dimensional space just by a single $n \times n$ matrix.

Dot Product: Dot product provides a way to reason about the direction of two vectors relative to each other. Dot product of vector a and vector b is possible if both are of same dimensions. Following equation shows how to calculate the dot product of two given vectors.

$$\mathbf{a} \cdot \mathbf{b} = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = a_1 \times b_1 + a_2 \times b_2 \quad (2.10)$$

The above equation shows the algebraic definition of the dot product of two vectors, and shows that it is actually a scalar value. Following is a geometric formulation of a dot product:

$$\mathbf{a} \cdot \mathbf{b} = \begin{bmatrix} a_1 \\ a_2 \end{bmatrix} \cdot \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = |a||b|\cos(\theta) \quad (2.11)$$

The $|a|$ and $|b|$ correspond to the magnitude of the two vectors and is calculated as $\sqrt{a_1^2 + a_2^2}$ and $\sqrt{b_1^2 + b_2^2}$ respectively. The angle θ denotes the angle between two vectors.

Dot product is used to reason about how much two vectors point in the same direction. If the dot product of vector a and b equals the product of their magnitudes, this entails that $\cos(\theta) = 1$. Since $\cos(\theta)$ is equal to 1, the angle θ between the two vectors must be **zero**, therefore the vectors must be pointing in the same

direction. This entails that, the more similar is the direction in which the vectors point, the higher the value of their dot product would be. Consequently, the dot product of two orthogonal (perpendicular) vectors is zero.

Linear Transformation in Deep Learning: Applications of linear transformation are many, but in the context of ML and Deep Learning one can establish context by referring to the previous discussion about AI (section 2.1.1) and Deep Learning (section 2.1.2). It was described how Deep Learning aims to learn the mappings from input to output data. When both inputs and outputs are modelled as vectors, these mappings can be represented by **learned** transformation matrices which transform the input vector into the output vector. While training a supervised machine learning algorithm, both input and output vectors are known and the aim is to learn/calculate the transformation matrix. Once this transformation matrix is learned, one can predict the output vector for any unseen input vector by multiplying the input vector with the learned matrix.

Modelling entities as matrices and vectors not only facilitates representational convenience and faster calculations, but also introduces techniques that can provide insights into the training data. Linear Algebra is full of such techniques and the one that tops the charts is called **Eigen Value Decomposition**.

Eigen Vectors, Eigen Values and Eigen Decomposition: A matrix transforms vectors from one space (dimension) to another. When a vector is linearly transformed, the original vector rotates and stretches to a new point in space and has a new span (a straight line). But there are some special vectors that when transformed (using a matrix), **do not** change their spans i.e. they are only stretched and **not rotated**. By definition, a span of a vector is all vectors that can be reached by extending the vector in both directions. Using this definition, one can say that the transformed version of this special vector **still** lies in its old span. Using spans to define eigen vectors, Eigen vectors of a matrix (a linear transformation) can be understood as all those vectors whose span remains

unchanged when multiplied (transformed) with this matrix. Although the span remains unchanged as the eigen vectors are not rotated, but they can be stretched by a scalar. This scalar is called the **Eigen value** corresponding to a eigen vector. The following equation captures the intuition of eigen vectors and eigen value.

$$\mathbf{A}\mathbf{v} = \lambda\mathbf{v} \quad (2.12)$$

Equation 2.12 can be interpreted as follows: If a vector \mathbf{v} is an eigen vector of a matrix \mathbf{A} then its transformation by \mathbf{A} must be equivalent to scaling \mathbf{v} by a scalar λ where λ is referred to as the eigen value of vector \mathbf{v} .

Arranging all eigen vectors and eigen value for a given matrix \mathbf{A} together in a matrix followed by some basic matrix arithmetic, the following result is achieved.

$$\mathbf{A} = V \text{diag}(\lambda) V^{-1} \quad (2.13)$$

In equation 2.13, \mathbf{A} is the matrix being decomposed, \mathbf{V} is a matrix of all possible eigen vectors, and $\text{diag}(\lambda)$ is a matrix of corresponding eigen values. The **diag** represents **diagonal matrix** i.e. its non-diagonal elements are zero. **Diagonal matrices** simplify computations, for e.g. calculating a square of a diagonal matrix is just squaring all its diagonal elements which are just scalars. This ease of computation makes them easier to work with, therefore, whenever possible, a diagonal matrix representation must be chosen. The V^{-1} represents the **inverse of matrix \mathbf{V}** . A good way to think about an inverse is as a reverse transform of the original matrix, i.e. if a vector is transformed by a matrix and then by its inverse, it remains unchanged. The equation 2.13 more formally represents the **Eigen Decomposition** of matrix \mathbf{A} . Eigen decomposition of a matrix apart from providing higher computational efficiency (from when the same operation is done with the original form of matrix), also provides insights in the algebraic nature of the decomposed matrix. This is because the eigen values actually measures the

variance of data around an eigen vector. Unfortunately, Eigen decomposition only exists for **square matrices** ($n \times n$) but fortunately, there are other techniques of decomposition.

SVD - Singular Value Decomposition: Unlike Eigen decomposition, the SVD exists for any $m \times n$ matrix. Before explaining SVD, the notion of **Rank** of a matrix must be established. Earlier discussions about the linear independence of two vectors confirms that the two vectors which are linearly independent to each other **cannot** be expressed as linear combination of each other, i.e. both these vectors are needed to represent the space. Extending this idea to a matrix (collection of vectors), rank **p** of a matrix can be defined as the total number of linearly independent vectors (columns) of the matrix. The rank cannot be greater than the number of columns thus $p \leq n$. Using SVD, any $m \times n$ matrix **A** can be decomposed as a sum of matrices of rank 1, each containing one of the 'p' linearly independent vectors. Each summation unit 'i' is obtained by multiplying **Left Singular Vector** u_i of dimension $(m \times 1)$ with the transpose of **Right Singular Vector** v_i of dimension $(n \times 1)$ which results in matrix of dimensions $(m \times n)$; same as the original matrix. Further, each rank 1 matrix is scaled by its corresponding **Singular Value** (σ). A matrix **A** can be decomposed as follows:

$$\mathbf{A} = \sigma_1 u_1 v_1^T + \sigma_2 u_2 v_2^T + \dots + \sigma_p u_p v_p^T \quad (2.14)$$

Stacking all the 'u' and 'v' terms in matrices, SVD can be represented in a similar fashion like eigen value decomposition.

$$\mathbf{A} = U \text{diag}(\sigma) V^T \quad (2.15)$$

SVD and Eigen Decomposition: SVD uses eigen decomposition as follows. Given a matrix A, the matrices AA^T and $A^T A$ are calculated. For any given matrix of arbitrary dimensions both AA^T and $A^T A$ are square matrices. Eigen

vectors of AA^T and $A^T A$ are calculated and are composed in matrices U and V (of equation 2.15) respectively. The eigen vectors for both matrices are same and the singular matrix ($diag(\sigma)$) value is actually the squared root of the corresponding eigen values.

But what is the importance of knowing the rank and using it for SVD? Supposing there is a very large (high dimensional) matrix, and its rank 'p' is significantly lower than the number of columns. This indicates that there are many columns ($n - p$) which are **linearly dependent** to each other and thus do not represent independent dimensions of data. The corresponding σ term would be negligible for such vectors. From a SVD decomposition of a given matrix, one can simplify the representation of a given matrix by just taking the terms whose corresponding σ values are significant. This reduces the otherwise high dimensional (many columns) data to dimensions (columns) that are most significant, such a reduction in dimensions positively impacts the performance of the learning algorithm.

Principal Component Analysis: PCA or Principal Component Analysis is a dimensionality reduction technique used in varied disciplines. Given some numerical data, each data point can be represented by a vector, and all the vectors are combined together into a matrix. The key objective of PCA is to find the most relevant dimensions (or basis) using which this data can be best described. Since basis are orthogonal to each other, it is desired that the dimensions calculated from PCA of the matrix are orthogonal. Intuitively, given a data in many dimensions (hence many basis), PCA aims to find the dimensions along which most of the data is centered and varying. If the found dimension is chosen to analyze and represent data, it will be easier to interpret this data as the number of dimensions has reduced. Apart from dimensionality reduction, PCA can also be understood as an instrument of feature extraction, where feature refer to the most relevant features across which most variation in the data is observed. PCA for any arbitrary matrix can be done using singular value decomposition. SVD decomposes a given

matrix in several compositions each corresponding to a singular value. These singular values are squared root of eigen values, which in turn show variance of data around the corresponding eigen vector. PCA can then choose 'k' vectors ('k' is significantly less than dimension of data matrix) corresponding to high singular values (or variance) as new basis of the data. These selected basis are called principal components of the data.

Linear algebra provide means to conveniently model/represent real world data in a well studied mathematical object called matrix. Once such a representation is realized, known properties of matrices can be applied to explore the data and to represent mappings of vectorized input features to their corresponding vectorized output values. Elimination of redundant dimensions is another powerful construct and is known as **dimensionality reduction**.

2.2.2 Probability and Statistics

2.2.2.1 Probability

Probability is a tool to quantify uncertainty. The entity whose uncertainty is being quantified is called a **random variable**. Random variable could be **discrete** or **continuous** in nature. Supposing the random variable is the result of tossing an unbiased coin. The two possible outputs; heads and tails of the random variable result are referred to as its **sample space**. Assuming the coin is fair, there is a 50% chance of heads and a 50% chance of tails to be the final state of result. Such a probability can be expressed as follows:

$$\mathbf{P}(result = heads) = \mathbf{P}(result = tails) = 50\% = \frac{1}{2} \quad (2.16)$$

Probability Distribution: is a collection of probabilities corresponding to every possible state of all possible random variables in the entity being modelled (chapter 3, Deep Learning [16]). Depending upon whether the variable is discrete or continuous, the probability distribution is given by a **Probability Mass**

Function(P) or **Probability Density Function(p)** respectively. Both of these functions can also represent distributions of more than one variable and then the corresponding distribution is called a **joint** distribution. For a given probability mass function the sum of all probabilities must be 1. This is called the **Normalization Constraint** . The **joint probability** of two discrete random variables X and Y is represented as follows:

$$\sum_i \sum_j P(X = x_i \text{ and } Y = y_j) = 1 \quad (2.17)$$

Marginal Probability is the probability of a subset of variables from a joint probability distribution. The marginal probability for any $X = x_i$ in the equation 2.18 can be understood as the sum of all probabilities corresponding to every possible state of random variable Y when $X = x_i$ (chapter 3, Deep Learning [16]). This result can be generalized for more than one random variable.

$$\forall x \in \mathbf{x}, P(X = x) = \sum_y P(X = x \text{ and } Y = y) \quad (2.18)$$

Conditional probability gives a probability of an event happening(event 1) given that another event has already happened/observed (event 2). The events here could mean a variable X taking a particular value x given that variable Y has already taken on value y, such a conditional probability can be expressed as:

$$P(Y = y \mid X = x) = \frac{P(Y = y, X = x)}{P(X = x)} \quad (2.19)$$

The denominator here is the marginal probability ($P(X = x$ or $P(x)$) . The notion of conditional probabilities give rise to a fundamental rule called **Chain rule**. Chain rule decomposes a joint probability distribution over arbitrary number of random variables. Using chain rule, one can express a joint probability

distribution as combination of conditional probabilities of the constituent random variables as follows (chapter 3, Deep Learning [16]).

$$P(a,b,c) = P(a \mid b, c)P(b, c) \quad (2.20)$$

$P(b,c)$ can be further decomposed as:

$$P(b,c) = P(b \mid c)P(c) \quad (2.21)$$

After substituting 2.21 in 2.20 the final decomposition of the joint probability distribution $P(a, b, c)$ is given as:

$$P(a,b,c) = P(a \mid b, c)P(b \mid c)P(c) \quad (2.22)$$

Expected Value or expectation over a probability mass function is the summation over products of all possible states of a random variable(s) and its corresponding probability. It can also be thought of as multiplying each of the likely outcome by its likelihood and adding all the values to output a single value. Expected value is often used to form an informed data driven assumptions about the most likely value a random variable would take. The expected value is equal to the arithmetic mean (average) of all possible states given that the probability for each state is same.

Variance(σ^2): Describes the spread of a distribution (Chapter 2.8, Think Stats [8]). This spread can be visualized as a measure of how the values vary around the **mean** value of the distribution. The mean(μ) is analogous to average and is used to calculate variance. A lower value of variance means that the distribution is clustered near the expected value and a higher value represents that data is widely spread and not clustered around the mean (Chapter 2.7, Think Stats [8]).

Standard deviation(σ) is another metric calculated by taking the square root

of variance. Given a random variable defined by a function $f(x)$ with its expected value $E(f(x))$, the variance can be calculated as:

$$\text{Var}(f(x)) = E[(f(x) - E[f(x)])^2] \quad (2.23)$$

The **Covariance** between two random variables is a measure of how related they are. A high absolute value of covariance (not considering the sign) for two variables indicate that both of them simultaneously diverge significantly from their respective means (chapter 3.8, Deep Learning [16]). The sign of the calculated covariance is equally significant, for e.g. if both the variables are positive or negative simultaneously, then their covariance is a positive value. Contrary to this, if one variable is positive while other is negative, the covariance is negative. Covariance jointly captures the relation of two variables with their own respective means and also with each other. The covariance of two given random variables (represented as vectors) with zero mean (the mean of each variable is subtracted from all its value) is equivalent to their dot product (explained in section 2.2.1) (Chapter 7.4, Think Stats [8]). Thus, covariance is positive if the two vectors point in the same direction and negative when two vectors point in opposite direction and is zero when the two vectors are orthogonal or independent (similar to linearly independent vectors explained in 2.2.1).

There are many known probability distributions that are used to model different behavior of random variables. Any probability distribution can be expressed using features like mean and variance. Distributions are important because they can be used to model assumptions for the random variable in question. For e.g. given a random variable whose all states have the same probability, then a **Uniform Distribution** could be used to model its probability distribution. A uniform distribution assigns the same probability to each state of the random variable. For a random variables that have two outcomes a **Bernoulli Distribution** is

used. There are many distributions that can be explored depending on the nature of random variable being modelled.

Normal Distribution or Gaussian Distribution: is one of the most commonly found **continuous** distributions. When plotted on a graph, the distribution forms a bell shape which is referred to as a bell curve. Corresponding to the highest point on this bell is the mean of this distribution. Rest of the data is symmetrically divided on either side of this mean. Precisely, 68% of the data is within **one** standard deviation(σ) away from the mean(μ), i.e. between $\mu - \sigma$ and $\mu + \sigma$, 95% lies between **two** standard deviation which is $\mu - 2\sigma$ and $\mu + 2\sigma$, and 99.7% is spread between **three** standard deviation. Thus, if a given continuous random variable is known to have a normal distribution, a lot of inferences can be made about the spread of its data points. Another significant use of normal distribution is realized by the **Central Limit Theorem**. The theorem states that when sufficient number of random samples (each containing datapoints) are collected from a large unknown and arbitrary distribution of data and the mean of each sample is calculated, the distribution (plotted graph) of these means **approaches** a normal distribution. Further, the mean of this approaching normal distribution is equal to the mean of the original distribution, and its variance is equal to that of the original distribution when reduced by a factor of 'n' where 'n' is the size of the sample (at least 30). Further, the larger the number of samples and size of each sample, the more normal the distribution becomes.

Bayes' Theorem: Provides a proven formula to update the probability for a certain event given some prior information. Specific to supervised machine learning, a hypothesis is a function that maps inputs to their output labels using a matrix (explained in Linear Algebra section 2.2.1). Each value of this matrix is known as a parameter, and this parameter matrix is often represented by the term θ . The parameters of the matrix θ are **learned** during the training phase. During training, the parameter matrix θ is updated iteratively which in turn

updates the hypothesis. Every succeeding update takes into account the **prior** state of the hypothesis. Using the tool of conditional probability, this problem can be reformulated as estimating a conditional probability that the calculated hypothesis **H** is correct (first event) when the training data is observed (second event). This conditional probability can be represented as $P(H|Data)$. Using Bayes' theorem, one can calculate this conditional probability as follows:

$$P(H | Data) = \frac{P(Data | H) P(Model)}{P(Data)} \quad (2.24)$$

In the above equation, $P(Model)$ is called as the **prior** probability. It represents the information one has (if any) about the hypothesis and its parameters **before** observing the data. The prior is usually estimated by domain experts. Alternatively, a probability distribution (e.g. Normal or Bernoulli) could be assumed over the parameters and using this distribution, a **prior** could be created as a starting point for learning. After each data sample is fed to the learning algorithm, the quantity $P(H | Data)$ is calculated. $P(H | Data)$ is called the **posterior** probability. Before the next training iteration begins, the prior is updated and is assigned the value of the posterior. This process continues iteratively until the training is complete.

After developing the aforementioned intuitions about probability, supervised learning can be expressed as predicting the output label y (calculated using hypothesis H), of a given input x (comes from training/testing data), using the conditional probability $P(y|x)$. Where y is the predicted output label and x is the seen/unseen input.

2.2.2.2 Statistics

Statistics and Probability are two related disciplines of mathematics. While probability predicts the likelihood of future events, statistics analyze the data collected in the past. Although this boundary is at times fuzzy, but one could call prob-

ability a theoretical approximation of a distribution and statistics as an applied branch of mathematics which analyzes a given distribution. Skiena in his book 'Calculated Bets' [46] explains this distinction as *"Probability theory enables us to find the consequences of a given ideal world, while statistical theory enables us to measure the extent to which our world is ideal"*.

Statistics form the basis of modern day algorithms for Machine Learning. Both Machine Learning and Statistics develop understanding of how the data is distributed, but the key difference between the two is that Machine Learning does this with the aim of making predictions on unseen data, whereas statistical inferences aim to understand the relationship between the data. Prediction of unseen future data is not the central aim for a statistical inference task but just a possible by-product.

A supervised ML algorithm maps a given input to its corresponding output label with the help of a function (or hypothesis). This function contains parameters that are updated iteratively until the function reaches the desired accuracy. The estimation of these parameters and the function itself falls under a topic of statistics called **Point Estimation**. Through point estimation, a single most appropriate prediction of some quantity of interest is provided (chapter 5.4, Deep Learning [16]). The quantity of interest could be a vector of parameters or even a function. A point estimation learns a mapping between the input and output labels, the resulting point estimate is essentially a function. Goodfellow et al. refer to such point estimates as **Function Estimators** (Pg. 124, Deep Learning [16]). When observed closely a function estimator is just a point estimator in the function space (Pg. 124, Deep Learning [16]). Thus, estimating a mapping from input x to output y can be interpreted as either estimating parameters or estimating a function mapping from x to y . Notationally, an estimation of a quantity q is denoted as \hat{q} .

Machine Learning is mostly about estimating data distributions whose true nature is elusive to us. This estimation requires sampling as much data (training set) as we can in order to establish an empirical basis. Different learning constructs (algorithms) are then applied to approximate (fit) the distribution of collected data samples. These learning algorithms differ in the assumptions they work with and a good match happens when the assumptions of the learning algorithm exists in the sampled data.

Maximum Likelihood Estimation(MLE): is one of the statistical techniques used to fit the given data to match our assumed probability distribution. Consider a distribution $P(D)$ that is unknown. After sampling data from the world i.e after attaining pairs of input x and output label y , an estimate about the distribution is made which is represented as $\hat{P}(D)$. Further, this estimation is parameterized by a parameter θ and the estimated distribution is now represented as $\hat{P}(D; \theta)$. MLE then provides the value of θ that maximizes the likelihood of this estimated distribution matching the true (but hidden) distribution ($P(D)$). For e.g. $\hat{P}(D; \theta)$ can be assumed to follow a Gaussian probability distribution. The parameters that define a Gaussian distribution are mean and variance, this mean and variance can be used to approximate θ . The process of calculating the θ that maximizes a given function is simply by calculating the partial derivative (explained in 2.2.3) of the function w.r.t θ and equating it to zero. Solving for θ then gives the value that when plugged in $\hat{P}(D; \theta)$ maximizes the likelihood of observing the true (but hidden) distribution $P(D)$.

Maximum a Posteriori Estimate(MAP): The two schools of thought in probability & statistics are that of Frequentist and Bayesian. While the Frequentist approach of estimating probabilities is just by observing the given data, Bayesian on the other hand allows inclusion of prior belief for estimation. MAP can be considered as the bayesian version of MLE. The divide between frequentist and bayesian can further be understood on the basis of how θ is modelled. The pa-

parameter θ is no longer **just** a parameter in the bayesian approach but a **random variable**. Since a random variable comes from a probability distribution, θ can now have its own probability distribution that can be modelled using a prior belief. While MLE finds the best estimate of θ that maximizes the probability (likelihood) of seeing the observed data, MAP answers to the question, given the data what is the most likely θ ?

$$\text{MLE} : \theta = \text{argmax}_{\theta} P_{\theta}(\text{Data}) \quad (2.25)$$

$$\text{MAP} : \theta = \text{argmax}_{\theta} P_{\theta}(\theta | \text{Data}) \quad (2.26)$$

In the above equations argmax refers to the value of theta that maximizes the function on the right. Expanding $P(\theta | \text{Data})$ using bayes rule (equation 2.24) results in the following equation.

$$P(\theta | \text{Data}) = \frac{P(\text{Data} | \theta) P(\theta)}{P(\text{Data})} \quad (2.27)$$

The term $P(\text{Data} | \theta)$ can be calculated using maximum likelihood estimation. $P(\theta)$ represents the prior and $P(\theta | \text{Data})$ represents the posterior probability, hence the name **maximum a posteriori probability**. One of the advantages of bayesian approach is that even in scenarios where training data is not ample, a correct prior belief can make the model work. Alternatively, in cases where the prior is highly incorrect, ample amount of data is needed to correct the model.

In Supervised Learning, classification is one of the most common objectives of a learning algorithm. Classification refers to mapping (or classifying) a given input to its correct class label. The label of each class can be modelled as a random variable and its distribution can be approximated. Approaches like these which aim to learn probability distribution of each class in order to differentiate between them fall in the category of **Generative Modelling**. Generatively modelled classifiers

learn to differentiate between two classes by estimating their respective distributions, thus they capture the essence of each class. This is opposed to another class of classification algorithms called **Discriminative Classifiers** (for e.g. Perceptron explained in section 2.3.2), which instead try to learn a **decision boundary** separating the two classes, details of which are deferred to later sections. Seemingly, the Discriminative classifiers have outdone their Generative counterparts in the modern day Machine Learning scenario.

2.2.3 Calculus

"Calculus is the mathematical study of continuous change, in the same way that geometry is the study of shape and algebra is the study of generalizations of arithmetic operations." - Magno Urbano (Chapter 2, Infinitesimal Calculus [48]).

Calculus or 'Infinitesimal Calculus' (Chapter 5, Mathematics for Machine Learning [7]) is a mathematical discipline that deals with approximation of areas spanned under any arbitrary curve (representing some mathematical function) and its rate of change. Integral calculus (or integration) approximates the area (or volume etc.) and differential calculus (or differentiation) provides reasoning about how the function changes around a given point. Figure 2.2 provides a visual representation of integration and differentiation on a function/curve $f(x)$. While both integration and differentiation serves Machine Learning in different ways, this section only details a particular contribution of differential calculus that led to the development of **Gradient Descent** algorithm.

Gradient Descent Gradient descent is an optimization algorithm. Optimization refers to the iterative process of finding an optimum solution for a given problem. In order to find the optimum solution, an optimization objective is to be defined. One such optimization objective is to find the **minimum** value of a given function. Given a function $f(x)$, its minimization objective attempts to find a value of its parameter 'x'; such that when $f(x)$ is parameterized by 'x', the

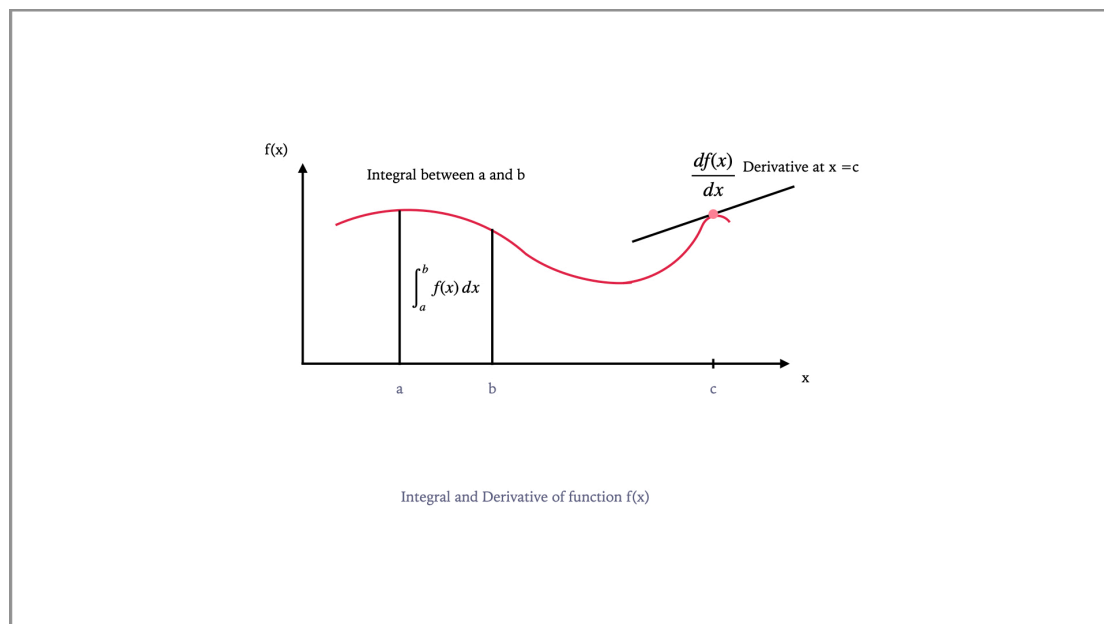


Figure 2.2: Simplistically, integration of a function refers to the area between two values of its parameters. Function $f(x)$ is parameterized by x . The value of the area spanned by the function between $x = a$ and $x = b$ is given by integrating the function from a to b . The derivative signifies the rate of change or gradient of the function at a given point. The derivative of $f(x)$ at a point c is shown. The slope is **positive** because it goes upwards as x increases. A negative slope would decrease as x increases. Source: Redrawn from figure 1.1, Pg 2, Tom M. Apostol - Calculus, Volume-1 [1]

resulting value is the smallest possible value of $f(x)$. This point of smallest value is called as the **minima** of $f(x)$.

Since the gradient descent finds a minima and does not create one, it is essential that the function $f(x)$ has a minima that can be found. Gradient descent provides best results when $f(x)$ has only **one** minima. Functions which have only one minima are known as **Convex** functions. A non-convex function can be visualized as having more than one locations where the slope of the derivative is zero, amongst all such locations, the one where the function has the least value is called as the **Global minima**. Thus, a non-convex functions can have more than one local minima but only one global minima, whereas a convex function only has only one minima which is both its local and global minima.

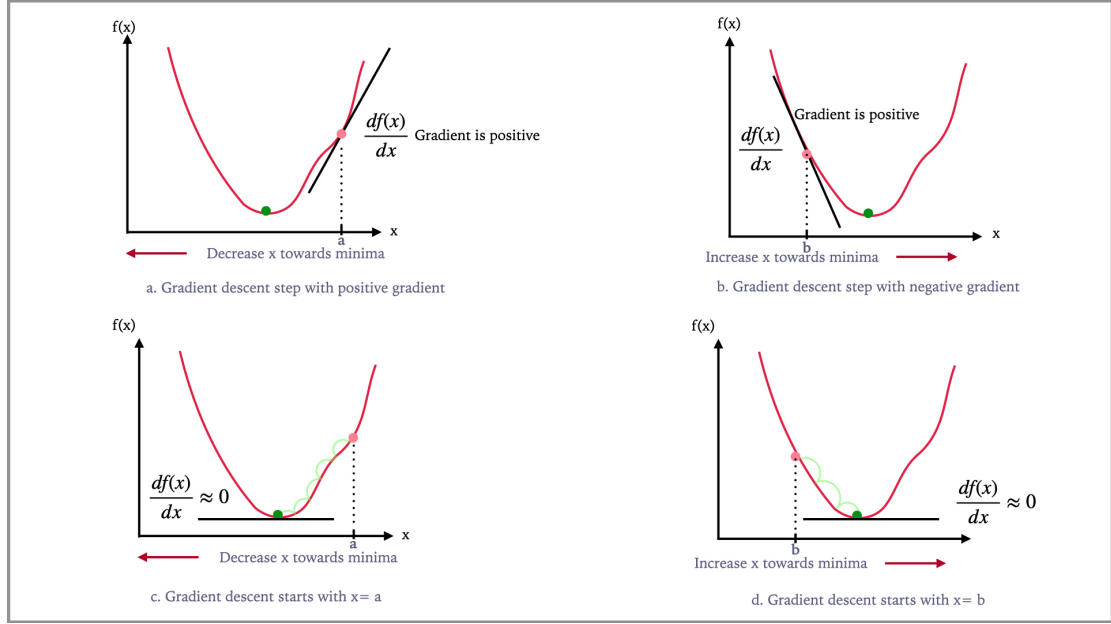


Figure 2.3: Interpreting gradient descent: a) Derivative at point $x = b$ results in a positive gradient. x is decreased towards the minima. b) Derivative is positive and x must be increased towards the minima. c. and d. shows all steps of gradient descent (in green) when a and b are used as starting points respectively.

The algorithm is named Gradient descent after the term gradient. Gradient is a vector quantity that represents the slope as well as the direction of the slope. It is calculated by taking the derivative of the function at a given point. Thus, if the slope is negative, the gradient would be negative and vice versa. Gradient descent is an iterative algorithm that updates the parameter ' x ' until the minima is reached. Each update is guided by a predefined step-size which correspond to the amount by which the parameter ' x ' will be adjusted. The update step of gradient descent formulation can be written as follows:

$$x_{new} = x_{old} - (step_size)(gradient) \quad (2.28)$$

The above equation decreases parameter 'x' when gradient is positive and vice versa. Gradient descent iteratively finds the minima of $f(x)$ using the following algorithm.

1. Start with any random value of $x = x_0$
2. Calculate the gradient (derivative) of $f(x)$ at $x = x_0$
3. Observe the slope at $x = x_0$
 - a) If slope is positive -> Decrease x by a small step. See figure 2.3 (a)
 - b) If slope is negative -> Increase x by a small step. See figure 2.3 (b)
4. Repeat 1-3 until x is unchanged (or slightly changed) after several updates, or the slope is almost **zero**. 2.3 (c) and (d).

Using the above formulation, the algorithm finds the minima of a convex function $f(x)$. An alternate intuition of the above process is that each step calculates the gradient at a particular value of x, this gradient is then used to reason about how the function $f(x)$ will change when a slight change in its parameter 'x' is introduced. Thus, the gradient descent algorithm uses gradients as a tool to vary the parameter x such that the resulting value of $f(x)$ is the smallest possible value. Further, since the function $f(x)$ has only one parameter, this example corresponds to gradient descent in one dimension.

Multi Variable Functions: The functions that have more than one parameters are called multi-variable functions. For e.g. $f(x,y)$ represents a multi variable function with two parameters. Gradient descent can be applied to this function in a similar fashion but with slight changes. First, since there are two parameters (x and y), each update step updates these two parameters **simultaneously**. Second, the parameter 'x' is updated by calculating the gradient of the function $f(x,y)$ with respect to (w.r.t) 'x' **only**. Similarly 'y' is updated by the gradient of 'y' w.r.t. to $f(x,y)$. Thus two separate gradients corresponding to x and y are calculated

for each update step. The gradient of a multi-variate function w.r.t. to a single variable is called as the **Partial Derivative** of the function w.r.t. that variable, and the branch of calculus that defines the rules for this is called **Multivariate Calculus**.

Machine learning employs gradient descent for minimizing a given objective function sometimes know as the **Loss Function**. In practice, the objective function has many parameters that can be simultaneously updated using gradient descent and the rules from multivariate calculus.

Chain Rule: Sometimes, functions are composed of other function, i.e a function $g(x)$ can be an input to function $f(g(x))$. This can be expressed as :

$$u = g(x), \text{ and } y = f(u) \quad (2.29)$$

In such a composition of functions, the final function 'y' is affected by parameter 'x' **indirectly** through function 'u' (or $g(x)$). To calculate how a change in parameter 'x' affects the variable 'y', the chain rule can be applied as follows:

$$\frac{dy}{dx} = \frac{dy}{du} * \frac{du}{dx} \quad (2.30)$$

The chain rule can be applied to any depth of composed functions. The gradient descent and chain rule are combined together to formulate one of the most essential algorithms in Deep Learning which is known as **Back Propagation**. There are many variants of gradient descent, Zhang (2019) [52] provides a detailed overview of such variants along with insights in the structure of gradient descent algorithm.

The intuitions built in this section (2.2) would serve as basis for upcoming sections. The next section introduces Machine and Deep Learning using these intuitions and provides backward references to these concepts when required.

2.3 Mathematical Introduction to ML

Now that some mathematical intuitions are built, Machine and Deep Learning can be introduced more concretely.

2.3.1 Ingredients of a Machine Learning Problem

A machine learning problem has two stages viz; estimation and generalization.

Estimation: In supervised machine learning, estimation refers to the process of learning a mapping between the input data and the corresponding output label. This mapping is represented by a **hypothesis**. The process of learning this mapping is called training, and is realized by a **training algorithm**. The subset of available data used in training is called the **training dataset**. During estimation, in order to empirically reason about the success of a hypothesis, the difference in the value of actual label and the label calculated by the learned mapping (hypothesis) is calculated for each training example. This difference is then summed, and the resulting value is an indicator of correctness for the chosen hypothesis. This calculation represents the loss, and is implemented using a **Loss Function**. A perfect hypothesis will have a zero loss.

Generalization is the ability of the learned hypothesis to correctly map inputs of **unseen** data to their correct output label. **Testing dataset** is used to evaluate the generalization or test performance of the learned hypothesis. The loss function used to evaluate testing performance is same as that in estimation.

Modern day machine learning problems are approached by casting them into the aforementioned setup. Instead of estimating the hypothesis directly, the training algorithm tries to **minimize the loss** using an optimization algorithm (e.g. Gradient Descent mentioned in section 2.2.3). Once the loss is minimized, the hypothesis that was used to calculate this loss is considered as a good approximation of input-output mappings.

Classification: One of the most discussed problem in machine learning is that of Classification. Given a dataset where each data point belong to a particular class, machine learning is applied to correctly classify the data points in their respective classes. Each of these classes can be represented by a numerical label (0,1,2 etc.). When there are a total of two classes, the machine learning classifier is called a binary classifier.

A larger part of machine learning is to be able to express a given problem in stages of estimation and generalization, and to make design choices that find the best input to output mapping.

Equipped with the knowledge of its ingredients, different machine learning formulations can now be observed in the following sections.

2.3.2 The Perceptron

The Perceptron's training algorithm and its vectorial representation detailed in this chapter are inspired from video lectures 5 and 6 by Professor Kilian Weinberger (2018, [49]).

The Perceptron is a machine learning algorithm developed at the University of Cornell by Frank Rosenblatt in the late 1950s. Perceptron was an ingenious classification algorithm that could classify data points in a binary setting, i.e. with two class labels(+1,-1).

Perceptron can be expressed using the constructs described in section 2.3.1. Amongst other available classes of hypothesis, the Perceptron uses the class of **linear hypothesis**. A linear hypothesis defines a **linear** mapping between the inputs and their corresponding labels(outputs). Mathematically, the Perceptron's hypothesis can be expressed as follows:

$$H(x; W, b) = Wx + b \tag{2.31}$$

The above equation can be read as: ' H ' is a function of x and is parameterized by ' w ' and ' b '.

For a geometric intuition, one can imagine that the datapoints of the two classes are plotted on a graph of matching dimensions. Here dimensions represent the number of values/features each data point has and **not** the total number of training examples. Once all the available training data points are plotted, the Perceptron tries to find a boundary that separates the two planes. Since Perceptron chooses a linear hypothesis, this boundary would be linear. This linear boundary is called a **separating hyperplane**.

An algebraic intuition of this hyperplane can be developed using the concepts from Linear Algebra section 2.2.1. The algorithm estimates a vector ' w ' that is orthogonal (or perpendicular) to the plane that represents a linear separation between the two classes. The term Wx is actually a **dot product** of the weight vector W and input vector ' x ', and hence gives a sense of the orientation of the input vector x with respect to vector w . According to Perceptron's algorithm, all points lying on the side of hyperplane where vector ' w ' points, are classified as $y = +1$ and vice versa as shown in figure 2.4(a). The vector b is an offset required to provide flexibility to move this hyperplane around. Absence of b , would mean that the learned hyperplane must always pass through the origin. This hyperplane can be visualized as an infinitely extending geometric object in all directions. For a 2 dimensional space, the hyperplane would be a 1 dimensional space which represents a line as shown in figure 2.4(a). In general, an n -dimensional data space would have an $n-1$ dimensional hyperplane.

Output label: Since Perceptron is a binary classifier, all the input data points should map to either of these labels. The output label ' y ' could either be $+1$ or

–1. Given an input x , its corresponding predicted label \hat{y} will be calculated as follows:

$$\hat{y} = \text{sign}(H(x; w, b)) = \text{sign}(wx + b) \quad (2.32)$$

The 'sign' in the above equation refers to the actual mathematical sign of the computation.

Interpreting the output: If the value of $Wx + b$ is positive, the input ' x ' will be classified as class $+1$ and as -1 otherwise. Thus, if the algorithm correctly classifies an input, then the product of the actual output label y and the predicted output label (\hat{y}) calculated by $\text{sign}(Wx + b)$ must be greater than zero. This is because, if the input corresponding to class label $y = -1$ is **correctly** classified, then the $\text{sign}(Wx + b)$ must be $-ve$ for this classification to happen, and the product of two negatives is a positive. The same can be argued for $y = +1$ case. Therefore, for every correct classification, the value $(y)(\text{sign}(Wx + b))$ must be **greater than zero**. Similarly, If a datapoint is **misclassified** then the product $(y)(\text{sign}(Wx + b))$ must be **less than zero**.

Training Algorithm: Perceptron's hypothesis is a linear hyperplane. In order to learn the correct hypothesis, the Perceptron must learn a hyperplane that separates the two classes. Since this hyperplane is represented by vector ' w ' and an offset b , the training algorithm attempts at learning ' w ' and ' b ' which reduces the training error. The algorithm to **learn** W and b is as follows:

1. Set $w = 0$, $b = 0$, misclassification counter; $m = 0$.
2. Iterate (loop) over all examples.
3. In each iteration do the following
 - a) Calculate the product $(y)(\text{sign}(Wx + b))$

- i. If $(y)(\text{sign}(Wx + b)) \leq 0$, then example x is misclassified. W and b must be adjusted as follows to correctly classify the example x .
 - A. $W = w + yx$ and $b = b + y$
 - B. increment misclassification counter by 1; $m = m + 1$
 - C. go to next training example
 - ii. If $(y)(\text{sign}(Wx + b)) > 0$, then example is correctly classified.
 - A. W and b need no adjustment for this training example, go to the next training example.
- b) Count the total number of misclassifications made in this run. Equals to value of m .
- i. if no error was made i.e. $m = 0$, then, the hyperplane is perfect. Exit.
 - ii. if $m > 0$, use the learned hyperplane as starting value (for w and b) and loop over entire dataset again.

The step of updating the vector W can be intuitively understood as adjusting the hyperplane so that the new linear boundary correctly classifies the current training example (or datapoint). This is done by adding the vector $y.x$ to vector ' w '. If the correct label of the point x was -1 , then the updated hyperplane vector would be $w - x$, thus shifting the linear boundary to incorporate the point in $y = -1$ space, this is shown in figure 2.5(c) and (d). When x belongs to 1, the new ' w ' becomes $w + x$ as shown in figure 2.5(a) and (b).

If the input datapoint has n features, then x and W are $n \times 1$ dimensional vectors. For a binary classifier, y is always a 2×1 dimensional vector.

Once the correct ' w ' and b are learned, the Perceptron algorithm can also be visualized as a linear transformation of an input vector x by a vector ' w ' to a vector y . Thus, the hypothesis $H(x; w, b)$ learns a linear mapping (explained in

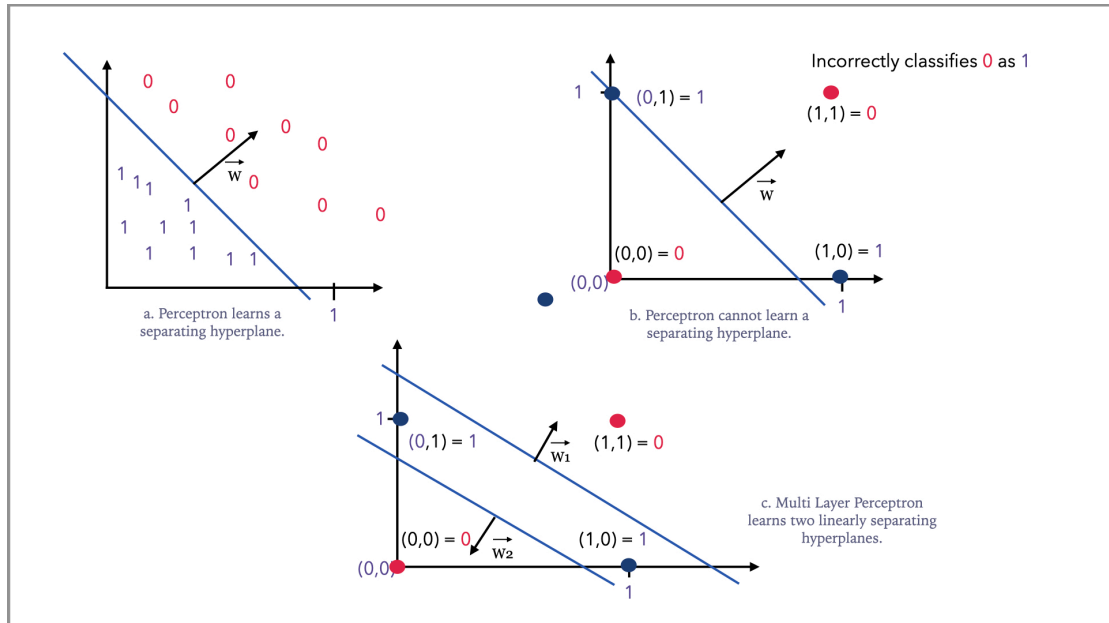


Figure 2.4: The Perceptron: a) Perceptron learns a linear boundary(blue line) between two classes of label '0' and '1'. The vector 'w' is orthogonal to that linear boundary(hyperplane). b) The XOR-DataSet: Since no 'single' linear boundary exists, the Perceptron fails to classify the samples correctly. c) MLP - Multilayer Perceptron: A handcrafted two layer Perceptron with 1 hidden and 1 input layer is capable of learning two linear boundaries and hence correctly classifies the XOR dataset. Neural Networks can learn these boundaries themselves using backpropagation.

section 2.2.1) between the input and the output vector and thus belongs to the class of linear classifiers.

The end of Perceptron: The Perceptron claims that it can learn a hyperplane that separates the two classes **provided** such a linear hyperplane exists. Perceptron instantly became successful, and was thought of as an impeccable discovery in the field of AI. A book named **Perceptron** [30] in the praise of this work was written that also mentioned that Perceptron can do great things but **cannot** learn a decision boundary between outputs of simple XOR function shown in figure 2.4(b). This is because the XOR function is not linearly separable. Perceptron's inability to estimate such a basic XOR function decreased its popularity, and led to a long period of its dormancy. Attempts at handcrafting Perceptron to specifically learn

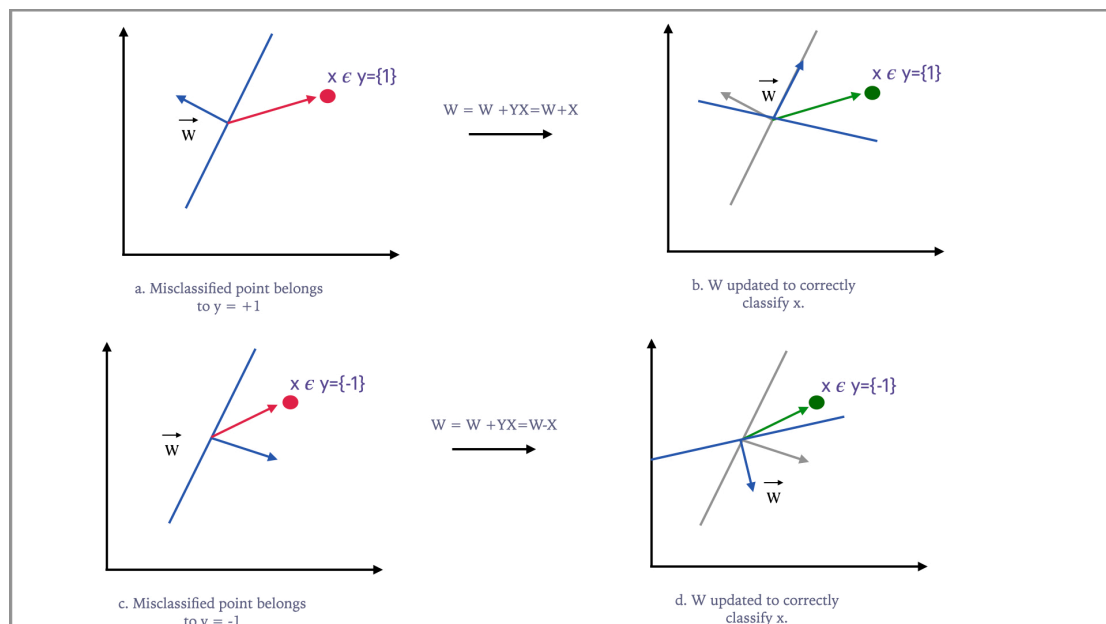


Figure 2.5: Perceptron's Training Algorithm: All points in direction of W are classified as 1. a) A point x of label $y=1$ is on the wrong side of the hyperplane and hence misclassified. b) w is updated as $w + yx$, since y is $+1$, the resulting vector is $w + x$, and can successfully classify the point. c) A point x of label $y = -1$ is misclassified. d) Since y is -1 , $w + yx$ becomes $w - x$, and now point x is on the correct side of the hyperplane.

the XOR dataset were able to learn two separate linear boundaries (shown in figure 2.4(c)), but this solution was of little importance as handcrafting features for a problem is a non-trivial task. This inability of the Perceptron was resolved by neural networks 2.3.5.

Susceptibility to Outliers: Outliers are datapoints that belong to class A but reside in the space of class B. This is a common problem when dealing with noisy data. Perceptron's algorithm will not stop until there is even a single misclassified point. Thus, it is not good at handling of outliers in dataset.

Interpreting the Output: Apart from the inability to learn a non-linear boundary, Perceptron's classification output was also critiqued. It was argued that in most practical scenarios, a simple Yes/No prediction is not the best choice. For e.g. if the weather forecasting app needs to predict two classes as rain or no-rain,

then it is better if the user gets a probabilistic estimate about the application's belief of raining that day. A desired output would be - *there is a 60% chance that it will rain today* as opposed to *It will rain today*.

Despite its shortcomings, Perceptron established firm grounds for modern day machine learning algorithms, especially for Neural Networks.

2.3.3 Feature Transformation

The intuitions presented in this chapter are derived from chapter 4 from the video lecture series by professor Leslie Kaelbling (2020, [21]).

Features refer to the different values of a single data example. They play an important role in formulating a machine learning problem. The last section concluded that Perceptron can find a hyperplane that linearly separates the data; **if** such a hyperplane exist. In practice, it is hard to find a dataset that can be linearly separated. However, feature transformation can be applied to solve this problem. It is claimed that a given dataset in d dimensions, can be represented in some higher $d + k$ dimensions, and in this higher dimension there exists a linear hyperplane that successfully separates the data. This transformation is applied to a function that operates on every feature, and thus **transforms** the entire example , and consecutively, the entire dataset to a higher dimensional space.

Figure 2.6 (a) A classification problem in one dimension. The two classes 'green' and 'red' are linearly inseparable in their original forms. A transformation to higher dimension space is thus needed to separate them linearly. The transformation is encoded using a function φ . The function 'phi' takes a one dimensional input 'x' and transforms it into a two dimensional space by calculating (x, x^2) . The new space is called a **transformed feature space**. In this transformed space, it is now possible to linearly separate the two classes, as shown in figure 2.6(b).

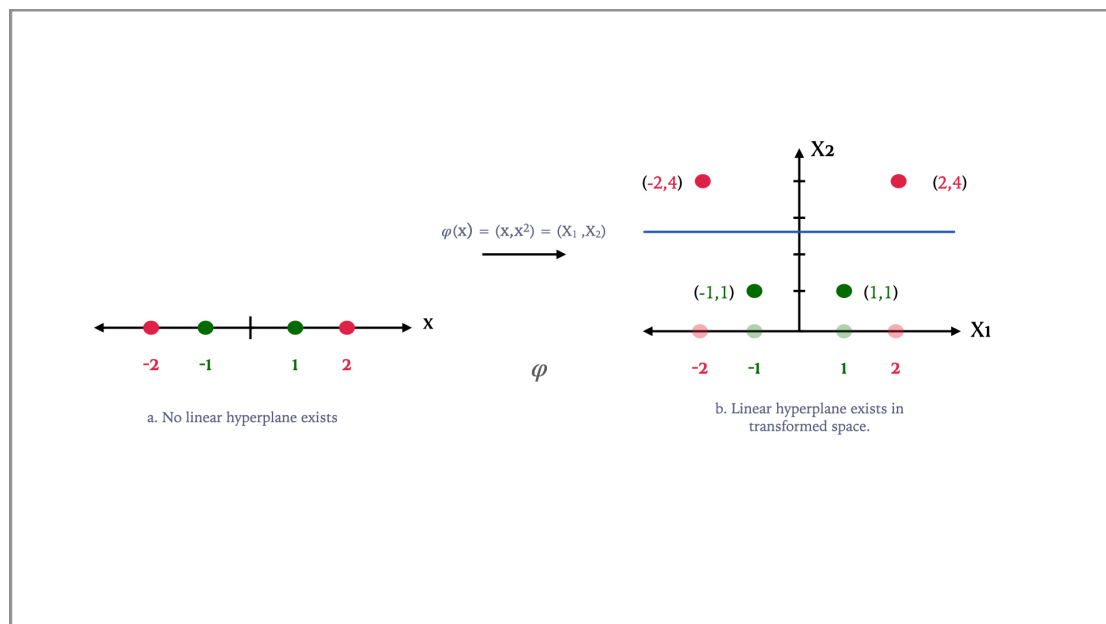


Figure 2.6: Transforming Feature Space: a) Shows a binary classification problem in 1 dimension. Data points of two classes blue and green, cannot be linearly separated. b) A feature transformation function ϕ maps the 1-dimensional space to a 2-dimensional feature space. A linear separator (blue line) now exists in this transformed space.

Finding such representations of data is essential and is studied under a facet of machine learning called Feature Engineering (earlier introduced as representational learning in section 2.1.2). Feature engineering encompasses many other tasks like making design choices about numerical representation of data or combining several features to create new features and becomes complex with the increasing dimensions of data. Many modern day Machine Learning algorithms derive their accuracy from accurate feature engineering done by domain experts. Deep Learning reduces the dependency on feature engineering with the help of Neural Networks (explained in section 2.3.5). In order to understand the working of a Neural Network, it is imperative to understand the algorithm of Logistic Regression introduced in the upcoming section.

2.3.4 Logistic Regression

This section utilizes the chapter 5 from the video lecture series by professor Leslie Kaelbling (2020, [21]).

Logistic Regression is another linear classification algorithm from the literature of statistical analysis. Like Perceptron (explained in section 2.3.2), Logistic Regression is also a **linear classifier**, however, its approach to find the linear hyperplane and the interpretation of its prediction makes it a better choice than the Perceptron.

Hypothesis: Similar to Perceptron, the hypothesis of logistic regression is also parameterized by 'w' and b and equals $wx + b$. As explained in the previous section, once the Perceptron calculates this quantity, it checks for the **sign** of the result, and predicts the label based on this sign. Logistic Regression on the other hand, computes the **sigmoid** of this term, the predicted output label ' \hat{y} ' is thus given as:

$$\hat{y} = \text{Sigmoid}(H(x; w, b)) \quad (2.33)$$

The **Sigmoid function** is defined as follows:

$$\sigma(wx + b) = \frac{1}{1 + \frac{1}{e^{(wx+b)}}} \quad (2.34)$$

For any given input, the sigmoid function outputs a value between 0 and 1. The benefit of such a transformation is that it can be interpreted as a **probability**. Figure 2.7 shows the 'S' shaped graph of the sigmoid function. Notice that for any highly negative or highly positive value of 'z' the function outputs 0 and 1 respectively, thus mapping all inputs to outputs in range 0 and 1. Further, the function transitions at value 0.5, this value can be used as a **threshold** for classification such that if $\hat{y} = \sigma(wx + b)$ is greater than 0.5, the input x belongs

to class label 1 and to class label 0 otherwise. This idea can be represented as follows:

$$\hat{y} = \begin{cases} 1, & \text{if } \sigma(H(x; w, b)) \geq 0.5 \\ 0, & \text{if } \sigma(H(x; w, b)) < 0.5 \end{cases} \quad (2.35)$$

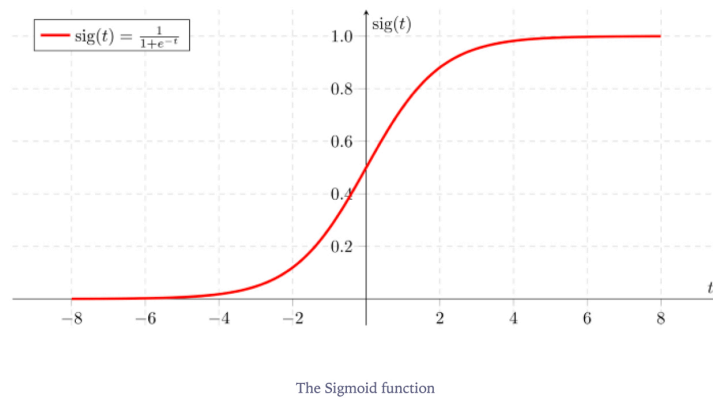


Figure 2.7: Sigmoid: The 'S' shaped Sigmoid function has two horizontal asymptotes at 1 and 0. its value at $x = 0$ is 0.5 and is used as a threshold for classification. Source: <https://towardsdatascience.com/derivative-of-the-sigmoid-function-536880cf918e>

Interpreting the output: The output of sigmoid is interpreted as the probability of the input belonging to a particular class (mostly to class 1). For e.g. if the output of $\sigma(wx + b)$ is 0.7, then the logistic classifier is 70% sure that the example x belongs to class 1. Since 0.7 is greater than the threshold value of 0.5, the classifier assigns the label (\hat{y}) '1' to input x . Similarly, if the output value is 0.1, then the classifier is only 10% sure that the example x belongs to class 1. Since being 10% sure is not enough (threshold is 50%), the input x is assigned to class 0. In order to calculate the probability of x belonging to class 0, the output

probability is subtracted from 1. This is because the sum of all probabilities must be equal to 1(explained in 2.2.2). Therefore, when the classifier outputs 0.1, the probability of x belonging to class 0 is 0.9 ($1 - 0.1$), hence the classifier is 90% sure that the input example x belongs to class 0. Extending this probabilistic intuition, the Logistic Regression classifier can be seen as an algorithm that assigns a probability to each data point. This probability can be represented as :

$$P(x_i) = \begin{cases} \sigma(H(x_i; w, b)), & \text{if actual } y = 1 \\ 1 - \sigma(H(x_i; w, b)), & \text{if actual } y = 0 \end{cases} \quad (2.36)$$

Since $\sigma(wx + b)$ is equal to the predicted label (\hat{y}), the above equation can be written as:

$$P(x_i) = \begin{cases} \hat{y}_i, & \text{if actual } y = 1 \\ 1 - \hat{y}_i, & \text{if actual } y = 0 \end{cases} \quad (2.37)$$

The above equation has two if statements, that can be combined into a single statement as follows:

$$P(x_i) = (\hat{y}_i)^{y_i} (1 - \hat{y}_i)^{(1-y_i)} \quad (2.38)$$

Since, the logistic classifier assigns these probabilities **independently** to every input datapoint, the joint probability of all the datapoints ($P(D)$) can be represented as the **product** of their individual probabilities as shown below:

$$P(D) = \prod_{i=1}^n (\hat{y}_i)^{y_i} (1 - \hat{y}_i)^{(1-y_i)} \quad (2.39)$$

Since probabilities lie between the interval 0 and 1, product of a lot of probabilities can be difficult to handle as the resulting value can be very small. To avoid this, a common trick of taking the log of the product term is used as it converts this

product into a sum. Taking log of the above equation results in the following equation.

$$P(D) = \sum_{i=1}^n (y_i \log(\hat{y}_i) + (1 - y_i)(1 - \log(\hat{y}_i))) \quad (2.40)$$

In order for the classification to be correct, it is desired that the assigned probability to each datapoint be as high as possible (at least greater than 0.5). Thus, the intention is to **maximize** the probability $P(D)$. Further, \hat{y} depends on the hypothesis, which in turn is parameterized by parameters 'w' and 'b'. Therefore, maximizing the probability $P(D)$ can be formulated as a **Maximum Likelihood Estimation** or MLE(explained in section 2.2.2.2) problem, where $P(D)$ is parameterized by 'w' and 'b'. Taking the form of MLE described in equation 2.25, MLE of $P(D)$ can be written as.

$$\text{MLE} : w, b = \text{argmax}_{(w,b)} P_{(w,b)}(Data) \quad (2.41)$$

The above equation provides an answer to the question - What is the value of parameter 'w' and 'b' for which the value $P(D)$ is maximum? Using the concepts described in 2.2.2.2, in order to calculate MLE, the partial derivative of $P(D)$ w.r.t to 'w' and 'b' is set to zero and the resulting 'w' and 'b' correspond to the maximum value for $P(D)$.

Loss Function : As described in section 2.3.1, the notion of a loss function is that it evaluates the performance of a machine learning algorithm. Loss function represents the loss and hence its desired value is zero. If the logistic classifier correctly classifies all the training examples, the $P(D)$ should be high and loss should be low. Thus, maximizing the $P(D)$ can also be formulated as **minimizing** the loss. Minimizing a given function is same as maximizing it after multiplying

by '-1'. Using this idea, the definition of P(D) in equation 2.40 is multiplied by '-1' to generate a loss function.

$$Loss(\hat{y}_i, y_i) = (-1) \sum_{i=1}^n (y_i \log(\hat{y}_i) + (1 - y_i)(1 - \log(\hat{y}_i))) \quad (2.42)$$

Intuitively, the loss function takes as input the predicted value and the actual value, and calculates the loss (or error) using the equation 2.42. This loss function of logistic classifier has multiple names viz; **log loss**, **negative log-likelihood loss function** and **cross entropy**.

Training Algorithm: Logistic Regression formulates the machine learning problem as an **optimization problem**. Optimization problem is the class of problems that have multiple possible solution and the goal of optimization problem is to find one of them (explained in section 2.2.3). In the case of logistic classifier, the optimization problem is formulated as **minimizing** the loss function. The value of 'w' and 'b' corresponding to the smallest value of this loss function is considered as the solution.

The minimum value of loss function is calculated using Gradient Descent(explained in section 2.2.3). It is possible to use gradient descent on logistic regression because the sigmoid function is **differentiable** and the resulting loss function is **convex** in nature. Since convex functions have only one minimum value, applying gradient descent results in a sufficiently optimal value of parameters 'w' and 'b'.

Logistic Regression establishes statistical bridge to machine learning and exploits probabilistic techniques to formulate machine learning as an optimization problem. This introduces possibilities of applying various kinds of optimization algorithms to find better solutions for more complex problems. Despite being a reliable classifier and offering easy extensibility to multi class classification, logistic regression is still a linear classifier and hence only works if data is linearly separa-

ble. The next section introduces Neural Networks which are capable of classifying linearly inseparable data.

2.3.5 Artificial Neural Networks: ANN

The ideas presented in this section are combined from video lectures of Professor Kilian Weinberger (2018, [49]) and from chapter 9 of the video lecture series by professor Leslie Kaelbling (2020, [21]).

Artificial Neural Networks (or Neural Networks) are capable to learn **non-linear** decision boundaries, hence are different from their linear predecessors. In order to visualize the network part of neural network, it is imperative to visualize Perceptron and Logistic Regression Classifier(LRC) as shown in the figure 2.8. Both of them take vectors of 'x' and 'w' as inputs, and compute sum of the products of their individual elements (same as taking a dot product of vector x and vector w), the result of this computation is then passed to either a sigmoid function (LRC in figure 2.8(b)) or a sign function (Perceptron in figure 2.8(a)) to compute the final output. Assuming logistic regression as a single unit of composition, a neural network can then be considered as many such units stacked on top of each other forming a single **layer** of the neural network. When multiple layers are stacked adjacent to each other, a **network** of layers is created .

Neuron: Each unit of composition of a Neural Network is called a **neuron**, and is called so after the biological neuron of the mammalian neural network. Researchers claim the biological neural network as the source of inspiration for designing the Artificial Neural Networks (or ANN); hence the name. Each neuron first calculates a dot product of input 'x' and weight vector 'w' in order to generate an output variable 'z'. The output of a single neuron 'z' then passes through an **activation function** 'f' to calculate the **activated** output 'a'. A detailed view of neuron is shown in figure 2.9. If Logistic Regression Classifier is used as a neuron, then the activation function used in the neural network would be

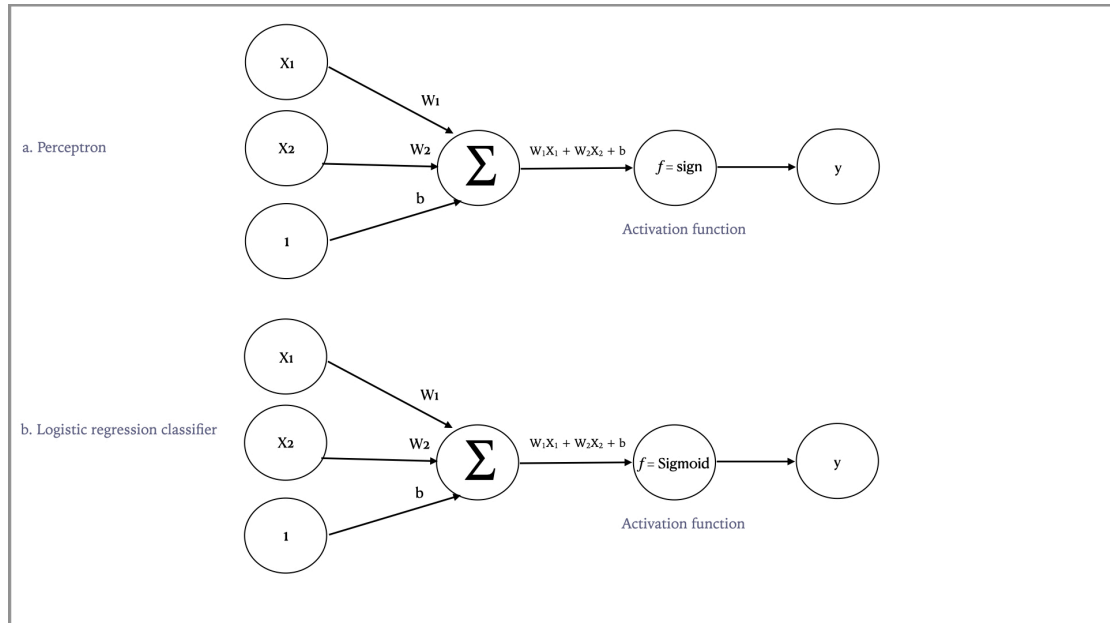


Figure 2.8: a) The Perceptron visualized as a neuron with a linear activation function. The sign function (or step function) is neither differentiable nor linear, hence cannot be an activation function for a neural network. b) Logistic regression classifier represents a single neuron of a neural network. The sigmoid activation function is non-linear and differentiable, making it a suitable choice for neural network's activation function. Both a) and b) have the offset scalar 'b' as input, this is a common practice to include 'b' as an input corresponding to 'w' value = 1.

the sigmoid function. Further, the activation function must be non-linear and differentiable. The requirement of being non-linear conforms that it brings the desired non-linearity to the neural network. Differentiability of the activation function ensures that gradient descent (detailed in section 2.2.3) can be applied to minimize the loss of the neural network.

Layers: The input and output are generally called as input and output layers, and all the layers in between them are called the **hidden layers**. Figure 2.10 shows the operations of the first hidden layer, i.e. the layer with data example as input. Considering a data example has 'm' features/dimensions, it can be represented as a $m \times 1$ dimensional vector. Corresponding to each neuron, each feature of the input has an associated 'w' value (known as its weight), which is a

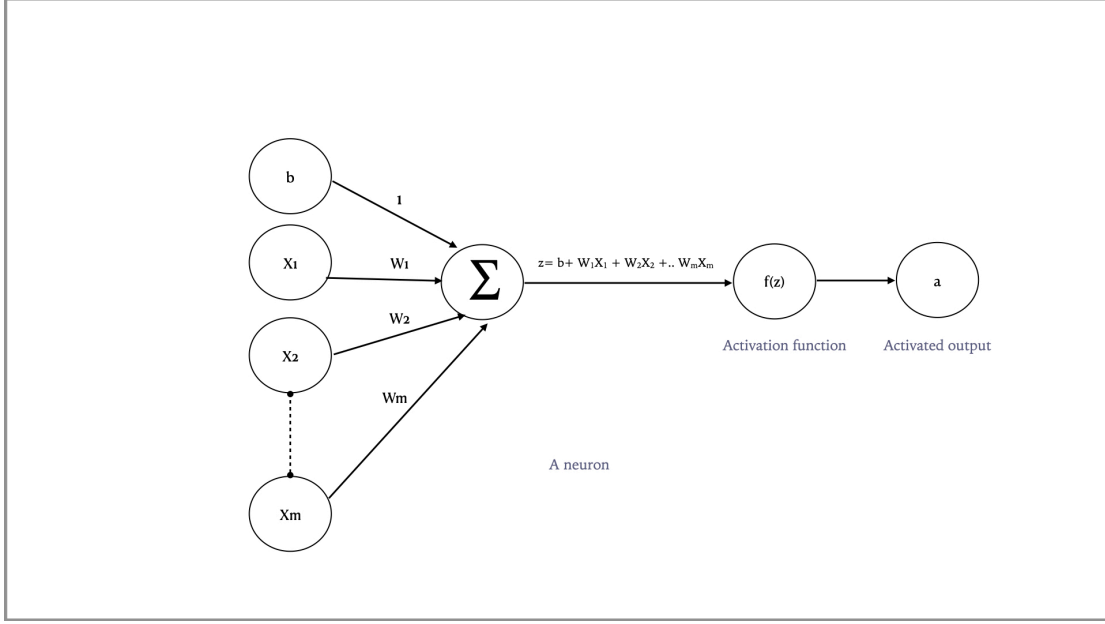


Figure 2.9: Generalized representation of a single neuron of a neural network. For an 'm' dimensional input vector, the neuron calculates a dot product of input vector with the associated weight vectors to generate a pre-activation output 'z', which is then passed through an activation function 'f' to calculate the activation of a neuron.

$m \times 1$ dimensional vector. If the hidden layer has 'n' neurons, then all the $m \times 1$ dimensional 'w' vectors can be stacked adjacent to each other resulting in an $m \times n$ dimensional matrix 'W'.

Each neuron first calculates the linear transformation 'z' of its input, which is used to calculate the activation 'a' of the neuron. The **activations** of all 'n' neuron in a layer 'l' are stacked together in an $n \times 1$ dimensional vector 'A'. The hypothesis of each neuron is $wx + b$, where 'b' is scalar value representing the offset. Stacking all these scalar values across 'n' neurons, result in a $n \times 1$ dimensional vector 'B'. Thus the final dimensions of a layer 'l' with m inputs and n neurons are as follows:

- Input vector $X_l : m \times 1$
- Weight matrix $W_l : m \times n$

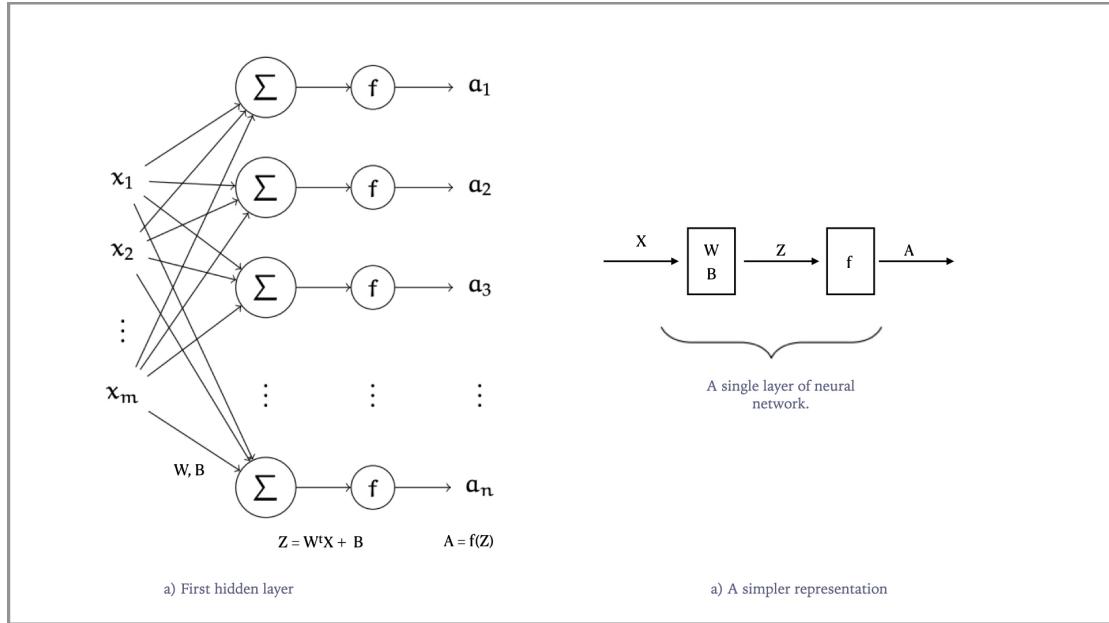


Figure 2.10: a) Shows a single of a neural network. The input is a datapoint of 'm' features. Each feature has an associated weight 'w'. All inputs and their corresponding 'w' values are vectorized into matrices X and W respectively. The pre-activation output Z is thus calculated by a matrix -vector multiplication follows by addition of intercept vector B. Final activations of all neurons are stored in vector A, which is calculated by applying the activation function on each element of Z. b) Shows a simpler representation of the same layer. Source(b): <https://openlearninglibrary.mit.edu/>

- Bias vector $B_l : n \times 1$
- Linear computation vector $Z_l : n \times 1$
- Activation vector $A_l : n \times 1$

Supposing 'f' represents an activation function acting on Z, the Activation vector A can be represented as :

$$A = f(Z) = f(W^t X + B) \quad (2.43)$$

The above equation is a valid matrix multiplication and can be verified by examples of the same described in Linear Algebra section 2.2.1. The activation function f, is applied on each element of the 'Z' vector individually. The calculated 'A' vec-

tor becomes the input for the **next** layer of neural network, and depending on how many neurons are present in that layer, the required dimensions of all matrices and vectors can be calculated. When all inputs are connected to all units (neurons) in a layer, the layer is called as a **Fully Connected Layer**. Further, the flow of computation is from input layer to the output layer, i.e. the output of layer 'l' is the input of layer 'l+1' and the output of layer 'l+1' **cannot** be the input of any previous layer 'l'. Because of this unidirectional flow of computation in the forward (from first to last layer) direction, this neural network is called as a **Feed Forward** neural network. Figure 2.11 shows all the layers of such a neural network. A neural network which allows such a feedback is then called a **Recurrent Neural Network** (RNN).

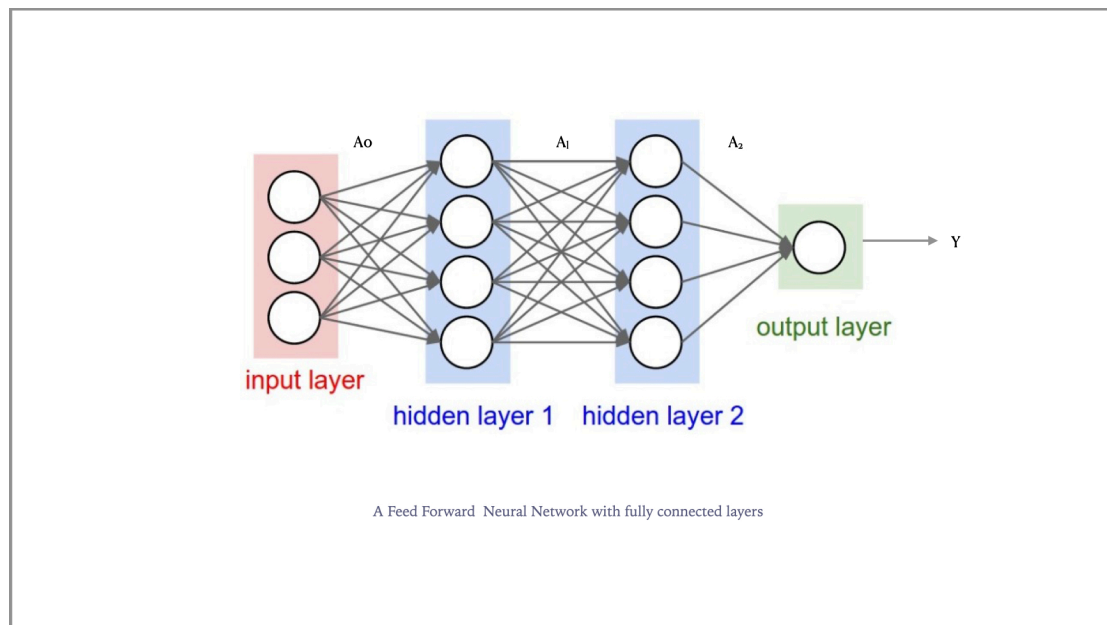


Figure 2.11: A Feed Forward neural network with two fully connected hidden layers. A_0 is a common notation to represent the input data. A_1 and A_2 show the activations of hidden layer 1 and 2 respectively. The arrows point from left to right. Considering moving to the right as moving forward, the flow of computations always flow in the forward direction i.e. A_2 **cannot** be an input to the first hidden layer. Source: <https://cs231n.github.io/neural-networks-1/>

Loss Function: Since each neuron in a layer has its own loss function (log loss of LRC), the total loss of a layer is the sum of individual losses of its neurons. Similarly, the loss of entire neural network is the combined loss of its layers. In order to calculate the loss for a single training example, computations must be done from leftmost to the rightmost layer. This computation is called as a **forward pass**. A forward pass of an input x_i calculates the corresponding predicted label \hat{y}_i for that input. This predicted label is then used to calculate the loss for this training example.

Training: Neural Networks are also trained using the basic principle of gradient descent which is also used to train the Logistic Regression Classifier(LRC). In LRC, the gradient (derivative) of a loss function w.r.t. a 'w' value indicates how much the 'w' value affects the loss function, and gives an estimate on how should 'w' be changed so that this loss is minimized. In a Neural Network, the loss function is calculated after a forward pass is done for an input. This loss function directly depends on the weights associated with the last layer and indirectly to the weights in all other layers. In order to optimize weights in all layers, their gradient w.r.t. to the loss function must be calculated. The algorithm encompassing all these gradient calculations of individual gradients is known as **Back Propagation** or **Error Back Propagation**. Using the **chain rule** from calculus (2.2.3), back propagation devises a strategy to find gradient of 'w' values of each layer w.r.t the loss function, this calculation reveals how the weights of each layer affect the Loss function. Once all the gradients are calculated, a single step of gradient descent for optimizing the neural network is taken. The calculation of gradients starts from the last layer of the neural network and goes **backwards** till the first layer. Since the flow of computation is **backwards**, and the algorithm can be thought of as propagating the 'error' or loss through the network, it is rightly named as error back propagation. Figure 2.12 shows a back propagation using

a simplistic notation of neural network. Sathyanarayana (2014, [45]) provides a gentle introduction to backpropagation.

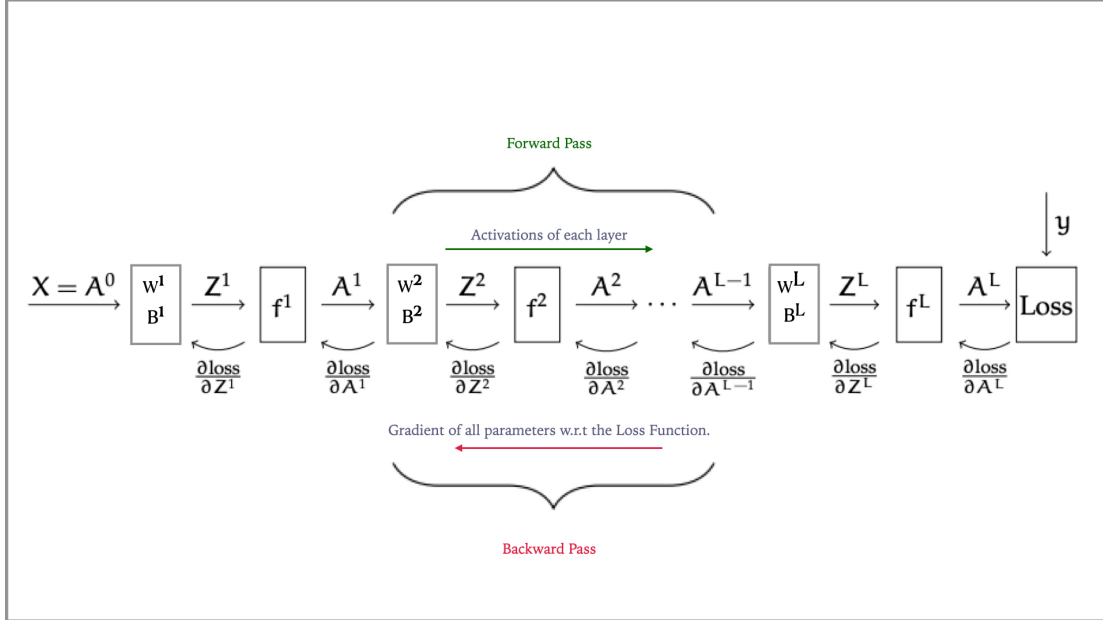


Figure 2.12: Back propagation in a neural network. Loss is calculated using a forward pass. In order to know how much the weights of all layers need to be updated, their gradient w.r.t. the loss must be calculated in a backward pass. Adapted from source: <https://openlearninglibrary.mit.edu/courses/course-v1:MITx+6.036+1T2019/course/>

Multi Class classification: Often Neural Networks employed for classification tasks, perform their inference on multiple classes as opposed to two in the case of a binary classifier. Logistic regression is extensible in binary case by using One-versus-All method, which splits the multi-class classification problem into subsets of binary classification problems. In a multi-class case, the output of the neural network must give probability estimates of all the classes. The input example is then classified as the class with highest probability. Therefore, for 'm' different classes, the output layer must have 'm' neurons. Further, the sum of all output probabilities must be 1 for each input. Such an output can be achieved by using a **softmax** function [33]. Softmax is an activation function that takes the

pre-activation output (z) of the last hidden layer (also called logits)and assigns probabilities to each value of the vector such that their sum is 1.

Choice of Activation Functions: Sigmoid was the initial choice to train the hidden layer of Neural Networks, until another activation function named **Rectified Linear Unit** or 'ReLU' was introduced. 'ReLU' provides faster convergence (finding a minima) which made it an industry standard and the most preferred activation function. Sigmoid, however is still required in the output layer of a binary classifier. Nwankpa et al. (2018, [33]) compares contemporary activation functions and provide an analysis of their adoption in Deep Learning applications.

Neural Networks are Universal Approximators: A mathematical construct is said to be universal estimator if it can approximate any function in the universe. Approximation of a function $f(x)$ can be defined in terms of its approximated function $g(x)$ and a very small value ϵ as follows:

$$f(x) - g(x) = \epsilon \quad (2.44)$$

The above equation entails that the difference between the true function $f(x)$ and its approximation $g(x)$ must be a very small insignificant value ϵ , which means that the approximated function is similar to the true function.

Neural Network can approximate any function. Cybenko(1989, [6]) provide theoretical intuitions and prove that Artificial Neural Networks are indeed universal approximators. While this theoretical proof holds, finding a good approximation in practice is still a challenging problem. Increasing neurons and adding more layers are common practices to improve these approximations, however such alterations can also introduce new problems. The features of a neural network such as the number of neurons in a layer, number of total layers, and the step-size of gradient descent algorithm etc., are known as **Hyperparameters**. The process of finding the right value for each hyperparameter is called as **Hyperparameter Tuning**.

Being universal approximators, Neural Networks can learn any complex non-linear function. One way of visualizing the learned non-linear function is as a combination of specific parts of all the functions learned in each hidden layer of the neural network. Thus, the computations and boundaries learned in the first hidden layer, are reused to perform new computations and learn complex boundaries using the previous layers as building blocks. This composition of simpler representations into complex ones is referred to as the Representational Composeability by Deep Learning in section 2.1.2.

There are many flavours of Artificial Neural Network that are adapted to suit the domain they are applied to. One of such domains is that of image analysis for which a particular Neural Network is crafted and is called the **Convolutional Neural Network** (or CNN). The upcoming section is dedicated to CNN.

2.3.6 Convolutional Neural Network: CNN

This section is largely adapted from chapter 9 of the video lecture series by Professor Leslie Kaelbling (2020, [21]).

Convolutional Neural Networks (**CNN**) are amongst the most successful variants of Neural Networks and they are specifically designed to perform on image processing tasks. The word Convolutional is indicative of the **convolution** operation that is the first operation performed on an image fed to a CNN. A black and white (also known as greyscale image) image is represented as a matrix of pixels, with each pixel value depicting a shade of grey using a range of values between 0 (white) and 255 (black). A coloured image (also known as RGB image) generally has three such matrices corresponding to a similar scale of Red, Green and Blue. Such an image is said to have three **channels**. For ease of discussion, a greyscale image is a good reference as the same intuitions can be easily extended to coloured images.

Convolution Operation: A convolution operation in its essence is similar to a dot product in many ways. The convolution operation is done between a vector of pixel values from a sub-section of the image and the vector of elements of an object called a **filter**. To provide a simpler visualization, let us consider a one dimensional image which is represented by a vector of pixels whose values could be either '0' or '1', such an image is shown in figure 2.13. The figure 2.13 shows a convolution operation across all the pixels of the 1-Dimensional image using a filter of length 3. The filter takes a dot product with pixel vectors and spans across the entire image while doing so. The resulting image vector is shown in figure 2.13(c). The convolution operation is a linear operation and is followed by a transformation by using the **Rectified Linear Unit** function or ReLU (figure 2.13(d)). ReLU is an activation function that outputs the $\max(0, \text{input})$. Thus if the input to the ReLU function is a negative value, it outputs 0, otherwise, it outputs the input.

Filter as a Feature Detector: The filter used in figure 2.13 consists of three values $[-1, +1, -1]$. When the filter is moved across the image, its each step can be visualized as evaluating the neighbourhood of a single pixel by **centering** the filter on that particular pixel. The result of a convolution operation using this particular filter would be equal to 1 only when the pixel vector is equal to $[0, 1, 0]$, i.e. the neighbours of the central pixel are zero. Thus, the filter is specifically looking for **isolated** pixels in a given image, i.e. pixels whose neighbours do not have any image content. The convolved image vector is then passed through the ReLU function which only outputs values that are greater than zero (i.e. all 1 values of this example) which is indicative of filter's findings. Similarly, different filters can be used to find other kinds of **features** in an image such as presence of an edge (left edge or right edge) etc.

Two dimensional Images: A 2-D image works well with a 2-D filter. Figure 2.14 shows convolution of a 2-D image by a 2-D filter. Every single operation can

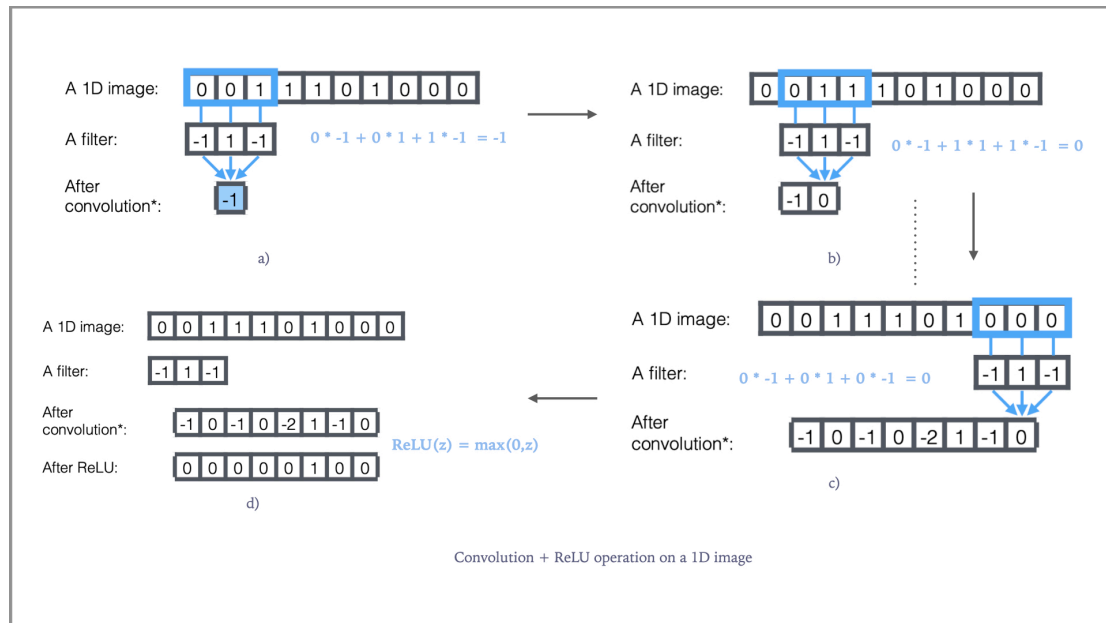


Figure 2.13: Convolution operation using a 1-D filter on a 1-D image: a) First Convolution operation: Shows the starting position of the filter. Every operation calculates the dot product between the filter vector and the pixel vector (shown in blue). b) Convolution operation on the second position: The filter now moves right by one pixel and convolves over this set of pixels. c) The last convolution operation. The original image now has a new representation after the convolution operation. d) ReLU of the convolved image. ReLU outputs $\max(0, \text{pixel-value})$. Adapted from source: <https://openlearninglibrary.mit.edu/courses/course-v1:MITx+6.036+1T2019/course/>

now be considered as a dot product of the filter and pixel matrices. The final image in figure 2.14(d) shows the resulting convolved image. It is evident that the final image has lesser dimensions (3×3) than the original image (5×5). Such a reduction is undesirable because a CNN has more than one convolutional layer and a subsequent reduction in each layer can cause the final image to be very small. To resolve this shrinking problem, a technique called **padding** is applied. Figure 2.15 shows a padded version of the input image from figure 2.14, the image from 2.14 is extended by adding pixels of value 'zero'. Such a padding with zero-valued pixels is known as **zero-padding**. Due to this padding, the final convolved image now has the same dimensions as the original image. Similar to the 1-D example,

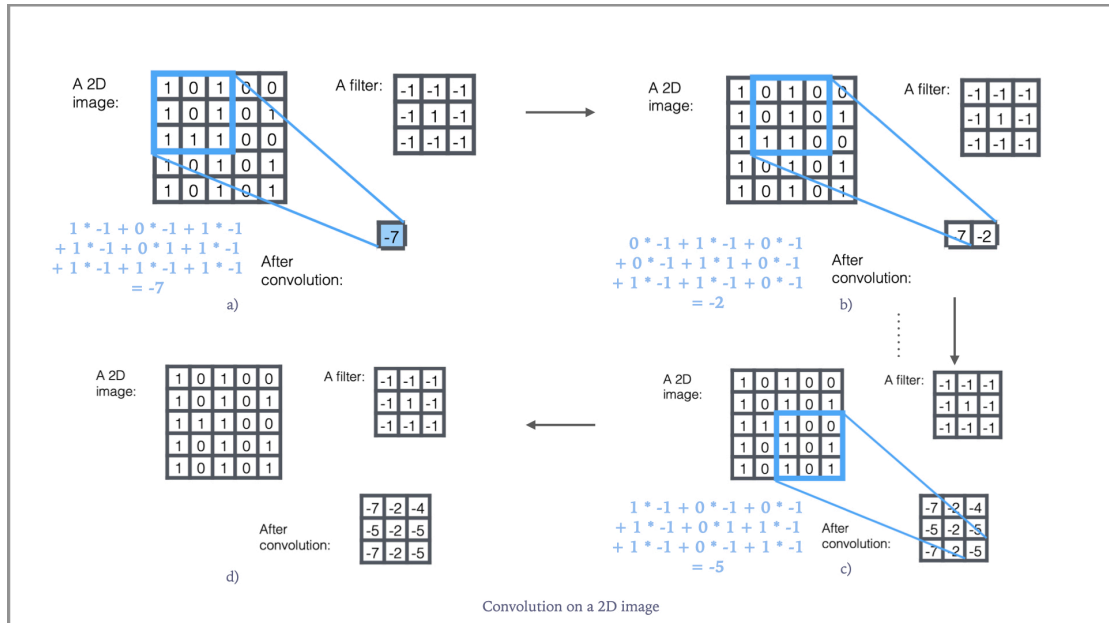


Figure 2.14: Convolution operation using a 2-D filter on a 2-D image: a) First Convolution operation: Shows the starting position of the filter. Every operation calculates the dot product between the filter matrix and the pixel matrix (shown in blue). b) Second Convolution operation: The filter now moves right by one row and convolves over this set of pixels. c) Last convolution operation. d) The final representation of the convolved image. Adapted from source: <https://openlearninglibrary.mit.edu/courses/course-v1:MITx+6.036+1T2019/course/>

a ReLU operation is applied to the final convolved image but is not shown in the figure.

Pooling Layer: The features extracted by the convolution filter are detected relative to their original location in the image, i.e the detected feature's position in the convolved vector would change according to its corresponding position in the image. Thus if the object of interest is slightly moved, its convoluted vector will also be different. This entails that the convolution operation is susceptible to slight modification(s) in the location of the input feature. This property is undesirable because an object can have two images and in one of the images the object could be slightly shifted to the right/left. In such a scenario, it is desirable that a single convolution filter (that detects features of the given object) should be able to detect

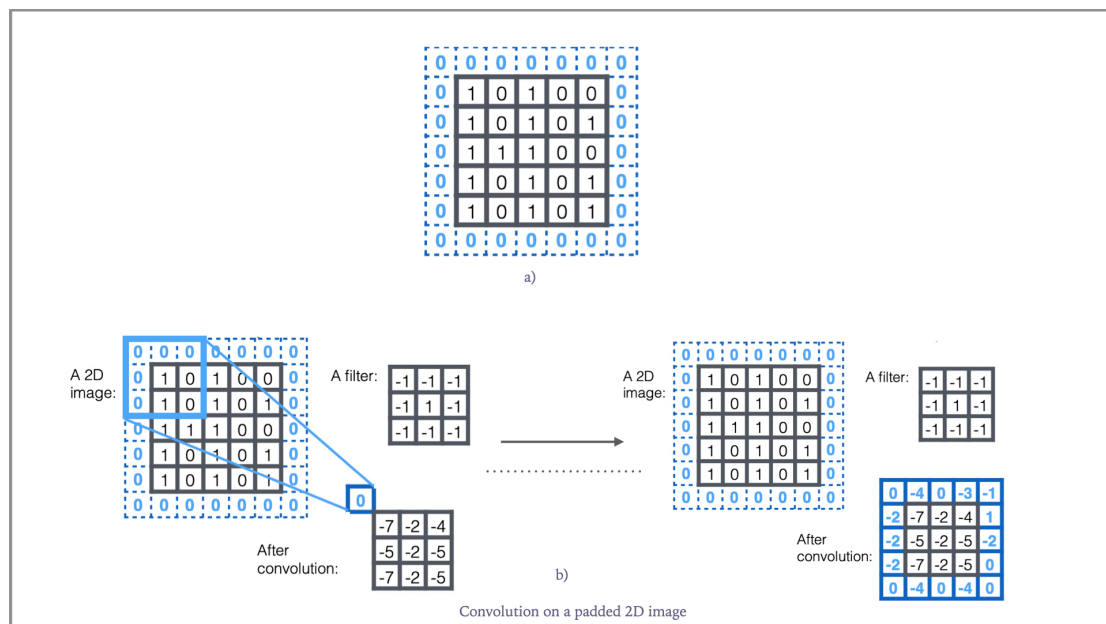


Figure 2.15: Convolution operation on a padded image: a) Shows the image from figure 2.14 after zero padding. b) Shows the additional values added in the previously calculated convoluted value. Adapted from source: <https://openlearninglibrary.mit.edu/courses/course-v1:MITx+6.036+1T2019/course/>

the object's features in both images. The ability of a network to handle such a slight shift is called as **Translational Invariance**. Pooling layers are introduced to make CNN translation invariant. One variant of pooling layer is called as max-pooling layer, where a 2D filter is grazed across the convoluted (and *ReLU'ed*) output and the max-pooling operation outputs the **maximum** pixel value in the corresponding pixel matrix. Pooling aims at reducing the spatial (height and width) dimensions of the convoluted output. This is done by summarizing the neighbourhood (e.g. by taking max or average) of each value in the convoluted output, and thus slight variations in the input image do not affect the final output obtained after pooling operation.

Once several convolutions and pooling operations are performed, the resulting image representation is of lower dimensions but has the granularity of complex features as opposed to that of independent pixels in the input image. Each convo-

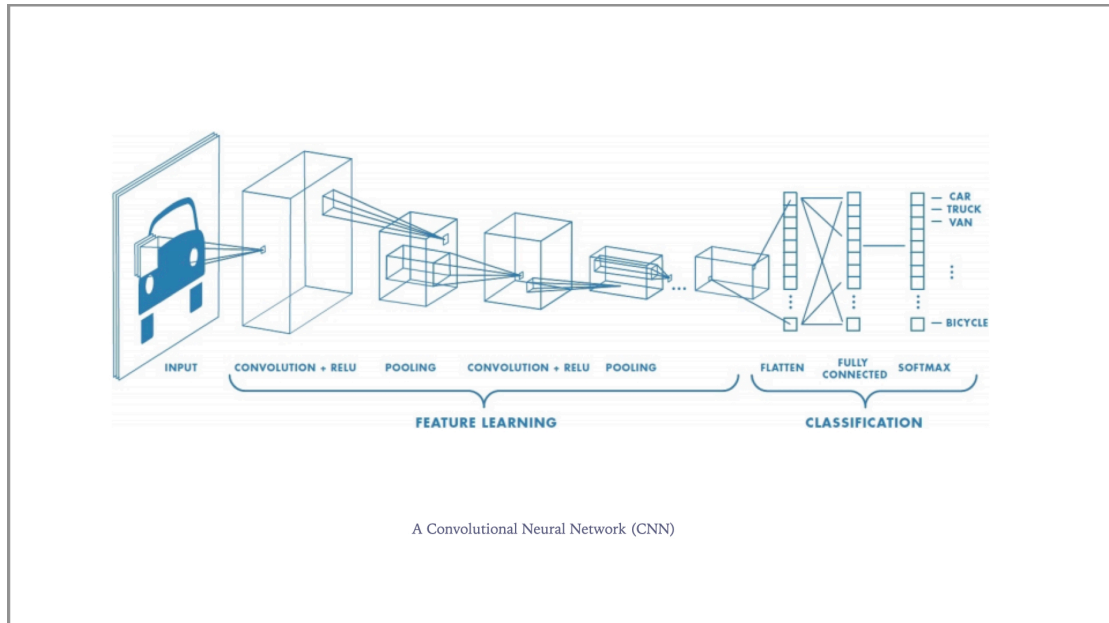


Figure 2.16: A CNN with all layers. The CNN has two sets of Convolution, ReLU and Pooling layers. These two sets represent the feature extraction process. The fully connected layer is later used for classification and takes the flattened (unrolling a matrix into a vector) output of last pooling layer as input. Source: <https://www.mathworks.com/solutions/deep-learning/convolutional-neural-network.html>

lution, ReLU and pooling operation can be understood as extracting features from the image, whose complexity increases as more layers are added. A visual representation of this idea can be seen in image 2.1. Thus, the resulting representation of an input image is composed of simple but meaningful features present in the image. Hence, CNN (Deep Learning) provides **representational composeability**(explained in section 2.1.2).

Fully Connected Layer: Once the features are represented, a fully connected layer (explained in section 2.3.5) is implemented and the learned features are used as input to this layer. The fully connected layer is simply the vanilla neural network explained in section 2.3.5. For a multi class classification problem, a softmax function is to be used in the last layer of the CNN. Similarly, for a binary classification a sigmoid function would suffice.

Parameters: The CNN uses the same underlying training algorithm of gradient descent and back propagation. There are many handcrafted feature detectors (another name for convolution filters) such as line or edge detectors but the main aim of CNN is to **learn** the filters required for a specific problem during training. Thus, apart from the weights of the fully connected layer, the CNN also learns the filters used in the convolutional operation. The size of the filter however is a design choice and cannot be learned.

Figure 2.16 shows an example of a CNN with two sets of convolution (followed by ReLU) and pooling layers. After the feature extraction stage, a fully connected layer of neural network is used to perform classification. The output of classification is interpreted using a softmax function (multi-class classifier).

Absence of CNN would mean directly feeding an input image to a fully connected layer, this might work in theory but would be extremely expensive to train. This is because for a simple image of 640×480 resolution, a total of 307200 (corresponding to each pixel input) weights in the first layer would be required. CNN on the other hand convolve over the raw inputs using a $k \times k$ filter. A common choice of k is 3, hence only a total of 9 weights are to be learned for the first layer as opposed to 307200 in the case of a classical Neural Networks. This reduction in training cost has further added to the popularity of CNN.

Both ANN and CNN are designed to work on data that resides in a euclidian space. In other words, for data that can be faithfully (without loss of information) represented in a coordinate system of choice, ANN and CNN provide efficient mechanism to solve any objective function. However, many real world entities and their interactions cannot be represented faithfully in any euclidian coordinate system. Such entities are said to exist in a non-euclidian space. Graph Neural Networks or GNN are specially crafted neural networks that can work on such non-euclidian data without the need of a lossy euclidian representation. The

following section uses the concepts of ANN and CNN to build an understanding about Graph Neural Networks.

2.3.7 Graph Neural Networks

The presented ideas around Graph Neural Networks are collated from lecture 8 of video lecture series presented at Stanford University(2019, [20]).

Graphs are an abstract data structure that can be used to model almost any real world entity/situation. A graph structure consists of nodes and edges. Nodes in a graph are usually representative of entities and edges signify connections between such entities. A single node can be connected to more than one nodes and the connected nodes are referred to as the **neighbours** of the node in question. Figure 2.17(a) shows structure of a basic graph. To visualize representation of a real world application by graphs, one can imagine the inner workings of a social networking platform on which a user is connected to other users through a notion of connection (for e.g. through sending connection requests etc.). Such a relationship (connections) between two entities (users) can be modelled with graphs, where nodes of a graph are different users and edges emerging from a user connect her with other users of choice. Connections could be of many types, for e.g. user '1' ***follows*** user '2', or user '1' ***is friends with*** user '2' etc. Such information about connections can be encoded in the edges of the graph. Once such a graph structure is created, graph specific algorithms can be applied to learn the nature (the target problem defines this nature) of a particular user based on the type of users it is connected with or more specifically, its neighbours.

Graph represents Non-Euclidean space: The graph structure explained above, inherently exists in a 3-Dimensional setting where there is no notion of a coordinate system. The above example of a graph representing a social network cannot be **faithfully** represented using a defined euclidian coordinate system. Any attempts at such representations will result in a lossy representation at a

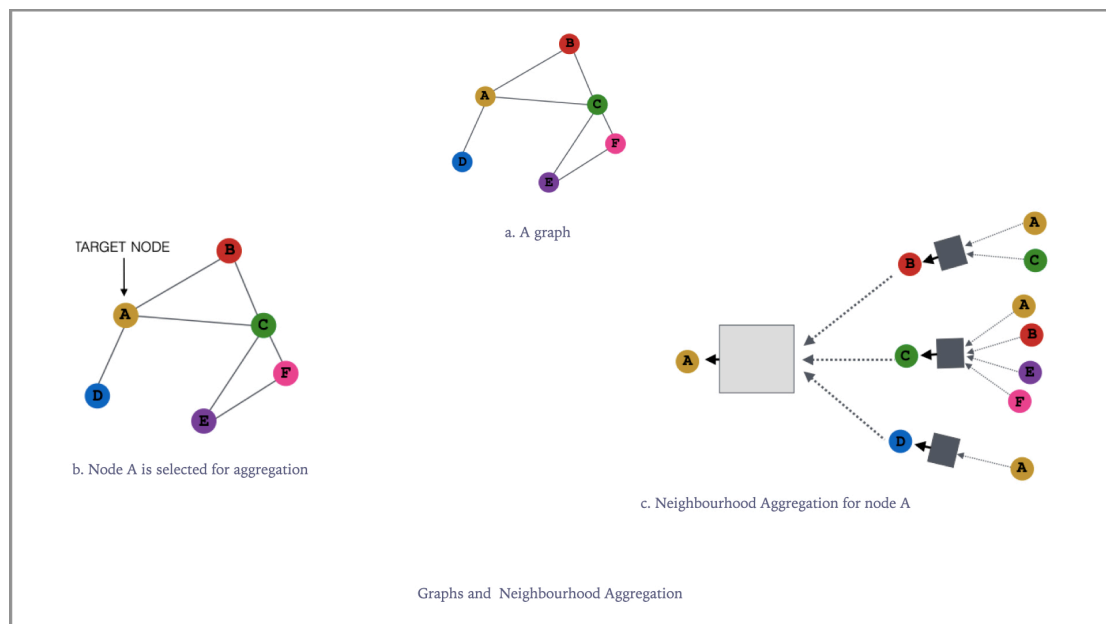


Figure 2.17: Graphs and Neighbourhood Aggregation : a) An arbitrary graph with 6 nodes (A-F) connected with edges. b) Node A is the target node selected for this example. c) Shows neighbourhood aggregation for node A. The computation graph is created using the neighbourhood of A. Node B,C and D are direct neighbours of A and hence are connected directly in the computation graph. Nodes B,C and D further have their own computation graph where the node A is also an input. Image adapted from source: <http://web.stanford.edu/class/cs224w/slides/08-GNN.pdf>

cost of losing the graph's hidden structural information. It is therefore desired to keep the graph structure as it is, and develop algorithms that makes use of hidden information in these structures.

Graph Neural Networks: Graph Neural Networks (GNN) or Graph Convolutional Network (GCN) can be understood as a generalization of neural networks to graph data structures (Jie Zhou et al., 2019 [53]). Traditional neural networks (and CNNs) are trained to work on euclidian data (e.g. 2D images), therefore to treat graphs as they are, an extension of neural network to work on non-euclidian structures is necessary. That is, given an input graph and its corresponding ground truth, a GCN should learn to alter the input graph structure to match the ground truth. Ground truth can be either a complete graph itself, or more specifically

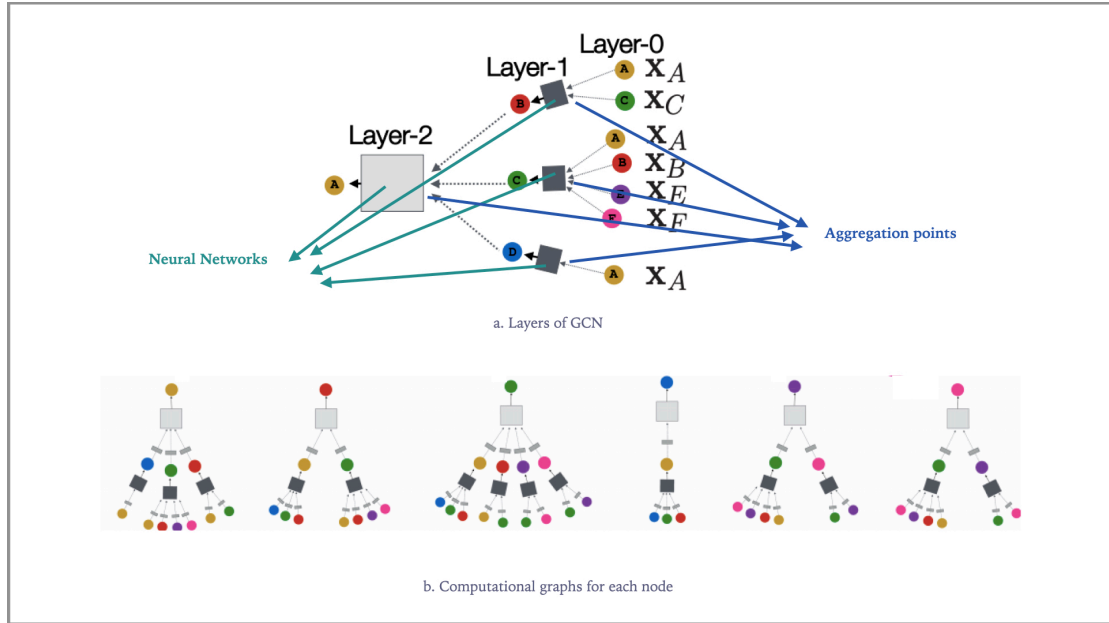


Figure 2.18: Graph Neural Networks : a) Deconstruction of computation graph of node A (previously shown in figure 2.17(c)). Each aggregation operation is followed by a neural network composition to learn relevance of individual contributions by neighbouring nodes. Layer-0 is analogous to input layer of a traditional neural network and Layer-1 and Layer-2 are representative of hidden layers. Layer-1 takes node vectors (input value of each node) as input. b) Every node of the graph creates a computation graph and results in a similar 2-Layer GCN. Analogous to a neural network weights across each layer are shared amongst all computation graphs. Image adapted from source: <http://web.stanford.edu/class/cs224w/slides/08-GNN.pdf>

could be a graph's edge or node. Thus given an input graph, a trained GCN can either predict a complete graph or predict values for edges and node features which is also known as link prediction and node labelling respectively.

Convolutions in GCN: The idea of convolutions is particularly interesting in GCNs, this is because, in a traditional CNN2.3.6, the convolution operation refers to the process of scanning a 2D euclidian pixel space (arranged in 2D pixel coordinate) with a filter/kernel. Each convolution operation represents multiplication of kernel matrix and pixels matrix of similar dimensions. In terms of the central pixel targeted by the convolution filter, the convolution operation can be defined as an **aggregation** of the central pixel with its upper, lower, left and right

neighbouring pixel. Thus convolution operation for a given central pixel takes into account its immediate neighbours. In a graph structure, there is no idea of a coordinate system, or the notion of up/down and left/right. Graphs operate on the notion of neighbours and neighbourhood, consequently, the idea of convolution in CNN **cannot** be directly applied to graphs.

Convolutions in GNN is realized for each node by collecting information from its neighbours and creating a new representation for the node using some predefined rule of aggregating this information. For e.g. one rule of aggregation could be to take an average of the sum of information (assuming some vector encoding node features) passed by a node's neighbour to the node. This process is known as **message passing** or **neighbourhood aggregation**. Figure 2.17(c) shows neighbourhood aggregation for node A. The order of neighbourhood aggregation defines a **computation graph** for each node. Each node has its own computational graph as shown in figure 2.18(b).

Neural Networks in GCN: While the rules of aggregation are pre-defined, the magnitude by which each neighbour's information is affecting the final representation (such that it matches the ground truth) is difficult to approximate. For e.g. if a node (with two neighbours) in an input graph does neighbourhood aggregation to match values in some ground truth graph, it is possible that value of one of its neighbouring nodes affects the output more than the other neighbouring node. A **weight** can therefore be assigned corresponding to each node's value and the same can be **learned** using a fully-connected feed-forward neural network (explained in section 2.3.5). Similarly, weights can also be assigned to all edges and their target representation could be reached from the input graph once such weights are learned. Figure 2.18(a) captures this intuition and shows aggregation points and neural network for node A's computation graph (shown in figure 2.17(c)). Aggregation points can consist of functions like sum or average, and the boxes represent fully connected neural networks of varying complexity.

Figure 2.18(a) captures the notion of layers in a GCN. Each node's computation graph (figure 2.18(b)) has two hidden layers (layer 1 and 2) and one input layer (layer 0). The weights across all corresponding layers are **shared**. Thus, layer-1 of computational graphs of all nodes share the same weights. Similarly, weights across all the second layers (layer-2) are also shared.

Geometric Deep Learning: Graph neural networks are studied under the recent field of **Geometric Deep Learning**. Graphs and their related concepts are a part of a field called Graph Theory which is studied in both mathematics and theoretical computer science. Geometric deep learning explores and exploits the relationship between Deep Learning and Graph Theory using a family of neural networks (GNN) that are designed to work on Non-Euclidean data.

Graphs are applicable in multiple domains and their generalization to neural networks is a recent phenomenon. Geerts et al. (2020 [12]), Zonghan Wu et al. (2020, [50]) and Jie Zhou et al. (2019 [53]) provide a comprehensive survey of contemporary approaches in Graph Neural Networks.

All previous sections establish general principles of ANN, CNN and GNN. This thesis presents an existing approach that applies these constructs to the field of Computer Vision. The next section introduces Computer Vision and enables the reader with the required vocabulary and concepts to understand the implemented approach.

2.4 Specific Topics in Computer Vision

Computer Vision is often described as sub field of Artificial Intelligence. Its purpose being extracting semantic information from images. It can also be described as a related field of Computer Graphics. While Computer Graphics deals with capturing the 3-Dimensional world on a 2-Dimensional image (both digital and physical), computer vision employs techniques to derive semantic meaning from these images. Example of typical tasks in Computer Vision could be object de-

tection, image segmentation and reconstructing the 3D scene using a 2D image. Object detection refers to the task of identifying and localizing a particular object of interest in a given digital image. Identification answers the question of whether the object is present in the image, whereas localization gives the location of the object in a given image. In practice, the localization algorithm outputs coordinates of a bounding box (a rectangle encompassing the object of interest). These coordinates are represented in pixel coordinates of the image.

Pixel Coordinates: A digital image is a collection of pixels. In order to refer a particular pixel in an image, a pixel-coordinate system is used. The origin of this coordinate system is generally on the bottom left (when the image is viewed from front).

The 3D World: The objects whose image is created using a digital camera are mostly 3D in nature and exist in a real 3D world. This 3D world can be represented by a 3D coordinate system. Given a 3D object, the process of creating its 2D digital image can be understood as a transformation of its 3D geometry (originally referenced by a 3D world coordinate system) to a 2D pixel coordinate system. Using the intuitions of matrices and vectors from Linear Algebra section (2.2.1), a point on the real world object can be represented as a 3×1 vector. The point's transformed location in the 2D pixel coordinate system is then a 2×1 dimensional vector. Such a vector transformation can be encoded in a matrix that can perform a series of desired transformations. Once such matrix is calculated for a given setting, any arbitrary point in the 3D world can be successfully mapped to its 2D location in the image. Such a transformation is a result of a projection of 3D world on a 2D image and is therefore called as a **Projective Transformation**. The matrix used for encoding such a transformation is known as the **camera projection matrix** and is commonly represented using the symbol (π).

Projective Transformation: A dimension of the 3D world is lost when it is tricked into a 2D picture. The squares and circles amongst other planar objects

lose their essence in the 2D representation of a 3D space. What maps such planar objects to a picture is an instance of a Projective Transformation. Almost all geometric properties of objects are lost when they undergo projective transformation. For e.g. a circle becomes an ellipse, angles do not remain same and even the ratio of distances is not preserved. What remains of these geometric objects is their property of straightness. This preserved property becomes a general guideline for projective transformation. Therefore, any mappings of the points of a given plane is called its projective transformation if all the straight lines are preserved. i.e they are still straight. These transformations are studied under the field of **Projective Geometry**.

Why not Euclidean Geometry?: The classical euclidean geometry cannot be used to study projective transformation. This is because Euclid's lemma that two parallel lines meet at infinity is inherently contradictory, as it is starkly opposed by another, more convincing dictum, that 'infinity' does not exist! Leaving no room for such abstract ideas in science, researchers suggested a more formal way of defining what infinity is in this context. A new geometric object or space is created by extending the regular euclidean space. The two parallel lines are now said to meet at what are called 'ideal points', which lie in this extended space. This extension of the euclidean space where this intersection happens is called the **Projective space**. In essence, the claim is that the point in euclidean space where parallel lines meet; the point at infinity, is now being represented by the ideal point in projective space. In a 2D euclidean space, a point 'P' is represented by coordinates (x,y) . In order to represent this point in the coordinates of the projective space, a third coordinate is added to 'P', and it becomes $(x,y,1)$. This is a convenient representation as the point(s) in projective space can be converted to euclidean space by taking the first two coordinates of projective space point represented by $(x,y,1)$ and vice versa. However, this transformation is only valid if the third coordinate of the projective space coordinate is 1. In scenarios

where a projective space coordinate is represented by (x, y, k) , the corresponding euclidian point is calculated by first dividing all three values of projective space point by 'k'. The resulting coordinates $(x/k, y/k, 1)$ are then used to derive the point's euclidian counter part which is given by $(x/k, y/k)$. Extending this idea further, a set of points $[(x, y, 1), (2x, 2y, 2) \dots (kx, ky, k)]$ lying in projective space, represent a **single** point (x,y) in euclidean space. This is why these points are called homogeneous coordinates of point (x,y) . Intuitively, a point in 2D euclidean space forms a line in its projective space. Thus, in the real world, two parallel lines appear to meet at a distant point, but in their projected image this distant point becomes a line. This section on euclidian and projective geometry is adapted from Hartley and Zisserman's *Multiple View Geometry* (chapter-1, [18]).

Intrinsic and Extrinsic Matrix: The transformation that the projection matrix encodes from world to pixel coordinates, can be divided in two stages. First, the real world 3D coordinate system is mapped to another a 3D coordinate system called the camera coordinate system. A camera coordinate system has its origin centered on the optical axis of the camera. The second transformation maps all points in the camera coordinate system to the pixel coordinate of the digital image. The matrix performing the second transformation depends on the type of camera and its focal length etc. Since these properties are intrinsic to the camera, the matrix encompassing them is called an **Intrinsic Matrix**. The Extrinsic matrix transforms the world coordinates to the camera coordinates and is called extrinsic because it only depends on factors which are external to camera such as its pose and location in the chosen world coordinate. For additional details on the projection matrix, David A. Forsyth and Jean Ponce' *Computer Vision: A Modern Approach* (1.2 & 1.3, [11]) can be referred.

The literature of Computer Vision details many native algorithms for extracting semantic meaning from images. In the last decade, the advances in deep learning have further empowered such algorithms by generalizing them to multiple use

cases. Niall O’ Mahony et al.(2019, [26]) draws a comparison between classical Computer Vision techniques and modern day Deep Learning aided Computer Vision.

The upcoming section detail object detection algorithms that have arrived in recent years by Deep Learning aided Computer Vision. It also establishes context for the 3D object detection approach evaluated in this thesis.

2.5 Previous work

Most relevant to this thesis, is the object detection task of computer vision. Object detection consists of both classification and localization of **all** objects in a given image or a video frame. While classification assigns a class label to the object, localization deals with identifying the location of the object by estimating the coordinates (in pixel coordinates) of a box (square or rectangle) that contains the detected object. These coordinates are estimated by a technique called **Regression**, which uses the same ideology of predictive pipeline as classification, but instead of predicting a class label, it predicts a continuous value. It uses the pixel coordinates of labelled bounding boxes as the ground truth and predicts these coordinates on unseen images. Regression uses a different loss function than a classification algorithm. The most commonly used loss function in a regression algorithm is known as the **Mean Squared Error** (or MSE) loss function. Stephane Lathuiliere et al. (2020, [23]) explain and evaluate regression with modern day neural network. A single CNN can be extended to do both classification and regression simultaneously.

2D Bounding Box Detection: Modern day object detection resulting in a two dimensional bounding box (square or rectangle) marks its beginning with R-CNN (Ross Girshick et al., 2014 [14]). Girshick et al. ([14]) used classical signal processing algorithms to generate **region proposals** that are cropped from the input image. Each region proposal is then treated as a single image and goes

through a CNN that extracts relevant features. These extracted features are then used for classification using a **Support Vector Machine**(SVM) classifier. SVM is a type of classification algorithm that is often used in combination with CNN. Evgeniou and Pontil (2001, [9]) elaborate on both theory and application of the support vector machine algorithm. The CNN in R-CNN is also trained to regress the coordinates of the bounding box. Since the network works with proposed regions, the 'R' in R-CNN stands for 'Region'. R-CNN was compute intensive due to a large number of region proposal per image (approx 2000). Over several years, improvements to RCNN were proposed resulting in Fast-RCNN (Ross Girshick, 2015 [13]) and Faster-RCNN (Shaoqing Ren et al., 2016 [43]). The Faster-RCNN was further extended for the task of **semantic segmentation** and **pose estimation**, the new architecture was called Mask-RCNN (K. He et al., 2017 [19]). Semantic segmentation is the task of assigning a class label to each pixel in the image and pose estimation is estimating the pose of an object by detecting its joints (e.g. joints of human beings in the case of pedestrian detection). Given an object of interest, Mask-RCNN can detect a set of object keypoints in an image. These keypoints are also known as landmark points. Another approach developed around the same time was that of **You Only Look Once** or YOLO (Joseph Redmon et al., 2015 [42]) based detectors. YOLO does not extract region proposals explicitly but divide the image in predefined grids (e.g. 7×7) and then generate bounding boxes for each grid. YOLO performs object detection in a single step as opposed to the two step process (Proposal + Detection) of the R-CNN family. Zou et al.(2019, [54]) surveys the evolution of object detection algorithms stemming from both classical and modern methods and presents a comprehensive survey of last two decades.

The need for 3D detection: Estimation of every real world 3D object as a rectangular box is inherently flawed, and it was argued that a 3D object must be detected in its entirety and not just as a 2D bounding box. One way of doing such

a 3D estimation is through hardware aided approaches of LIDAR (Light Detection and Ranging), that can scan the 3D geometry of the object through sensors. Such a scan results in a 3D representation of a 3D object through many points which are collectively called as a point-cloud. Although LIDAR faithfully represents a 3D object, but its hardware setup is costly and often not helpful for additional analysis other than recognizing the 3D geometry of an object. Therefore, architectures that could estimate 3D shape of an object from its 2D image without the need for additional hardware were needed. **3D estimation from 2D:** A 3D estimation of a 2D object is an ill posed problem. This is because the 2D image is a projection of a 3D object, therefore one dimension of the 3D object is lost. The lost dimension is precisely the distance of each point on 3D object from the camera, this lost distance is often referred to as the **Depth**(Lecture 2 & 3, [5]). Depth estimation on monocular (a single instance of an image) images from RGB cameras has been an active research area in computer vision.

The object of interest in this thesis is the vehicle class (car, vans, trucks etc.) of road users. All these objects are rigid bodies and thus have a **fixed** 3D geometry. This fact helps to model **prior** shape information that can be used to develop constraints to estimate a 3D bounding box. Arsalan Mousavian et al. (2017, [31]) exploits this geometric knowledge by first estimating a 2D bounding box and regressing (using regression) the height, width, length and orientation (yaw) of the possible 3D box. The final location of the 3D bounding box is then estimated by arguing that the projection of 3D bounding box should tightly fit inside the detected 2D bounding box. This entails that at least one vertex of the 3D box should lie on every edge of 2D bounding box (4 in total). Lijie Liu et al. (2019, [25]), Andretti Naiden et al. (2019, [32]), and Fang et al. (2019, [10]) build on the same idea and propose independent improvements for 3D bounding box estimation.

Another geometric approach is based on the assumption that vehicles of same category have similar size and geometry (e.g. cars). Thus, their 3D geometry can be represented by pre-estimated 3D CAD models which can be used as templates. Further, object detection algorithms like MaskRCNN can also localize keypoints(features) of a given object. Florian Chabot et al.(2017, [2]) first predict the keypoints and a probable template class for a detected object, and later in the second stage, a template from the predicted template class is matched using the detected keypoints and the annotated keypoints on the template.

Deformable Templates: Instead of using rigid templates (as Chabot et al., [2]), a template that could be **deformed** or adjusted according to the detected object would result in a more accurate 3D detection. Inspired by the seminal work of **Active Shape Models** (T. F. Cootes et al., 1995 [4]), Zia et.al (2013, [51]) attempted to develop wireframe models for car objects. These models were simplified versions of rich 3D CAD models and had annotated keypoints. A 3D-CAD model with 'n' annotated keypoints can be represented by $3n \times 1$ vector. After a training set of such CAD models is created, Principal Component Analysis or PCA (PCA explained in section 2.2.1) can be applied on the resulting vector space. Once PCA is done, each vector/shape can be represented as a sum of these principal components. Varying the coefficients of each principal component, results in a different shape of the object.

Occlusion Modelling: While many Deep Learning approaches use deformable models and achieve reasonable success in detection, modelling occlusion remained a challenge. Occlusion is omnipresent; vehicles in an image occlude their own parts (because all parts of a 3D object are never visible in a single image), vehicles could also be occluded by both other vehicles and other objects (street lights etc.). Therefore, a definite reasoning about occlusion is indispensable to accurate object detection in crowded scenes.

The presented approach: Most previous approaches either do not model occlusion or model it as a second class objective. The presented approach called Occlusion-Net (Reddy et al., 2019 [40]) explicitly reasons about occlusion statistics of a detected object. It uses wireframe models to reason about the final 3D shape of the detected car object. Such an explicit modelling for occlusion combined with 3D shape estimation results in a better approximation for 2D locations of the detected car’s keypoints (landmarks).

Need for 3D estimation: The aim of this thesis is to enable traffic researchers with 3D-constrained 2D detection of car objects in traffic surveillance videos. Amongst other possible post processing analysis, a 3D detection allows faithful estimation of the **ground plane** of a detected objects. Once such a ground plane is estimated, the centre of the detected ground plane can be used to estimate distance between two cars in pixel coordinates. If mappings between pixel and GPS coordinates for the image frame are present, pixel distances can be used to estimate real world distances between two car objects. Speed of the car object can also be calculated across several frames provided it is tracked. Such understanding of traffic junctions from their surveillance videos can facilitate future research in traffic analysis and its related disciplines.

The upcoming chapters are dedicated to Occlusion-Net’s design, an explanation of its implementation done during this thesis and the results of a qualitative evaluation of its performance on traffic surveillance videos collected by the presiding chair.

3 Design of Implemented Approach

The Occlusion-Net [40] incorporates a multi stage pipeline for training and inference. The objective of explicit occlusion modelling and localization of occluded keypoints is realized by using Graph Neural Networks or GNN (explained in section 2.3.7). Sections 3.3, 3.4 and 3.5 elaborate on the three instances of GNN used in Occlusion-Net. Section 3.1 provides insights in the custom dataset collected and annotated by the authors of Occlusion-Net. Occlusion-Net uses an Off-the-Shelf detector for extracting region of interests, details of which are explained in section 3.2

3.1 Training Dataset

The dataset used by Occlusion-Net is collected by the authors of Occlusion-Net and is named as Carfusion dataset [39]. Carfusion dataset was generated by recording busy traffic intersections using multiple mobile cameras simultaneously. Thus, there are several views of each car object. This publicly available dataset has 53000 images and their corresponding annotations in text files of the same name as image. These annotations are labelled manually. Each image has two associated text files for ground truth out of which one provides bounding box coordinates of the car object and the other contains keypoint annotations and their visibility status. Following is the annotation format for keypoints annotations file.

x-coordinate, y-coordinate, keypoint-ID, object-ID, visibility-status: 1 If visible, else 2.

The original Carfusion dataset has 14 keypoints out of which 12 are selected for training Occlusion-Net. The data set provides the location label of all the 12 keypoints i.e. both visible and occluded. However, the location of occluded keypoints is not used for training. The supervision for location of the occluded keypoints is explained in section 3.4. The Keypoint-ID in object annotation files is assigned based on the following table.

Car Section	Keypoint Name	Keypoint-ID
Wheels	Front Right	1
	Front Left	2
	Back Right	3
	Back Left	4
Headlights	Front Right	5
	Front Left	6
	Back Right	7
	Back Left	8
Car Top	Front Right	10
	Front Left	11
	Back Right	12
	Back Left	13

3D-CAD Models Dataset: Apart from 2D images, Occlusion-Net also co-alesces three dimensional Computer-Aided Design (3D-CAD) models. The CAD models are taken from a public repository called ShapeNet [3]. ShapeNet provides richly annotated and large scale CAD models of many objects out of which the authors chose 472 models of car objects. Each of these models are annotated with 12 keypoints (same as 2D car image) using an open source 3D simulation software called **Blender** [17].

3.2 Keypoint Detection : MaskRCNN

The Occlusion-Net’s [40] central algorithm is applied to the section of the image that contains the car object. The area of an image where the object of interest (here car object) is present can be generated by using any modern day object detection framework such as YOLO [42] or MaskRCNN [19]. These object detectors take an image as input and output pixel coordinates of bounding boxes that contain the object of interest. Supposing a given image with three car objects as input, the output will be pixel coordinates of three bounding boxes corresponding to each of these car objects. Thus, such detectors generate locations of region of interests from an image.

These detectors can be extended to incorporate detection of **keypoints** or features of the object of interest. Thus, for a car object the detector could detect features such as the front and back wheel etc. Occlusion-Net works with a total of twelve such keypoints, details of which are provided in the previous section. For the implementation and also in the published paper, the authors chose MaskRCNN (explained in section 2.5) as the *off-the-shelf* detector. The bounding boxes and keypoints generated by MaskRCNN serves as input to the initial stage of Occlusion-Net pipeline.

Training: MaskRCNN detector is trained only for car object category with visible keypoints. This means that during training, the features which are visible in the car’s image are fed to the training algorithm. The features that are not fed to the algorithm are then implicitly categorised as occluded. This assumption of categorising non-labelled keypoints as occluded keypoints is used to generate ground truth for the next stage of the training pipeline explained in 3.3

At test time however, the detector outputs estimated locations of all 12 keypoints; both visible and invisible. The authors demonstrate that only from such an output it is difficult to differentiate between visible and invisible keypoints

and thereby justify the need of explicit mechanism to differentiate between the two. The following section details such a mechanism which distinguishes between occluded and visible keypoints.

3.3 Occluded Edge Prediction : 2D-KGNN Encoder

The output of the MaskRCNN detector is fed as input to the 2D-KGNN. The 2D-KGNN is the abbreviation for *Two Dimensional - Keypoint Graph Neural Network*. The term 2D is indicative of the 2D locations of the keypoints. The 2D-KGNN predicts which keypoints are occluded.

Training: Since 2D-KGNN Encoder is a Graph Neural Network (explained in 2.3.7), it is evident that it expects a graph as an input. This graph is generated using the output of MaskRCNN. In order to generate the ground truth graph, keypoints are joined together such that a car shape is formed. Thus, the resulting graph has 12 vertices/nodes, each representing a keypoint. The visible and occluded keypoints are separated and added in different lists. The information of keypoints being occluded comes from the previous stage where the non-labelled keypoints were implicitly assumed as occluded.

The occluded keypoints from the previous step are added to list \mathbf{o} and the visible ones to list \mathbf{v} . The edge between two nodes encodes the information of whether any of the two nodes is occluded or not. An edge connecting two nodes V_i and V_j is represented as ε_{ij} . The visibility information of the two nodes connected by an edge is encoded based on following rule:

$$\varepsilon_{ij} = \begin{cases} 1, & \text{if } i \in \mathbf{v} \text{ and } j \in \mathbf{v} \\ 0, & \text{otherwise} \end{cases} \quad (3.1)$$

As per equation 3.1, the edge encodes 1 if both the nodes are visible i.e. they belong to set \mathbf{v} , if either of the two nodes is occluded i.e. belongs to set \mathbf{o} , the edge

encodes a zero. This is a convenient representation to display occlusion dynamics of a given graph. The GNN is employed to learn how to predict occluded keypoints by using the information encoded in the edges of the ground truth graph.

Each node and edge of the ground truth graph contains the following information.

1. Node: Information if the keypoint represented by the node is visible(V^l) or occluded.
2. Node: Human annotated x and y coordinates of visible keypoints in pixel coordinate frame.
3. Node: Type of keypoint. Each keypoint type has an associated numerical value as shown in section 3.1
4. Edge: Value of the edge (0 or 1).

The above graph is used as the ground truth to train the graph that is created by MaskRCNN's output during training. Following is the content of each node in MaskRCNN's output graph.

1. Predictions of x and y coordinates of all 12 keypoints in pixel coordinates(explained in 2.4).
2. The confidence score c that represents how certain MaskRCNN is about a keypoint's prediction, the higher the value, the more confident MaskRCNN is.
3. Type of keypoint.

This graph(created by MaskRCNN) is trained using a GNN (2D-KGNNE) which learns how to classify each edge as 0 or 1 using the ground truth graph. The edge prediction can be seen as a binary classification problem where an edge can either

belong to class '0' or '1'. Thus a cross-entropy loss (2.3.4) is used to train this classifier.

Prediction: The encoder is trained to approximate the term $q(\varepsilon_{ij}|V)$, which represents the conditional probability (explained in 2.2.2) that a given edge (of graph V) ε_{ij} between two nodes V_i and V_j is **occluded** when graph V is observed. Graph V is generated by MaskRCNN's output. The weights learned by the GNN during training are used to estimate ε_{ij} . Such predictions are a result of several **message passing or neighbourhood aggregation operations** details of which can be referred to in the original publication and intuitions can be built from section 2.3.7.

Once the 2D-KGNN learns to differentiate between the visible and occluded keypoints using edge values, Occlusion-Net proceeds towards estimating locations (x,y coordinate) of occluded keypoints.

3.4 Occluded Keypoint Localization : 2D-KGNN Decoder

After identification of occluded keypoints, their **localization** is done using 2D-KGNN Decoder. Localization in this context is the process of approximating the (x,y) pixel coordinates of the occluded keypoint. In order to achieve this, another Graph Neural Network called 2D-KGNN Decoder is employed.

Training: As in the case of 2D-KGNN Encoder, the 2D-KGNN Decoder also trains with an input graph and learns how to adapt this graph to predict information in a given ground truth graph. Once such learning is done, the trained decoder can predict desired information for any input graph. For training the 2D-KGNN Encoder, the occlusion status of the keypoints was the most important feature of the ground truth graph, similarly, for 2D-KGNN Decoder, the most critical feature of the ground truth graph must be the location of the occluded keypoints.

Section 3.2 explains that only locations of visible keypoints are fed to the MaskR-CNN detector. In order to generate the location of occluded keypoints, a technique from the multi-view geometry literature of computer vision is utilized. This technique is called **Point Transfer**. Point transfer allows for inferring the location of a point in a given image by utilizing its location from another set of images. Thus given the location of a point in two or more images, its location in a given image can be calculated. The Tensor (Linear Algebra, section 2.2.1) that maps such a transform is known as the **trifocal tensor**. Out of several approaches for calculating the Trifocal tensor, the authors use the one that utilizes pose of the calibrated mobile camera and camera’s intrinsic parameters (explained in 2.4). Once a trifocal tensor for a given set of views is calculated it can then be used to carry out such estimations. Hartley and Zisserman explain the fundamentals of trifocal tensor in chapter 15 *The trifocal tensor* of their book [18] on multiple view geometry .

The data collection process mentioned in 3.1 used multiple mobile cameras which were simultaneously recording each traffic intersection. Consequently, multiple views of each car object are present. Using this pretext, the authors propose that for any occluded keypoint in a given image, there must exist at least two images where the same keypoint is visible. Since all visible keypoints in the dataset are labelled, the occluded keypoint’s location in an image can be recovered from the images where the occluded keypoint is visible using the trifocal tensor. This is how the ground truth locations for occluded keypoints are generated for the 2D-KGNN Decoder. The ground truth graphs for the decoder contains the following information.

1. All information present in the ground truth for 2D-KGNN Encoder.
2. Node: x and y coordinates of occluded keypoints in pixel coordinate frame, calculated by the trifocal tensor.

The above ground truth graph and the output graph from 2D-KGNN Encoder is used as input to 2D-KGNN Decoder. The encoder then learns to estimate ground truth labels from this input graph. The input to 2D-KGNN Decoder has following information.

1. The graph V created from MaskRCNN's output.
2. Predictions ε of edge values (0 or 1) from 2DKGNN Encoder.

Prediction: Once trained, the 2D-KGNN Decoder can localize the occluded keypoints. The mathematical representation of this estimation is $P_{\theta}(V^g|V, \varepsilon)$. Where V^g are the occluded keypoints whose location the decoder wants to estimate. The term represents the estimation of a set of occluded keypoints V^g , given the output graph V from MaskRCNN, and the output ε of encoder describing which keypoints are occluded. θ is representative of weights that when learned, will maximize the likelihood of the conditional probability $P_{\theta}(V^g|V, \varepsilon)$ (Maximum Likelihood Estimation explained in section 2.2.2). Once all the weights are learned using GNN, message passing operations on an input graph using the learned weights will result in localization of occluded keypoints.

So far the Occlusion-Net pipeline identified (using encoder) and localized (using decoder) the occluded keypoints. However, these predictions are in 2D i.e. the car object and all its keypoints represent a 2D projection of a 3D car object (projective transformation explained in 2.4). The next stage of the algorithm tries to give this 2D representation a 3D lift in order to retrieve true geometry of the 3D car object.

3.5 3D Shape Estimation: 3D-KGNN Encoder

The 3D-KGNN Encoder estimates the shape W and the camera projection matrix π .

In order to extract a 3D geometry from the 2D keypoints, CAD models are used. The idea to represent objects as CAD models is taken from an early work by Zia et.al (2013, [51]). Zia et al. extended an even earlier work which introduced representation of objects using **Active Shape Models** ((T. F. Cootes et al., 1995 [4])). Active Shape Models represent an object's shape as a set of points (or vectors) and performs statistical inferences to capture variance between different points. This inference results in a Point Distribution Model. Principal component analysis or PCA (explained in 2.2.1) is then performed to capture the most relevant dimensions of the shape model. Zia et al. [51] on the same grounds represent shape of a car object by calculating the mean shape μ from all 3D CAD models, and m principal components. Each principal component is represented by its direction p_j and its standard deviation σ_j . This shape can be expressed as follows:

$$Shape = \mu + \sum_{j=1}^m \sigma_j p_j \quad (3.2)$$

Once such a representation is calculated, any 3D model of the car object class can be approximated by varying the magnitude of each dimension by some constant. Considering s_j as the constant/weight of the j^{th} principal component, any shape X can be represented as:

$$X(s) = \mu + \sum_{j=1}^n s_j \sigma_j p_j \quad (3.3)$$

In the above equation the shape X is parameterized by s and hence varies with different values of s . For a give mean shape of car object, varying s could lead to different representations of car objects like sedan, SUV and sports car etc.

Occlusion-Net also models the shape W in a similar fashion and represents it as a linear combination of mean shape b_0 and sum of n principal components, each defined by its direction b_k , standard deviation σ_k and weight β_k as follows:

$$W = b_0 + \sum_{k=1}^n \beta_k \sigma_k b_k \quad (3.4)$$

The 3D-CAD dataset is also annotated with the same 12 keypoints as 2D images.

Training: The 3D-KGNN encoder estimates the shape W and the camera projection matrix π (explained in 2.4). Mathematically, the encoder is defined as $q(\beta, \pi | V)$. V is the graph output by 2D-KGNN Encoder-Decoder network. The formulation can be interpreted as an estimation of β and π given the output graph of 2D-KGNN network. Once β is predicted, it can be substituted in equation 3.4 to calculate the 3D shape. The ground truth for training 3D-KGNN encoder contains:

1. All the information present in the ground truth for 2D-KGNN Encoder-Decoder.
2. A total of 472 3D CAD models of cars, annotated with same 12 keypoints as 2D image (annotations explained in section 3.1)
3. Camera projection matrix π

It is important to note that the above contents of ground truth **do not** contain a 3D shape corresponding to each 2D car in training dataset. In order for the model to learn to estimate the shape parameters, the camera projection matrix is used to facilitate training in a self supervised way. Self supervised learning algorithm themselves create the required ground truth. For estimating 3D shape of a particular car, the camera projection matrix is used to project a probable 3D CAD model on the car’s 2D image. This projection is compared with the ground truth 2D-annotations of car’s keypoints. If it does not match, the coefficient β

is adjusted until the 3D reprojection matches the 2D ground truth annotations. This matching results in creating an appropriate 3D shape for any given car. During this process the 3D-KGNN is also trained to estimate the camera projection matrix.

The input graph for training 3DKGNN Encoder is the output of predictions from 2DKGNN Decoder and contains the following information.

1. Predicted graph containing information about occluded edges ε and locations of occluded keypoints from 2D-KGNN Encoder-Decoder network.

The above input graph is trained using the self supervised training procedure explained before.

Prediction: Once trained, the 3D-KGNN encoder can predict shape W and camera parameter π from the output graph of 2D-KGNN network.

Enforcing 3D symmetry on 2D predictions: The 3D shape estimated is also used to enforce a 3D symmetry on the previously 2D detected keypoints. The estimated camera projection matrix π is used to project the estimated 3D shape 'W' on a given car and the 2D location of keypoints resulting from this projection is the final output of Occlusion-Net. This enforcement of a 3D symmetry corrects any geometrical irregularities in 2D detections (done by 2D-KGNN network) and improves the locations of both visible and occluded keypoints.

Summary Occlusion-net is a 4 stage pipeline whose first stage can be replaced by any *off-the-shelf* object detector. Three instances of graph neural networks (GNN) are trained simultaneously in an *end-to-end* fashion. The first two instances of GNN classify and localize occluded keypoints and the third one enforces constraints guided by 3D geometry of the detected car object. Explicit enforcement of 3D symmetry is essential to introduce geometric reasoning in 2D detections. Implementation details of realizing this design are explained in the upcoming chapter.

4 Implementation

The authors of Occlusion-Net provide a public version of their implementation [38]. Despite the availability of the code, the arduous task of deciphering a sparsely documented source code in order to derive meaning and a sense of its programming paradigm remains. The following sections details such challenges along with some aspects of the implementation followed by the refactoring and functional changes done in code to make it both usable and user-friendly. Section 4.1 enumerates the choice of tools and libraries chosen by original authors. The details of dataset acquisition and training scripts are given in section 4.2. Section 4.3 and 4.4 elaborate on code refactoring and functional changes done for adapting this implementation to meet the thesis goal. All code changes discussed here are committed in a git repository that was forked from original Occlusion-Net implementation [38]. This repository will be made public for future research at the supervising chair.

4.1 Open-Source Tools and Libraries

The public implementation is done in python 3.6. Different modules and libraries are leveraged to realize this implementation, details of which are explained below. Section 4.1.1 discusses code packaging followed by an overview of libraries used for data processing, Neural Network computation and image handling libraries are detailed in sections 4.1.2, 4.1.3 and 4.1.4 respectively.

4.1.1 Code Packaging: NVIDIA-Docker

The code is packaged inside a docker [29] container. Docker containers by default are agnostic to the underlying hardware. This is mostly a benefit of using docker, as the *dockerized* container can run on any underlying hardware. However, there are scenarios where the underlying hardware requires explicit user-level libraries and kernel modules that are not included in native docker containers. One such scenario is when Neural Networks are trained using NVIDIA GPUs. The processing might provided by these GPUs needs dependencies that run at both user and kernel level. In order to make the code running inside docker container access the underlying NVIDIA GPU, **NVIDIA Container Toolkit** [22] was created. NVIDIA container toolkit facilitates building new type of containers called NVIDIA-Docker containers. Code running in such containers can access the underlying GPU. Occlusion-Net’s implementation is packaged in an NVIDIA docker container and thus requires installation of NVIDIA Container Toolkit to access and run the source code.

4.1.2 Data Processing Libraries

Occlusion-Net utilizes python’s **NumPy** library [34] which provides easy-to-use implementations of a plethora of Linear Algebra computations. It enables representation of data using constructs of Vectors and Matrices from Linear Algebra (explained in section 2.2.1). In addition to representations of data, NumPy also provide effective and highly optimized functions for numerical computation on such representations. Optimized functions include user friendly implementations of otherwise complex operations like Matrix-Matrix multiplication, statistical analysis methods such as Principle Component Analysis(PCA), sorting and searching through high dimensional matrices etc.

Another advanced library for numerical and statistical analysis known as **Pandas** [28] is also used in the published code. Pandas extends NumPy and provides

better performance on larger datasets. It represents tabular data i.e. data in shape of rows and columns (like in an excel sheet) using a construct called Dataframes. Pandas specialize in handling such Dataframes and outperform NumPy when data has more than five hundred thousand or more rows.

4.1.3 Libraries for Neural Networks

For the initial *off-the-shelf* detector, a public implementation of MaskRCNN [27] is used. This implementation presents MaskRCNN as a ready to use software framework. It facilitates both training and inference pipelines, both of which are leveraged by Occlusion-Net.

For Graph Neural Network, and all related operations including their initialization and training, a python library named PyTorch [36] is used. PyTorch presents abstracted implementations of critical operations required to train a Neural Network. Such operations include forward-propagation (2.3.5), backward-propagation (2.3.5) and gradient descent(2.2.3) etc. PyTorch also provides tools like TorchVision to facilitate working with image data. Occlusion-Net leverages TorchVision to manipulate image data.

4.1.4 Computer Vision Library

In order to implement image operations that require computer vision algorithms, OpenCV or *cv2* [35] library is used. *cv2* houses more than 2500 algorithms for both machine learning and classical computer vision related tasks. Interfaces to annotate images are also provided and are used by Occlusion-Net to overlay predicted keypoints on processed frames/images.

4.2 Dataset Acquisition and Training

The public code base of Occlusion-Net provides both training data and trained weights of neural networks. Section 4.2.1 and 4.2.2 provide an overview of data acquisition and training respectively.

4.2.1 Acquiring Dataset

The **README.md** file in the public implementation [38] provides link to request access to the Carfusion dataset(explained in section 3.1). Amongst other details, the requester must specify her email address on which she wishes to receive the download link. The email received with download links contains python script that can be used for downloading all the data at once. A script named *carfusion_to_coco.py* in the received email can be used to convert this data to match the format of Microsoft’s COCO dataset [24]. This conversion is important as Occlusion-Net’s training algorithm accepts data in COCO format. The steps required to setup a running environment for Occlusion-Net are well documented in the README.md file of the public implementation.

4.2.2 Training Occlusion-Net

After downloading the dataset and configuring the environment, Occlusion-Net can be trained on the downloaded data using *train.sh* script provided in the repository of the public implementation [38]. On the lab machine of the supervising chair, Occlusion-Net took approximately 72 hours to finish training. The training pipeline stores the learned weights by the name ‘occlusion_net.pth’ in the Occlusion_Net/data folder.

Alternatively, the trained weights can directly be downloaded using the link provided in the README.md file of the public repository. The downloaded weight file is also of the name ‘occlusion_net.pth’ and must be moved to Occlusion_Net/data folder.

Downloading dataset and training is only recommended in scenarios when new training data is available or when the hyper-parameters of the training algorithm are changed. In such cases, it is imperative to re-train the associated neural networks to factor in new changes.

4.3 Refactoring

In order to explore the functional characteristics of Occlusion-Net, it was required to refactor certain aspects of it for making it suitable for video analysis use case. As an initial approach, Occlusion-Net was seen as a **blackbox** accepting images and returning their processed versions. However a dive inside this blackbox was required in order to adapt it to context. Sections 4.3.1 and 4.3.2 detail the first encounter whose limitation led to adaptations explained in sections 4.3.3 and 4.3.4.

4.3.1 Error in Docker File

After first run, the docker file failed with an error message that after debugging indicated towards a missing installation for Miniconda software. Miniconda is a distribution of the **Conda** package management software. This was resolved by replacing (in Dockerfile) 'curl' command with 'wget' to download latest Miniconda repository. Further, Miniconda was installed using the installation script named 'Miniconda3-latest-Linux-x86_64.sh' which was present in the downloaded repository.

After this step, the docker engine was able to install all the dependencies required by Occlusion-Net.

4.3.2 Image Directories as Input

The public Occlusion-Net implementation accepts one single image at a time for testing. This testing is initiated by calling *test.sh* shell script which in turn calls *infer.py* python script. The image to be processed is provided as an argument to

parameter named '*ul*' which is passed to *test.sh* at runtime. The parameter '*ul*' only accepts a single image for each run. Thus, in order to process 'n' images, the docker container running Occlusion-Net must be restarted 'n' times.

In order to observe the detection quality on multiple images, Occlusion-Net was modified to process an entire image directory in one go. This was realised by replacing the *test.sh* argument '*ul*' to '*ir*'. The '*ir*' parameter accepts a path to the image directory which is then parsed as an argument in *infer.py* file. This slight modification enables the user to run Occlusion-Net consecutively on multiple images in a single call to *test.sh*.

4.3.3 Frame Extraction in Parallel.

Now that the Occlusion-Net can work with multiple images it was required to extract the images/frames from different traffic surveillance videos. An open source library known as *cv2* [35] was used for this task. Initial attempts of extracting frames from videos worked well for videos which were of length 10-15 minutes. However, as the video length increased to 60 minutes or more, the process of extracting frame became a bottleneck in overall processing.

In order to reduce the frame extraction time, the paradigm of multiprocessing was utilized. The code for same is provided inside the 'usage' folder in the forked Occlusion-Net repository in the file named *videoToframes.py*. This python file accepts the video file's path as an argument to a parameter named '-path', and outputs the extracted frames to the 'Occlusion_Net/demo' folder. *videoToFrames.py* can extract frames from a high quality 60 minute video under 90 seconds as opposed to 30 minutes with a non-parallel program, thereby significantly improving the overall processing performance.

Each frame after being processed by Occlusion-Net is stored in 'Occlusion_Net/logs' folder. These processed frames are then converted back to a video using

another helper script called ‘FramesToVideos.py’ that also performs this task in a parallel fashion.

For some videos, the length of the resulting video of their processed frames was significantly **shorter** than their original video length. Probing further, exposed the problem of **missing frames**. Missing frames is the name given to all frames that Occlusion-Net could not process. Consequently, such frames are not present in the output folder ‘Occlusion_Net/log’.

Occlusion-Net was modified to store such missing frames in a separate directory. It was expected to see a correlation between the number of missing frames and the reduced video length, i.e. the larger the difference in the original and processed video, the more frames Occlusion-Net skipped for that video. However, such a correlation was not observed. Instead, an underlying susceptibility of *cv2* library was discovered as the cause of this problem.

FramesToVideos.py combines the processed frames into videos using the *videoWriter* functionality of *cv2* library. The *videowriter* stops reading new frames if there is a missing frame sequence (number) in the frame directory. For e.g. if frame number ‘10’ was **missing**, the *videowriter* stops reading frames after it processes frame ‘9’. This stopping results in a video that only has ‘9’ frames. This discovery explains the observation that there is no correlation between the shortened video lengths and number of frames skipped by Occlusion-Net. The length of the video depends on the frame-number of the first missing frame rather than the total number of missing frames. Thus, if a video has ‘100’ frames and Occlusion-Net successfully processes ‘99’ of them but skips frame-number ‘10’, the resulting video created using *FramesToVideos.py* will have ‘9’ frames in total.

Multiple workarounds were tried to resolve this issue one of which included copying the corresponding missing frame from the unprocessed frame directory thereby not breaking the frame sequence in processed frame directory. However, a robust program was not possible due to underlying limitation of *cv2*.

4.3.4 Video as Input

The problem of missing frames and their erratic reconstruction led to addition of a new feature which enables Occlusion-Net to accept video files as input, and output processed video files. *test.sh* was extended with a new parameter named '*-vid*' which accepts a video's file path as a parameter. The argument '*-vid*' when handled in 'infer.py' file sets a flag named **Video** to 1. This flag is checked inside an 'if' loop and when true, invokes the video processing pipeline that was added to 'infer.py'.

The video processing pipeline accepts the video as input and processes each frame with Occlusion-Net. The processed frame is written to the output video file before next frame is processed. The video output is written using *cv2*'s *videowriter* functionality.

Further, in order to facilitate analysis of missing frames for future work, all missing frames were stored in a separate 'missing_frames' directory inside Occlusion_net/logs/ path.

4.4 Functional changes

After adapting Occlusion-Net to suite the video analysis use case, its output was analyzed. As a part of this exploration few irregularities at functional level were observed. These irregularities consisted of both missing and mismatched outputs and are explained in section 4.4.1 and section 4.4.2 respectively along with the functional changes done to rectify them. The last two sections mention changes done to aid visual analysis.

4.4.1 Keypoints In Output

Once the videos are processed the processed data is stored in a *.json* file inside Occlusion_Net/log path. It was found that this *.json* file only contained the

bounding box coordinates of the detected cars and their corresponding object Id's. Thus, by default, the Occlusion-Net **does not** store the location of predicted keypoints. Since predicted keypoints are a must for any post processing analysis, they must be a part of the *.json* file.

In order to extract keypoints, the prediction pipeline was deconstructed. Keypoints and their confidence scores were then extracted from the prediction object. The code changes and extensions can be tracked in the corresponding git commit.

4.4.2 Mismatched BB and Keypoints

A close inspection of the *.json* output revealed that there was a mismatch between the bounding boxes (BB) and their corresponding Keypoints. Both BB and keypoint have an associated id. This id represents a particular car object in a given frame. For e.g. it was observed that BB with id '1' did not encapsulate Keypoints corresponding to id '1' but corresponding to id '2'. Such a mismatch can significantly mislead the post processing analysis and hence must be resolved.

The cause of this discrepancy arose from erratic handling of prediction object and was resolved by considering only top predictions for both BB and Keypoints, this implicitly matched their id's and resolved the problem.

4.4.3 Incorrect Class Labels

The detected car's class label was wrongly set to 'Person'. Such a mismatch can hamper future data analysis in situations where both 'Car' and 'Person' are possible outputs of the detector. This was resolved by adjusting the CATEGORY list inside *predictor.py*.

4.4.4 Bounding Box Overlay

As a part of qualitative evaluation of Occlusion-Net's performance and to verify certain claims by the authors, it was required to observe the position of bounding

boxes with respect to predicted keypoints. Details of these claims are explained in the section 5.2. Thus, to overlay or draw bounding boxes, adjustments were made inside *infer.py* after which each output frame had both Bounding boxes and Keypoints drawn on it.

Summary: The implementation process of Occlusion-Net presented several other challenges (other than the ones mentioned above) which are often encountered while approaching an unknown code base. After the refactorings and functional changes explained above, the public implementation of Occlusion-Net runs seamlessly and produces evaluation ready data. Next chapter encompasses a qualitative evaluation of this data.

5 Evaluation

This chapter presents a qualitative analysis of Occlusion-Net’s performance on video data collected by the supervising chair of Traffic Engineering. Section 5.1 compares Occlusion-Net’s performance on the this video data with the published results. Evaluation on heavy and moderate vehicles is presented in section 5.1.2. Section 5.2 verifies the failure analysis published by the authors [41]. The frames skipped by Occlusion-Net during processing are analyzed in section 5.3. The chapter ends with section 5.4 which provides a performance estimate of Occlusion-Net’s detection performance in frames-per-second (fps). All evaluations are performed on the lab machine (Ubuntu-18.04.5 LTS) at supervising chair which has NVIDIA’s ‘GeForce RTX 2070’ graphics card that houses a ‘TU106’ Graphics Processing Unit.

5.1 Qualitative Evaluation

The authors published both qualitative and quantitative results. However, empirical evaluation of Occlusion-Net requires analysis across multiple baselines and hence is out of scope for this thesis. The qualitative results published by the authors are thus chosen to evaluate both the correctness of implementation and the authenticity of published results.

5.1.1 Evaluation Across Occlusion Categories

A larger part of published qualitative evaluations display Occlusion-Net’s predictions by varying the nature of occlusion that a car object experiences. This leads

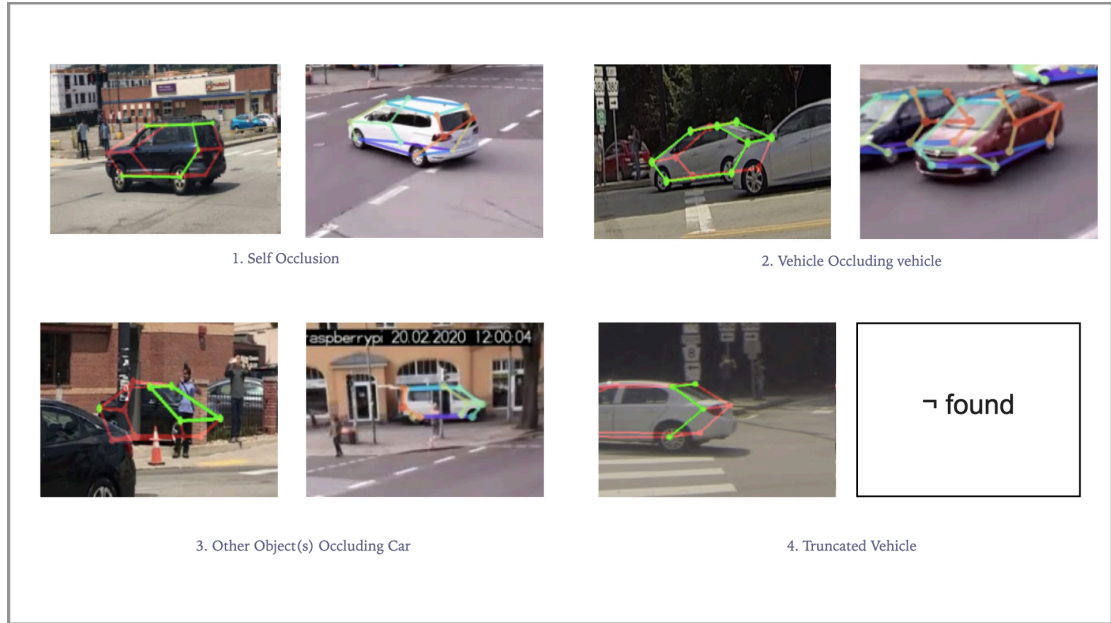


Figure 5.1: Qualitative evaluation across varying occlusion categories (1-4). Categories 1-3 are successfully reproduced. Category 4 remains irreproducible.

to four different occlusion categories viz; self occlusion, vehicle occluding vehicle, other object occluding vehicle and truncated vehicle. Each of these categories and a comparison between their published and reproduced results (using the public implementation) are explained below.

Self Occlusion: This category considers vehicle/car objects that are not occluded by any other object or vehicle but self-occlude some of their own parts/key-points. Such occlusion are typical and always exist in an image of any rigid body. One can never observe all parts of a 3D object in a single 2D image. The published results of this category were reproduced as it is on custom videos collected by the supervising chair and are shown in figure 5.1(1).

Vehicle Occluding Vehicle: This category includes vehicles that are occluded by other vehicles. The published results only consider occlusion of car objects with other car objects and not with other heavy vehicles. In the context of car objects occluding other car objects, the results were reproduced as shown in 5.1(2).

Other Object(s) Occluding Vehicle: Scenarios in which a car object is occluded by any non-vehicle object such as traffic signs, trees and humans are considered in this category. The 'other' objects include all objects that are not explicitly modelled (like car object) in Occlusion-Net. The results of this category are reproducible and shown in 5.1(3).

Truncated Vehicle: In traffic surveillance videos, it is expected to observe instances of vehicle objects which are only partially present in a frame. For e.g. when an object is entering the camera's view and leaving it, it is evident that such a vehicle object would appear **truncated** in several frames. This category considers such truncated car objects. The results of this category were not reproducible at all. Despite analysis through several videos, the truncated category remained irreproducible. Figure 5.1(4-left) shows the published results whose matching results were not reproduced.

Results of all four occlusion categories are shown in figure 5.1. Each category contains the image published by authors of Occlusion-Net(left image) and its reproduced image (right image).

5.1.2 Performance on Moderate and Heavy Vehicles:

Heavy vehicles such as trucks and moderate vehicles such as vans are also accurately detected in most of the frames. However, the performance on these two classes appears to be relatively less accurate than the car object class.

Inaccurate detections of heavy vehicles particularly suffer from the problem of **multiple hypothesis**. Multiple hypothesis is the name given to detections where a single vehicle is detected with more than one bounding box; each encompassing the same vehicle differently. Since Occlusion-Net is trained to estimate 12 keypoints in each bounding box it receives as input from MaskRCNN (explained in section 3.2), it ends up wrongly estimating 12 keypoints within each of these

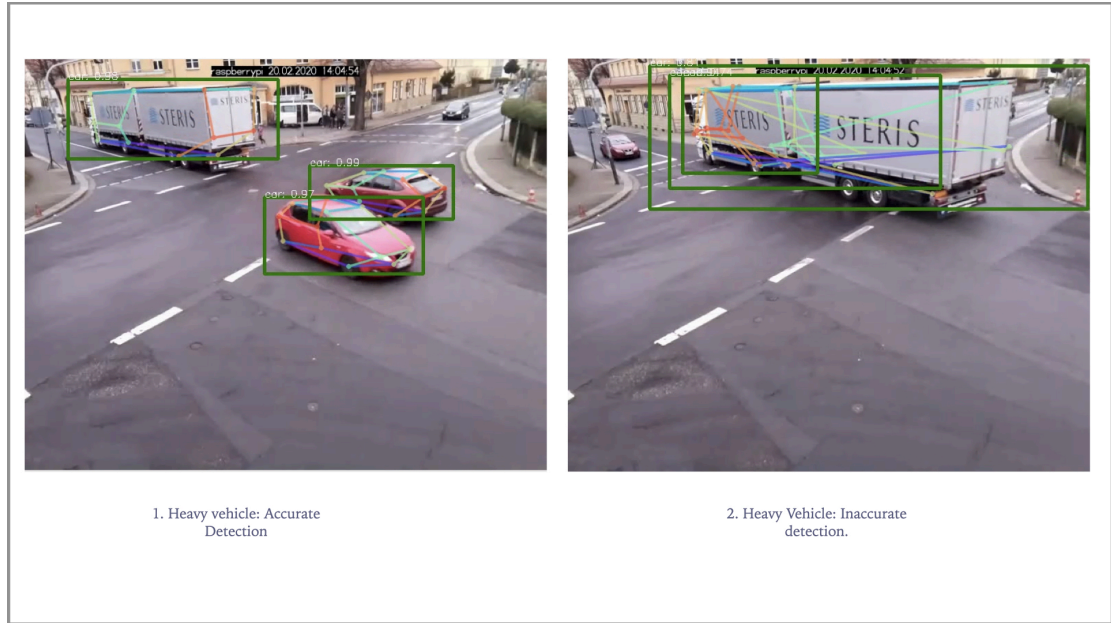


Figure 5.2: Performance on Heavy Vehicle: Image 1 shows accurately detected heavy vehicle. Image 2 demonstrates the problem of multiple hypothesis leading to inaccurate localization of heavy vehicle.

bounding boxes. Figure 5.2 shows the problem of multiple hypothesis on a heavy vehicle.

Another detection discrepancy observed in both heavy and moderate vehicles is that of a **partial detection**. Here the vehicle object is wrongly estimated with a bounding box that covers the vehicle partially. Occlusion-Net is then forced to localize all keypoints within this bounding box. The same can be observed for heavy and moderate vehicle in figure 5.3 and 5.4 respectively.

5.2 Verification of Failure Analysis

The authors of Occlusion-Net also published supplementary material [41], in which they provide a qualitative approximation of factors that lead to failed object detection by Occlusion-Net. The authors claim that the bounding box from the initial detector (MaskRCNN) plays a crucial role in the overall performance of the

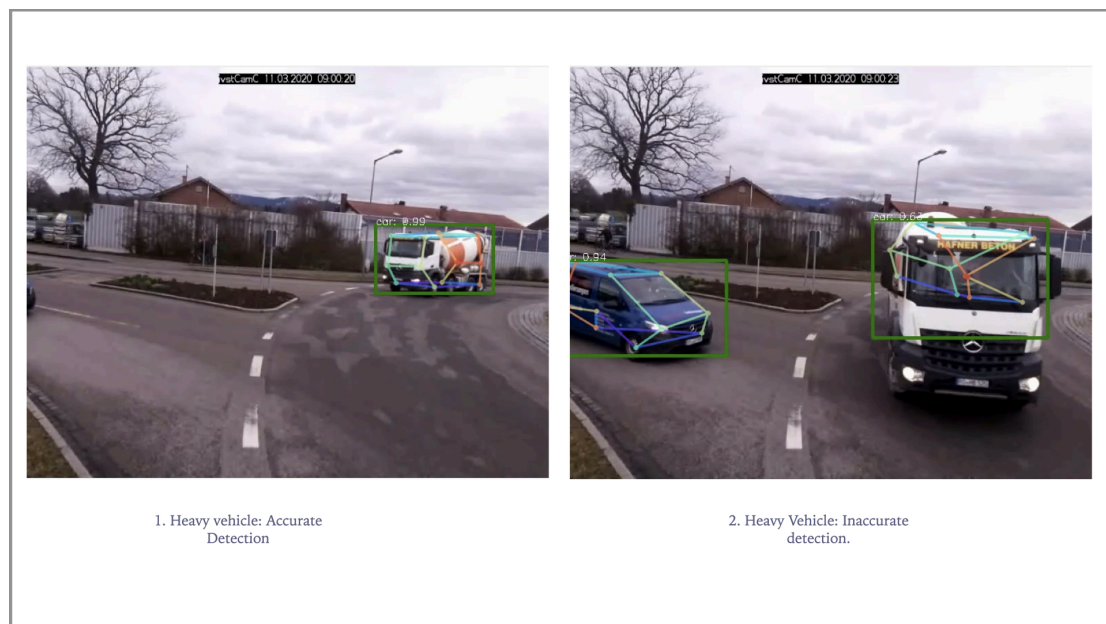


Figure 5.3: Performance on Heavy Vehicle: Image 1 shows accurately detected heavy vehicle. Image 2 shows inaccurate detection due to partially covering bounding box.

detection pipeline. This entails that each failed or erroneous detection must have an incorrectly approximated bounding box at its source.

In order to verify this claim, the Occlusion-Net was extended to overlay bounding boxes on output frames (described in section 4.4.4) and verify if erroneous detections correspond to wrongly detected bounding boxes. The failure analysis of moderate and heavy vehicles explained in section 5.1.2 was also possible only after overlaying the bounding boxes.

5.2.1 Truncated Cars

In detections across different videos, keypoints corresponding to truncated cars are always incorrectly localized. In figure 5.5(a), the detected bounding box for truncated car encapsulates only the visible end of the car, thus forcing Occlusion-Net to localize all the keypoints within this bounding box. Since, this bounding box does not contain all the keypoints, any attempt at localizing them inside the



Figure 5.4: Performance on Moderate Vehicle: Image 1 shows accurately detected moderate vehicle. Image 2 shows inaccurate detection due to partially covering bounding box.

box is bound to fail. Thus, Occlusion-Net incorrectly localizes all 12 keypoints on the visible parts of car object.

5.2.2 Overlapping Cars

Another failure class comprises of cars whose bounding box overlap due to their position w.r.t to each other and to the camera. As a result, one bounding box can for e.g. encompass two back-right wheels of two different cars. Occlusion-Net appears to be susceptible to such duplication of keypoints. Such a duplication leads to a distorted estimate of car's shape as shown in figure 5.5(b).

Figure 5.5 is generated from evaluation done during this thesis and provides visual confirmation that every failed keypoint localization has an incorrectly detected bounding box at source.

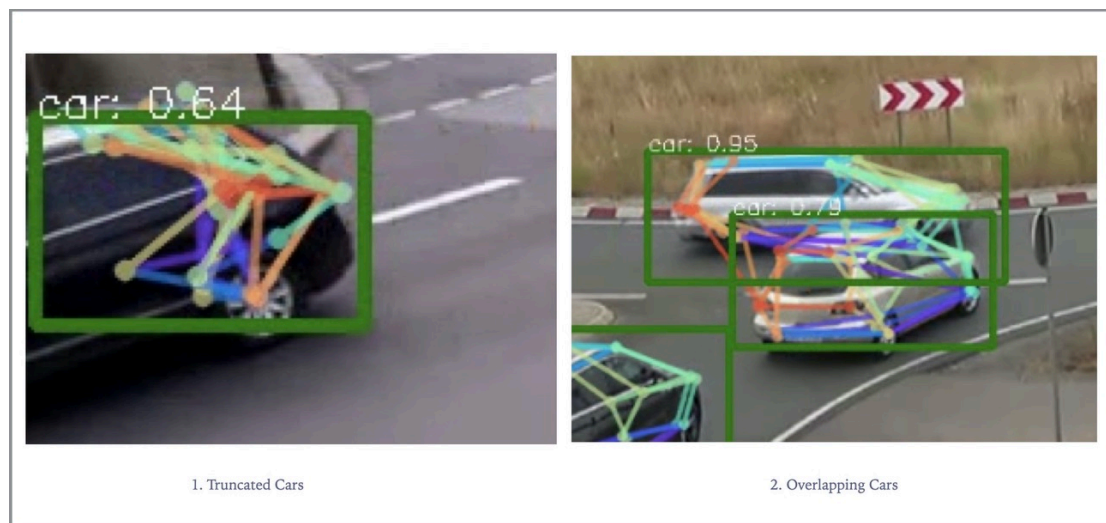


Figure 5.5: Failure Analysis Verification: Image 1 shows bounding box encapsulating only half of the car causing erroneous localization of keypoints. Image 2 shows overlapping cars causing duplication of several keypoints in each bounding box leading to distorted shape estimation.

5.3 Analysis of Missing Frames

In majority of videos processed with Occlusion-Net, it was observed that Occlusion-net skips (does not process) some frames of the input video. In order to reason about this occasional skipping of frames, these missing frames (introduced in section 4.3.3) were stored in a separate directory (explained in section 4.3.4) and were fed to Occlusion-Net using the image directory processing switch '*ir*' (mentioned in section 4.3.2). Unsurprisingly, these frames were skipped again. A closer inspection of these frames revealed that they either do not contain any vehicle object, or the object(s) is in a region where MaskRCNN does not detect a bounding box. Since in either case no bounding box is detected, Occlusion-Net pipeline cannot process the frame.

The skipped frame analysis is shown in figure 5.6 and 5.7. The image (1) of both figures shows a processed frame and highlights the *no-detection-area* in red. Image (2) and (3) in both figures shows skipped frames where no objects are present and objects are present **only** in no detection area respectively.



Figure 5.6: Skipped Frames: 1 shows a processed frame. No detection area is highlighted by area under the red box. 2 and 3 show two categories of missed frames.

The missing frame analysis proves that in order for a frame to be processed by Occlusion-Net, the frame must have at least one bounding box detected by the initial detector (MaskRCNN).

5.4 Test Performance

The original publication claims to record a speed of 30 fps(frames per second) at test time. However, the observed performance on the lab machine is around 9.5 to 10 fps. This difference can be attributed to a difference in the testing hardware which the authors did not made public.

In order to probe the affect of skipped frames on the overall processing performance, the processing time of several videos was recorded as shown in table 5.1. The skipped frames from each of these videos were stored in a separate directory. Processing speed was calculated in fps by dividing the sum of processed frames and skipped frames by total processing time.

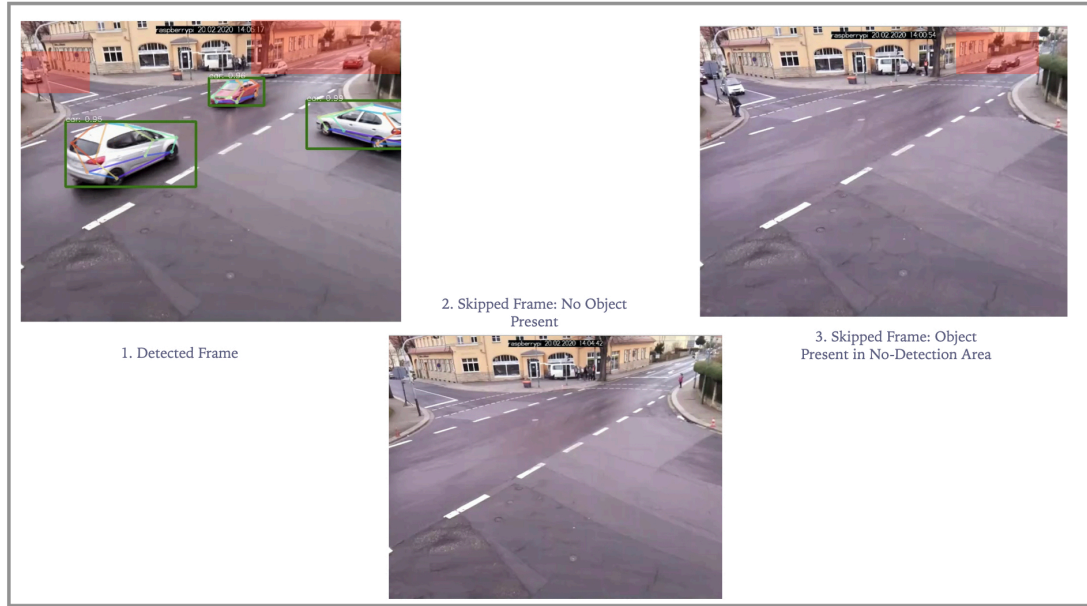


Figure 5.7: Skipped Frames: 1 shows a processed frame. No detection area is highlighted by area under the red box. 2 and 3 show two categories of missed frames.

In table 5.1, the video named 'raspberrypi_14-00-00.mp4' skipped 6395 frames from a total of 18005 frames and resulted in overall processing performance of 9.9 fps. Whereas, the video named "raspberrypi_FR20_11-45-00.mp4" skipped zero frames from a total of 18004 and had a performance of 9.6 fps. Despite the former video missing 6395 frames, its processing took similar time as the latter.

Missing frames do not have a bounding box and therefore are not processed by Occlusion-Net. This entails that most of the processing cycle is spent in MaskR-CNN's detections and Occlusion-Net's detection pipeline is capable of real time processing given that it receives the region of interest (image with bounding boxes) input with a matching rate.

The performance results also conclude that the overall performance is governed by the number of frames fed to the detection pipeline and both processed and skipped frames consume processing cycles in equal proportions. Thus, even if Occlusion-Net skips all the frames in a video, its performance would be equivalent to the case where it successfully processed a similar number of frames.

Video Name	Processing Time(s)	No. of Processed Frames	No. of Skipped Frames	Processing Speed(fps)
raspberrypi_11-16-51.mp4	1595	15757	10	9.9
raspberrypi_11-30-00.mp4	1839	17994	12	9.8
raspberrypi_14-00-00.mp4	1815	11610	6395	9.9
ivstCamC_FR20_09-00-00.mp4	4806	23817	25265	10.2
raspberrypi_FR20_11-45-00.mp4	1880	18004	0	9.6
raspberrypi_FR20_12-00-00.mp4	1881	18006	20	9.7
raspberrypi_12-15-00.mp4	1897	18014	1	9.5

Table 5.1: Performance measure of Occlusion-Net in frames-per-second(fps). Final processing speed is calculated by dividing total number of frames by processing time.

Summary: The presented qualitative evaluation reproduces all the published results except the one with truncated car category. While Occlusion-Net is capable of localizing heavy vehicles and moderate vehicles, its performance is best observed across the car object category. The reproducible results of failure analysis (published by the authors) followed by missing frame analysis (not published) confirms the dependence of Occlusion-Net on initial bounding box detector. Processing speed of Occlusion-Net appears to be limited by the detection performance of initial detector as well.

6 Conclusion

Deep Learning is a sub field of Machine Learning that gained traction in recent years due to the advent of high processing GPUs and availability of sufficiently large amount of training data. The entire discipline is now advertised as a related field of computer science. A larger part of this new found connection is because of the ease by which many complex machine and deep learning algorithms can be implemented with highly optimized and programmer friendly libraries. These libraries abstract complex estimation algorithms which are mathematical formulations of ideas from Statistics, Probability, Linear Algebra and Infinitesimal Calculus. Despite such all-encapsulating libraries, most of the algorithms require readjustments to adapt to the applied context. Such alterations require deep insights into the structure of these algorithms and thus demand familiarity with their constituting mathematical concepts. Knowledge of such concepts is as important as the knowledge of the domain in which various machine learning algorithms are applied. Deep learning provides robust solutions to many classical problems from the domain of computer vision, most important of which is that of object detection. It is therefore imperative to understand the required concepts from computer vision Literature. Chapter 2 attempts at building foundations for all the concepts discussed in the later chapters.

The implemented approach called Occlusion-Net [40] utilizes the recent construct of Graph Neural Network to model the object detection problem and justifies the need for explicitly modelling occlusions rather than treating them as a sub-task of object detection. This separation distinguishes Occlusion-Net from

other approaches. Further, the novel approach of providing supervision for occluded keypoints using images where they are visible, significantly eliminates the dependency on human annotators whose accuracy in labelling occluded keypoints is often subjective. The lifting of 2D keypoints to fit a 3D shape is a combination of established model matching approaches with the recent approach of training a self supervised reprojection loss. This makes Occlusion-Net a combination of supervised and self supervised learning. The design details can be found in chapter 3.

The public implementation provided by the authors [38] is indeed helpful but is not readily adaptable to video analysis context. The absence of a tracking algorithm that can uniquely identify every object through several frames prevents from estimating object trajectories which are essential for traffic flow analysis. The provided implementation needs both functional changes and refactoring to fit the current use case. The functional changes done, do not affect the hyper-parameters of the trained neural networks but make the output more informative, accurate and analyzable. The entire code is packaged inside Nvidia-Docker and thus is only available to run on Linux distributions; making this implementation unavailable on Windows machines. All additions to the publicly available code base and its packaging details can be found in chapter 4.

Qualitative evaluation of Occlusion-Net confirmed all the published results except the one with truncated cars. Although the implementation did not correctly localize the keypoints of truncated cars, it performed well across several other occlusion categories and is capable of localizing heavy and medium sized vehicles as well. However, the car category is the one with most successful predictions. All of the observed erratic predictions arose due to faulty detections of the bounding box which is fed as an input (region of interest) to the detection pipeline. This dependency exposes Occlusion-Net’s susceptibility to the initial detector and calls for a more robust design for the initial detector.

Occlusion-Net’s unique graphical approach, ingenious supervision strategy, and explicit reasoning for occlusions provides new representation of the classical problem of object detection and thus sets new grounds for future research.

7 Future Work

In most settings, Occlusion-Net proves to be a robust detection algorithm. However, during this thesis it was discovered that it is as robust as the initial detector which generates the region of interest. In that light, it is imperative to consider alternative object detection algorithms. YOLO (You Look Only Once) version five [47] is the latest addition to the famous YOLO family of detection algorithms. The earlier versions of YOLO traded off accuracy for speed and thus were inferior to MaskRCNN in accuracy. However, the YOLO-V5 version claims to reduce this gap and therefore can be used as an initial detector for Occlusion-Net.

Initial detections can also be improved by introducing more training data from German traffic scenes. The new data would require multiple views of the same object in order to calculate the trifocal tensor which is essential in providing supervision for occluded keypoints. A possible workaround to bypass the need of multiple views and trifocal tensor is to provide human labelled annotations of both visible and occluded keypoints. This would also require code modifications to incorporate the labelled occluded keypoints in the training pipeline.

Other than Occlusion-Net, detection algorithms that do not need bounding boxes as inputs can also be explored. Pishchulin et al. [37] propose an approach for human pose estimation. The approach is bottom-up in nature, i.e it detects all the human body parts (similar to keypoints of car object) present in an image as a fully connected graph. Once such a graph is created, individual humans are detected by cutting this graph into individual groups of keypoints corresponding to each human object. Such an approach could be extended to keypoint based car

object detection. Since cars are rigid bodies, their expected geometry is known and can be leveraged in the final graph cutting stage.

Occlusion-Net is a detection algorithm. In majority of use cases researchers want to uniquely track the movements of an object through multiple frames. Tracking algorithms can take an object's detection in current frame as input, and can uniquely track the same object in the next frame by either matching its detection in the consecutive frame with the input detection, or by directly estimating the next possible position using the input detection. Such approaches are called *tracking by detection* and can be employed for meaningful analysis of detected and tracked objects.

The supervising chair is developing an in-house detection pipeline called Open-Traffic-Cam. Open-Traffic-Cam aims to facilitate research in questions related to traffic analysis and will be designed to work on windows operating system. If Occlusion-Net is extended with aforementioned features, it can be a part of Open-Traffic-Cam's detection and tracking step. Since Occlusion-Net is packaged inside Nvidia-Docker, its inclusion in Open-Traffic-Cam is only possible if its implementation can be adapted to run on windows machines. Amongst many other refactorings, this adaptation would primarily require to move the code outside NVIDIA-Docker container.

In nutshell, successful inclusion of Occlusion-Net's robust algorithm in any empirical result oriented research project would require contextual training data, a robust initial detection and ability to track objects.

Bibliography

- [1] T.M. Apostol. *Calculus, Volume 1*. Blaisdell book in pure and applied mathematics. Wiley, 1967. ISBN: 9780471000051. URL: <https://books.google.de/books?id=RQtRAAAAMAAJ>.
- [2] Florian Chabot et al. *Deep MANTA: A Coarse-to-fine Many-Task Network for joint 2D and 3D vehicle analysis from monocular image*. 2017. arXiv: 1703.07570 [cs.CV].
- [3] Angel X. Chang et al. *ShapeNet: An Information-Rich 3D Model Repository*. Tech. rep. arXiv:1512.03012 [cs.GR]. Stanford University — Princeton University — Toyota Technological Institute at Chicago, 2015.
- [4] T. F. Cootes et al. “Active shape models - their training and application”. English. In: *Computer Vision and Image Understanding* 61.1 (Jan. 1995), pp. 38–59. ISSN: 1077-3142. DOI: 10.1006/cviu.1995.1004.
- [5] Daniel Cremers. *Variational Methods for Computer Vision*. <https://vision.in.tum.de/teaching/ws2013/vmcv2013>. 2014.
- [6] G. Cybenko. “Approximation by superpositions of a sigmoidal function”. In: *Mathematics of Control, Signals, and Systems (MCSS)* (Dec. 1989). URL: <http://dx.doi.org/10.1007/BF02551274>.
- [7] Marc Peter Deisenroth, A. Aldo Faisal, and Cheng Soon Ong. *Mathematics for Machine Learning*. Cambridge University Press, 2020.
- [8] Allen B. Downey. *Think Stats*. O’Reilly Media, Inc., 2011. ISBN: 1449307116.
- [9] Theodoros Evgeniou and Massimiliano Pontil. “Support Vector Machines: Theory and Applications”. In: *Machine Learning and Its Applications: Advanced Lectures*. Ed. by Georgios Paliouras, Vangelis Karkaletsis, and Constantine D. Spyropoulos. Berlin, Heidelberg: Springer Berlin Heidelberg, 2001, pp. 249–257. ISBN: 978-3-540-44673-6. DOI: 10.1007/3-540-44673-7_12. URL: https://doi.org/10.1007/3-540-44673-7_12.

- [10] Jiaojiao Fang, Lingtao Zhou, and Guizhong Liu. *3D Bounding Box Estimation for Autonomous Vehicles by Cascaded Geometric Constraints and Depurated 2D Detections Using 3D Results*. 2019. arXiv: 1909.01867 [cs.CV].
- [11] David A. Forsyth and Jean Ponce. *Computer Vision: A Modern Approach*. Prentice Hall Professional Technical Reference, 2002. ISBN: 0130851981.
- [12] Floris Geerts, Filip Mazowiecki, and Guillermo A. Pérez. *Let’s Agree to Degree: Comparing Graph Convolutional Networks in the Message-Passing Framework*. 2020. arXiv: 2004.02593 [cs.LG].
- [13] Ross Girshick. *Fast R-CNN*. 2015. arXiv: 1504.08083 [cs.CV].
- [14] Ross Girshick et al. *Rich feature hierarchies for accurate object detection and semantic segmentation*. 2014. arXiv: 1311.2524 [cs.CV].
- [15] Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. <http://www.deeplearningbook.org>. Cambridge, MA, USA: MIT Press, 2016.
- [16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. <http://www.deeplearningbook.org>. MIT Press, 2016.
- [17] GPL. *Blender*. Version 3.0. URL: <https://www.blender.org/>.
- [18] Richard Hartley and Andrew Zisserman. *Multiple View Geometry in Computer Vision*. USA: Cambridge University Press, 2000. ISBN: 0521623049.
- [19] K. He et al. “Mask R-CNN”. In: *2017 IEEE International Conference on Computer Vision (ICCV)*. 2017, pp. 2980–2988. DOI: 10.1109/ICCV.2017.322.
- [20] Leskovec Jure. *CS224W: Machine Learning with Graphs*. <http://web.stanford.edu/class/cs224w/>. 2019.
- [21] Leslie Kaelbling. *MIT 6.036: Introduction to Machine Learning*. <https://openlearninglibrary.mit.edu/courses/course-v1:MITx+6.036+1T2019/course/>. 2020.
- [22] klueska. *nvidia-docker*. <https://github.com/NVIDIA/nvidia-docker>. 2017.
- [23] Stephane Lathuiliere et al. “A Comprehensive Analysis of Deep Regression”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 42.9 (Sept. 2020), pp. 2065–2081. ISSN: 1939-3539. DOI: 10.1109/tpami.2019.2910523. URL: <http://dx.doi.org/10.1109/TPAMI.2019.2910523>.
- [24] Tsung-Yi Lin et al. *Microsoft COCO: Common Objects in Context*. 2014. URL: <http://arxiv.org/abs/1405.0312>.

- [25] Lijie Liu et al. *Deep Fitting Degree Scoring Network for Monocular 3D Object Detection*. 2019. arXiv: 1904.12681 [cs.CV].
- [26] Niall O’ Mahony et al. “Deep Learning vs. Traditional Computer Vision”. In: *CoRR* abs/1910.13796 (2019). arXiv: 1910.13796. URL: <http://arxiv.org/abs/1910.13796>.
- [27] Francisco Massa and Ross Girshick. *maskrcnn-benchmark: Fast, modular reference implementation of Instance Segmentation and Object Detection algorithms in PyTorch*. <https://github.com/facebookresearch/maskrcnn-benchmark>. Accessed: [Insert date here]. 2018.
- [28] Wes McKinney et al. “Data structures for statistical computing in python”. In: *Proceedings of the 9th Python in Science Conference*. Vol. 445. Austin, TX. 2010, pp. 51–56.
- [29] Dirk Merkel. “Docker: lightweight linux containers for consistent development and deployment”. In: *Linux journal* 2014.239 (2014), p. 2.
- [30] Marvin Minsky and Seymour Papert. *Perceptrons: An Introduction to Computational Geometry*. Cambridge, MA, USA: MIT Press, 1969.
- [31] Arsalan Mousavian et al. *3D Bounding Box Estimation Using Deep Learning and Geometry*. 2017. arXiv: 1612.00496 [cs.CV].
- [32] Andretti Naiden et al. *Shift R-CNN: Deep Monocular 3D Object Detection with Closed-Form Geometric Constraints*. 2019. arXiv: 1905.09970 [cs.CV].
- [33] Chigozie Nwankpa et al. *Activation Functions: Comparison of trends in Practice and Research for Deep Learning*. 2018. arXiv: 1811.03378 [cs.LG].
- [34] Travis E Oliphant. *A guide to NumPy*. Vol. 1. Trelgol Publishing USA, 2006.
- [35] OpenCV. *Open Source Computer Vision Library*. 2015.
- [36] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035. URL: <http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf>.
- [37] Leonid Pishchulin et al. *DeepCut: Joint Subset Partition and Labeling for Multi Person Pose Estimation*. 2016. arXiv: 1511.06645 [cs.CV].
- [38] Dinesh Reddy. *OcclusionNet*. <https://github.com/dineshreddy91/OcclusionNet>. 2019.

- [39] N. D. Reddy, M. Vo, and S. G. Narasimhan. “CarFusion: Combining Point Tracking and Part Detection for Dynamic 3D Reconstruction of Vehicles”. In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2018, pp. 1906–1915. DOI: 10.1109/CVPR.2018.00204.
- [40] N. Dinesh Reddy, Minh Vo, and Srinivasa G. Narasimhan. “Occlusion-Net: 2D/3D Occluded Keypoint Localization Using Graph Networks”. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2019, pp. 7326–7335.
- [41] N. Dinesh Reddy, Minh Vo, and Srinivasa G. Narasimhan. *Supplementary Material: Occlusion-Net: 2D/3D Occluded Keypoint Localization Using Graph Networks*. <http://www.cs.cmu.edu/~ILIM/projects/IM/CarFusion/cvpr2019/>. 2019.
- [42] Joseph Redmon et al. “You Only Look Once: Unified, Real-Time Object Detection”. In: *CoRR* abs/1506.02640 (2015). arXiv: 1506.02640. URL: <http://arxiv.org/abs/1506.02640>.
- [43] Shaoqing Ren et al. *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*. 2016. arXiv: 1506.01497 [cs.CV].
- [44] Grant Sanderson. *3Blue1Brown, Essence of Linear Algebra*. Youtube. 2016. URL: https://www.youtube.com/playlist?list=PLZHQObOWTQDPD3MizzM2xVFitgF8hE_ab.
- [45] Shashi Sathyanarayana. *A Gentle Introduction to Backpropagation*. 2014.
- [46] S. Skiena. *Calculated Bets: Computers, Gambling, and Mathematical Modeling to Win*. Cambridge University Press, 2001. ISBN: 9781280433412. URL: <https://books.google.de/books?id=fEDdugEACAAJ>.
- [47] ultralytics. *yolo V5*. <https://github.com/ultralytics/yolov5#citation>. 2020.
- [48] Magno Urbano. “Infinitesimal Calculus”. In: *Introductory Electrical Engineering with Math Explained in Accessible Language*. John Wiley Sons, Ltd, 2019. Chap. 2, pp. 9–17. ISBN: 9781119580164. DOI: <https://doi.org/10.1002/9781119580164.ch2>. eprint: <https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781119580164.ch2>. URL: <https://onlinelibrary.wiley.com/doi/abs/10.1002/9781119580164.ch2>.
- [49] Kilian Weinberger. *CS4780/CS5780: Machine Learning for Intelligent Systems*. <https://www.cs.cornell.edu/courses/cs4780/2018fa/>. 2018.

- [50] Zonghan Wu et al. “A Comprehensive Survey on Graph Neural Networks”. In: *IEEE Transactions on Neural Networks and Learning Systems* (2020), pp. 1–21. ISSN: 2162-2388. DOI: 10.1109/tnnls.2020.2978386. URL: <http://dx.doi.org/10.1109/TNNLS.2020.2978386>.
- [51] M. Zeeshan Zia et al. “Detailed 3D Representations for Object Recognition and Modeling”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35.11 (2013), pp. 2608–2623. DOI: 10.1109/TPAMI.2013.87.
- [52] Jiawei Zhang. *Gradient Descent based Optimization Algorithms for Deep Learning Models Training*. 2019. arXiv: 1903.03614 [cs.LG].
- [53] Jie Zhou et al. *Graph Neural Networks: A Review of Methods and Applications*. 2019. arXiv: 1812.08434 [cs.LG].
- [54] Zhengxia Zou et al. *Object Detection in 20 Years: A Survey*. 2019. arXiv: 1905.05055 [cs.CV].