TECHNISCHE UNIVERSITÄT
BERGAKADEMIE FREIBERG
Die Ressourcenuniversität. Seit 1765.

Faculty of Mechanical, Process and Energy
Institute of Mechanics and Fluid Dynamics
Applied Mechanics-Solid Mechanics

Course of study: MA Computational Materials Science

MASTER THESIS

MACHINE LEARNING APPROACHES TO THE INVERSE PROBLEM OF IDENTIFYING CRACKS FROM ELECTRICAL SIGNALS IN STRUCTURAL HEALTH MONITORING

Submitted by
Vundurthy, Padmanabha Pavan Chandra
Matriculation Nr. 62750

to obtain the academic degree
Master of Science
(M. Sc.)

1st Examiner (Supervisor):	Prof. Dipl.-Ing. Björn Kiefer, Ph.D.
2nd Examiner (Reviewer):	PD Dr.-Ing. habil. Geralf Hütter

in partial fulfillment of the requirements for the degree of Masters in Computational Materials Science at the Technische Universität Bergakademie Freiberg.

June 22, 2021

Master Thesis : Declaration of Authorship

I, Padmanabha Pavan Chandra Vundurthy, here by declare that this thesis and the work presented in it are my own and has been generated by me as the result of my own original research.
I confirm that:

- This thesis is carried out while in candidature for a Master's degree in Computational Materials Science at the Technische Universität Bergakademie Freiberg;
- No part of this thesis has previously been submitted by me for a degree or any other qualification at this University or at any other institution;
- When the published or unpublished work of others are consulted, it is always clearly attributed, with an accurate and complete reference;
- When the work from other authors is quoted, the source is always mentioned. With the exception of such quotations, this thesis is entirely my own work;

Vundurthy, Padmanabha Pavan Chandra

Freiberg, 22nd June, 2021

Abstract

Through the past decade, different methods have been developed for monitoring the structural integrity of any specimen including those parameters that define and evaluate crack. In one approach there are methods detecting just the size and location of the cracks and in another approach there are parameters quantifying stress state at the crack tip. For many years, it was not feasible to monitor structural health continuously during the service period since the crack growth must be excluded. With the recent development of new sensor concepts exploiting the piezoelectric polymer films as sensors for the experimental determination of fracture mechanical quantities, they compensate the disadvantages of classical resistance strain gauges and make it feasible for continuous structural health monitoring. With the advent of nanosensors, which are highly sensitive, highly resistive and have a miniaturized active area, the idea to adapt the sensor concept over to nanosensors has been awakened. If the loading situation at the crack tip is to be determined based on the measured strain fields from nanosensors, then there is a necessity of the inverse boundary value problem from the theory of elasticity. Many researches depict that solving of non-linear equations could be a laborious process with minimum satisfaction or no reliability on the solution of the inverse problem. At this juncture, if there could be a method which reads the data and fits it a function and supplies the most reliable output. There arised a scope for the application of Artificial Neural Networks, more decessively, Deep Neural Networks which are proven by current researchers as a reliable method to solve realistic problems.

In this thesis, Neural Networks are implemented to build a model based on virtually generated training data from Griffith near crack tip solutions and optimized to give reliable performance. This model is evaluated using virtual experiments. Different layouts of sensor arrangement are also compared to suggest better layout and model performance to predict the desired output. In this case, the strain field is taken as the input from the sensors and corresponding SIF-I, SIF-II, T, Crack angle and position are trained as the outputs. The model behaves reliably and predicts the results with minimum loss.

Acknowledgements

First, I would like to express my sincere gratitude to my supervisor Dr.-Ing. habil. Geralf Hütter for his guidance and patient advice throughout this research. I am grateful to him for giving me this wonderful opportunity to implement the knowledge that I have acquired through my Master's studies. I will always be indebted to Dr. Geralf Hütter for the valuable resources and skills that I have acquired during this Master's Thesis.

I am also grateful to Prof. Dipl.-Ing. Björn Kiefer, Ph.D. for his intriguing lectures and continuous support through the thesis. I am indebted to all the faculty of IMFD and MiMM for their amazing lectures and support through the course of Computational Materials Science.

Furthermore, I would like to acknowledge my mother, father and brother who have supported me in every step of my life. Without their support and faith in me, this journey would'nt have been possible. I am especially grateful to my brother, Bhaskar, for his continuous motivation and emotional support.

Finally, I am grateful to my fellow students, whose insights and discussions helped me think out of the box and achieve better performance.

Padmanabha Pavan Chandra Vundurthy

List of Abbreviations and Symbols

Abbreviations

- ADAM - Adaptive Moment Estimation
- AEs - Autoencoders
- AI - Artificial Intelligence
- ANN - Artificial Neural Network
- API - Application Programming Interface
- DAG - Directed Acyclic Graph
- DIM - Displacement Interpolation Method
- DL - Deep Learning
- EPFM - Elastic-Plastic Fracture Mechanics
- GANs - Generative Adversarial Networks
- ICA - Independent Component Analysis
- LEFM - Linear Elastic Fracture Mechanics
- MAE - Mean Absolute Error
- ML - Machine Learning
- MLP - Multilayer Perceptron Networks
- MR - Machine Reasoning
- MSE - Mean Squared Error
- NN - Neural Networks
- PCS - Principal Component Analysis
- PVDF - Polyvinylidene Fluoride

- SDOF - Single degree of freedom method
- SGD - Stochastic Gradient Descent
- SIF - Stress Intensity Factor
- SIFT - Scale-Invariant Feature Transform
- SIMD - Single Instruction Multiple Data

Symbols

- w_{ij} - Weights of i^{th} layer and j^{th} unit
- b_{ij} - Bias of i^{th} layer and j^{th} unit
- $\sum$ - Transfer Function or Linear Aggregator
- θ - Activation Threshold
- g - Activation Function
- x' - X coordinate, Sensor Coordinate System
- y' - Y coordinate, Sensor Coordinate System
- x_0 - X coordinate, Crack Origin
- y_0 - Y coordinate, Crack Origin
- β - Crack Angle
- α - Inner Hexagon Angle
- ε - Strain
- ϕ - Polar coordinate, angle
- r - Polar coordinate, radius
- x - X Coordinate, Crack tip position
- y - Y Coordinate, Crack tip position

- μ - Shear modulus
- E - Elastic modulus
- K_I - Stress intensity factor for mode-I loading
- K_{II} - Stress intensity factor for mode-II loading
- L - Number of layers
- lr - Learning Rate
- $lrscheduler_i$ - Learning rate scheduler for i epochs
- σ - Stress

Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

Fracture Mechanics is the field that studies, prevents and maintains the safety of any structure in the presence or initiation of a crack , balancing the design and usability of any engineered structure. The consequences of fracture could be minor or major related to its damage and health hazard. Fracture mechanics proposes and estimates solutions for questions belonging to designing components and processes against fracture. As Alan T. Zehnder quotes in his book "Fracture Mechanics" that the driving forces are the loads at the crack tip, expressed in terms of the stress intensity factor and the energy available to the crack tip. The resistance of the structure or material to fracture is expressed in terms of toughness. The main criteria for fracture can be stated as a balance of the crack tip loads and the material's fracture resistance in other words"toughness."

Fatigue Crack Growth

There are fracture processes that occur far below the critical load and develop in a stable manner with a very low rate of growth . In order to describe them, the term "subcritical crack growth" was introduced. The most crucial form of appearance is "fatigue crack growth", whereby the crack gradually grows under alternating loads. Generally, fatigue occurring in materials is subjected to repeated cyclic loading can be defined as a progressive failure due to crack initiation (stage I), crack growth (stage II), and crack propagation (stage III) or instability stage. The crack initiation has the possibility to occur along the slip direction due to a local maximum shear stress. After many cycles, the crack may change in direction when the maximum principal normal stress (near crack tip) governs crack growth.

In general, fatigue is a form of failure caused by alternating or cyclic loads over a period of time. Therefore, as stated in [5], fatigue is a time-dependent failure mechanism related to micro-structural features. In summary, the fluctuating loading condition is not a continuous failure process as opposed to cyclic loading. The former is manifested in bridges, aircraft, and machine components, while the latter requires a continuous constant or variable stress amplitude until fracture occurs.

Structural Health Monitoring(SHM)

The purpose of SHM is to supply regular information about the condition of a structure so that assessment of the structural integrity could be estimated at any circumstance and respective precautionary measures could be implemented. A SHM system typically consists of an array of sensors [6] for periodic inspections of the structure. Main features of interest are initially extracted from the data provided by the sensors and then analyzed to estimate the current state of the structure. The optimum use of data acquired from the SHM can be possible by using it for the purpose of "Damage prognosis" which is defined in [6] as an estimate of a structure's remaining useful life. The underlying concept is based on detecting and characterizing current damage level as determined from the SHM system and evaluating it in terms of failure mechanics and a damage growth law. The aim is to determine whether damage in a structure is sufficiently small that failure can be precluded with a high degree of certainty within a preset interval. In that situation the structural system may be allowed to function for that interval; otherwise, the structure must be taken for further inspection, repair, or replacement.

1.1 MOTIVATION

Cracks in components can emerge from manufacturing processes or nucleate due to fatigue of material. Though, cracks can be tolerated in many cases until they reach a critical size. The manual monitoring of cracks within certain maintenance intervals is costly. Automated health monitoring solutions like integrated arrays of sensors, e. g. strain gauges, thus promise a considerable increase of efficiency. However, such methods require to solve the inverse problem of computing the crack location and the crack-tip loading (stress-intensity factor) from the electric signals. Different methods have been employed to solve this problem. The scope of the present thesis is to apply neural networks (NN) for solving this inverse problem and to verify this methodology with virtual experiments. In Figure 1.1, a sensor concept to determine crack loacation and angle from the strains measured at crack tip is presented in [1] using a PVDF film.

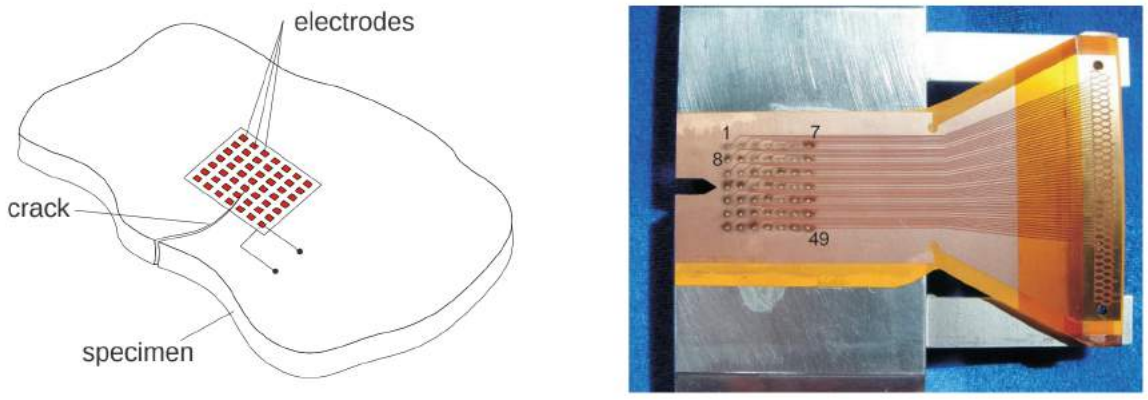


Figure 1.1: Sensor array for structural health monitoring of cracks [1]

Subtasks for the thesis:

- Theoretical construction of the strain fields for arbitrarily located cracks and loadings.
- Computation of respective virtual electric signals of nano strain gauges as training data for NN.
- Optimization of the structure of the NN and the sampling of the training data.
- Comparison of different layouts of sensor arrays.

1.2 SOLUTION FOR INVERSE PROBLEM

The Nano strain gauge captures the strain field at the specimen surface near the crack tip. If the loading situation at the crack tip is to be determined based on these measured strains, the solution of an inverse boundary value problem of the theory of elasticity is required [2]. If the strains at the crack tip are known, crack position with respect to the sensors, K-factors, T-stress and optionally further terms of the crack solution are the unknown quantities to be determined. In this thesis, the problem is limited to estimating the crack position on the specimen, cosine and sine of crack angle, K_I , K_{II} and T-stress. Deep Neural Networks is employed to make solve this inverse problem. There are seven unknowns requiring a system of output neurons to train the Neural Networks to predict these values. In this case, strain field from the nanosensors is given as input to the Neural Network.

However, the practical application of this sensor concept is focussed on the detection of unknown crack paths. This means that the crack position with respect to the sensor arrangement is not known and the coordinates of the measuring points in the crack coordinate system (x, y) thus are not available. Introducing a local coordinate system (x_0, y_0) related to the origin of the film, the position of any measuring point can be calculated by coordinate transformation to the crack coordinate system (x, y) [2]. As the coordinates of the sensors are known with respect to sensor arrangement, apart from the stress field, the fracture quantities are complemented by three more unknowns (x_0, y_0, β) describing the sensor position with respect to the crack faces.

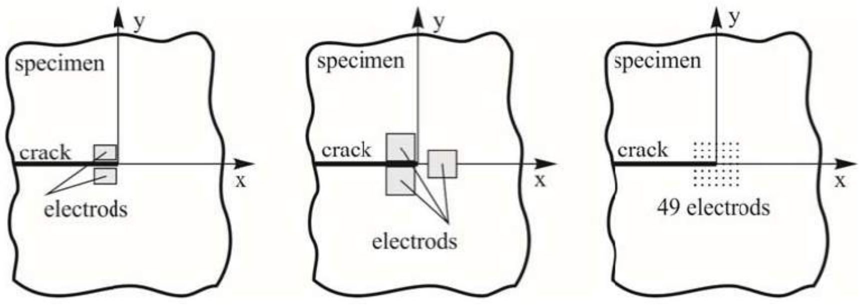


Figure 1.2: Position of piezoelectric elements in [2] and [3] with 49 electrodes

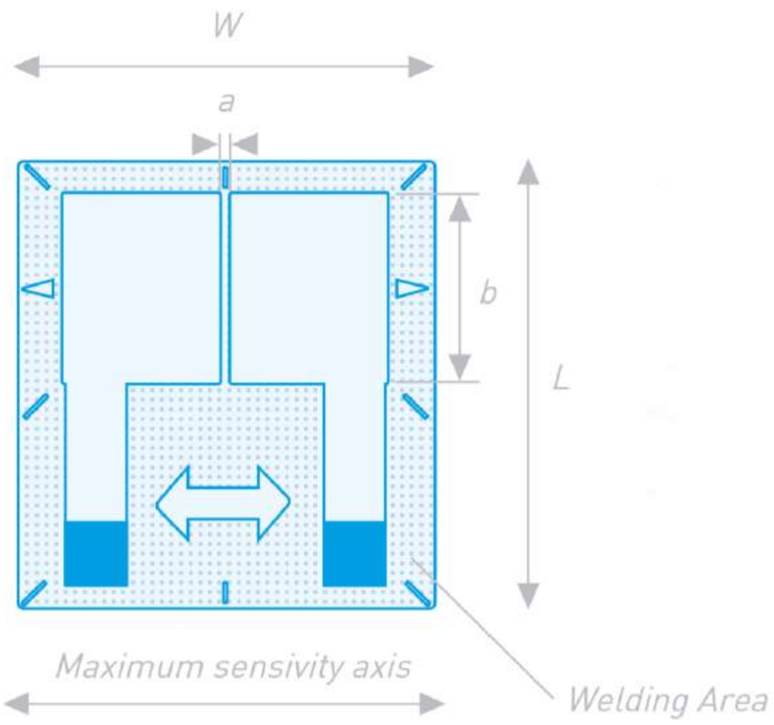
From Figure 1.2, the sensor concept using a PVDF film can be observed with the arrangement of 49 electrodes which is an inspiration for this thesis. This thesis focuses on a new approach using nanosensors at the crack tip with a similar electrode arrangement is implemented. Three sensor layout arrangements were compared, namely, regular 4×4 rectangular, 8×4 rectangular and hexagonal arrangement of sensor layout. The feasibility of applying neural networks to the solution of the problem under consideration is investigated by Dennis Bäcker, Andreas Ricoeur and Meinhard Kuna in [2]. It was suggested that this technique is frequently used in order to solve highly nonlinear inverse problems supported by [7] and [8]. Instead of actual sensor signals, the strain field measured through the sensors are artificially generated from analytical solution of the near crack tip solutions of GRIFFITH crack, and served as input values for training the neural network for the corresponding SIF's and crack position inspired from [9].

1.3 NANO STRAIN GAUGE

A strain gauge is an instrument employed to measure strain on an object. It was invented by Edward E. Simmons and Arthur C. Ruge in 1938 [10]. A general strain gauge composes of an insulating flexible backing which supports a metallic foil pattern. The gauge is fixed to the object using an adhesive. As the object is deformed, the foil is deformed, causing its electrical resistance to change. This resistance change, usually measured using a Wheatstone bridge, is related to the strain by the quantity known as the gauge factor [10]. A strain gauge takes advantage of the physical property of electrical conductance and its dependence on the conductor's geometry. When an electrical conductor is stretched within the limits of its elasticity such that it does not break or permanently deform, it will become narrower and longer, which increases its electrical resistance. Conversely, when a conductor is compressed such that it does not buckle, it will broaden and shorten, which decreases its electrical resistance. From the measured electrical resistance of the strain gauge, the amount of induced stress may be inferred. The benefits of nanotechnologies are high sensitivity, high resistance and miniatured active area.

Nano strain gauge creates a possibility to design more efficient sensors (weight, force, pressure, torque) without changing test specimen. By application of autonomous patches of many nano strain gauges on structures, an early detection of potential failures and prevention by anticipating operating maintenance could be performed. The dimensions in Figure 1.3 are W-Width of the substrate=6.5mm, a-Length of the active area=0.1mm, L-Length of the substrate=7mm, b-Width of the active area=3mm.

(a) Nano strain gauge from *Nanolike*



(b) Technical Specifications of a Nanosensor

Figure 1.3: Nano Strain Gauge [4]

The sensor layout used in [2] has 49 sensors having a 7×7 arrangement. In the current implementation, the concept is used to create a sensor layout of 4×4 regular rectangular arrangement of 16 sensors around the crack tip. This sensor layout is used to generate virtual training data that is required to train the Neural Network. The model is optimized to give the best performance (lowest model loss). This model is later compared with two other sensor arrangement layouts, namely,

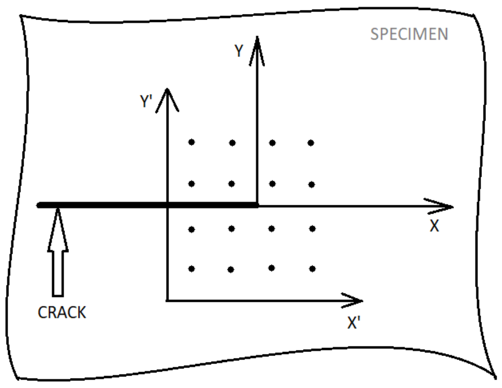


Figure 1.4: Sensor array for regular 4x4 Rectangular Layout

- Rectangular 8x4 Layout (32 Sensors)

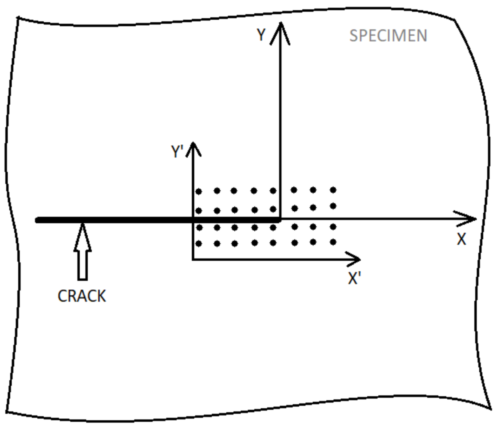


Figure 1.5: Sensor array for 8x4 Rectangular Layout

- Hexagonal Layout (13 Sensors)

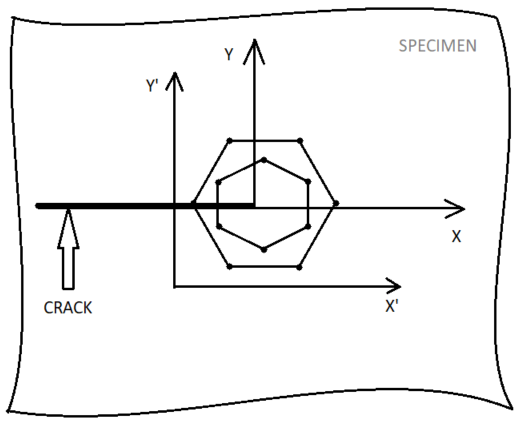


Figure 1.6: Sensor array for Hexagonal Arrangement

1.4 THESIS OUTLINE

Chapter 2 gives a brief introduction into the field of AI, Machine Learning and Deep Learning and few important differences in their concepts which have been notes while preparing for this thesis. The basic concepts with their root in solving nonlinear problems in real world applications are explained with an outlook for upcoming chapters.

Chapter 3 summarizes and explains how the Neural Network model is built with respect to the current inverse problem. It explains the model architecture, summary and tools used to compute the model. The difference between sequential and functional API which is a crucial tool used for multiple regression analysis is described in this chapter.

Chapter 4 generates the training data which is used in the current thesis on behalf of the sensor readings to train the Neural Network Model. This generation method is expained stepwise and with schematic representations of different layout of sensors useful in the thesis. The data is generated for different layouts with different dataset sizes for model performance studies.

Chapter 5 fits the training data to a Neural Network model and compares various parameters that crucially effect the performance of the model. Different hyperparameter tuning and optimization methods are performed over the model to improve its efficiency and generality to different datasets. Optimized performance from the model is extracted and presented.

Chapter 6 compares the performance of different sensor layouts, namely, 4×4 rectangular, 8×4 rectangular and hexagonal sensor arrangements along with 3×3 and 5×5 regular rectangular layouts.

Conclusion gives the summary and future scope of the thesis and the optimized parameters for the best model behavior.

Chapter 2

NEURAL NETWORKS AND DEEP LEARNING

2.1 THEORETICAL INTRODUCTION

Around late 1940s, both the appearance of digital computers and the development of modern theories for learning and neural processing occurred simultaneously. Since that time, the digital computer has been used as a tool to model individual neurons as well as clusters of neurons, which are called neural networks [11]. During 1950s, the first examples of these new systems appeared with a most common historical reference to the work done by Frank Rosenblatt on a device called the perceptron. From 1969 to 1980s, the field barely survived by the efforts of persistent researchers [12] like Marvin Minsky and Seymour Papert and later supported by [13], a book on "Parallel Distributed Processing". During the last decade (2011-2020) there has been a major advancement in the field of Neural Networks and their applications reach various fields. From image classification to face detection and from linear regression to building models for business investment, Neural Networks is solving all kinds of realistic problems.

ARTIFICIAL INTELLIGENCE is referred as an umbrella term to describe problems from simple analytics to advanced learning algorithms. During 1950, Alan Turing published a paper on the concept of "thinking" and defined a task to observe if a machine was able to achieve a level of reasoning that can be called AI. The first Artificial Neural network (ANN) was also created during these days. Between the 1950s and the 1970s, the world saw the first new big era of discovery in AI, with applications in algebra, geometry, language, and robotics.

John McCarthy defines in his paper [14], that “AI is the science and engineering of making intelligent machines, especially intelligent computer programs. It is related to the similar task of using computers to understand human intelligence, but AI does not have to confine itself to methods that are biologically observable”. It studies ways to build intelligent programs and machines that can creatively solve problems, which has always been considered a human prerogative. According to IBM [15], “**Artificial intelligence** leverages computers and machines to mimic the problem-solving and decision-making capabilities of the human mind”. It is a field, which combines computer science and robust datasets, to enable problem-solving. It also encompasses sub-fields of machine learning and deep learning, which are frequently mentioned in conjunction with artificial intelligence. These disciplines are comprised of AI algorithms which seek to create expert systems which make predictions or classifications based on input data.

AI is generally divided into three categories [15]:

- Artificial Narrow Intelligence - Weak AI (Narrow Capability)
- Artificial General Intelligence - Strong AI (General Capability)
- Artificial Super Intelligence - (Transcendent Capability)

Weak, or narrow AI, performs a particular task well, but it will not pass for human in any field outside of its defined capacities. General, or strong AI, is the point when machines behave like human. They make their own decisions and learn without any human input. Not only are they competent in solving logical tasks but they also have emotions [15]. Super Intelligence, is to make machines, way ahead of humans. Smart, wise, creative, with excellent social skills. Its goal to either make humans’s lives better or destroy them all.

There are numerous, real-world applications of AI systems today. Below are some of the most common examples[16]:

- Analysis of images acquired from artificial satellites.
- Speech and writing pattern classification.
- Face recognition with computer vision
- Control of high-speed trains.
- Stocks forecasting on financial market.
- Anomaly identification on medical images.
- Automatic identification of credit profiles for clients of financial institutions.
- Control of electronic devices and appliances, such as washing machines, microwave ovens, freezers, coffee machines, frying machines, video cameras,and so on.

The tasks that data scientists are focusing on right now (and which can help to create general and superintelligence) are [17]:

- **Machine Reasoning:** MR systems have some information at their disposal, for example, a database or a library. Using deduction and induction techniques they can formulate some valuable insights based on this information. It can include planning, data representation, search and optimisation for AI systems.
- **Robotics:** This field of science concentrates on building, developing, and controlling robots from roombas to intelligent androids.
- **Machine Learning:** It is the study of algorithms and computer models used by machines in order to perform a given task.

Machine learning is a subset of artificial intelligence (AI) that provides systems the ability to automatically learn and improve from experience without being explicitly programmed. In ML, there are different algorithms (e.g. neural networks) that help to solve problems.

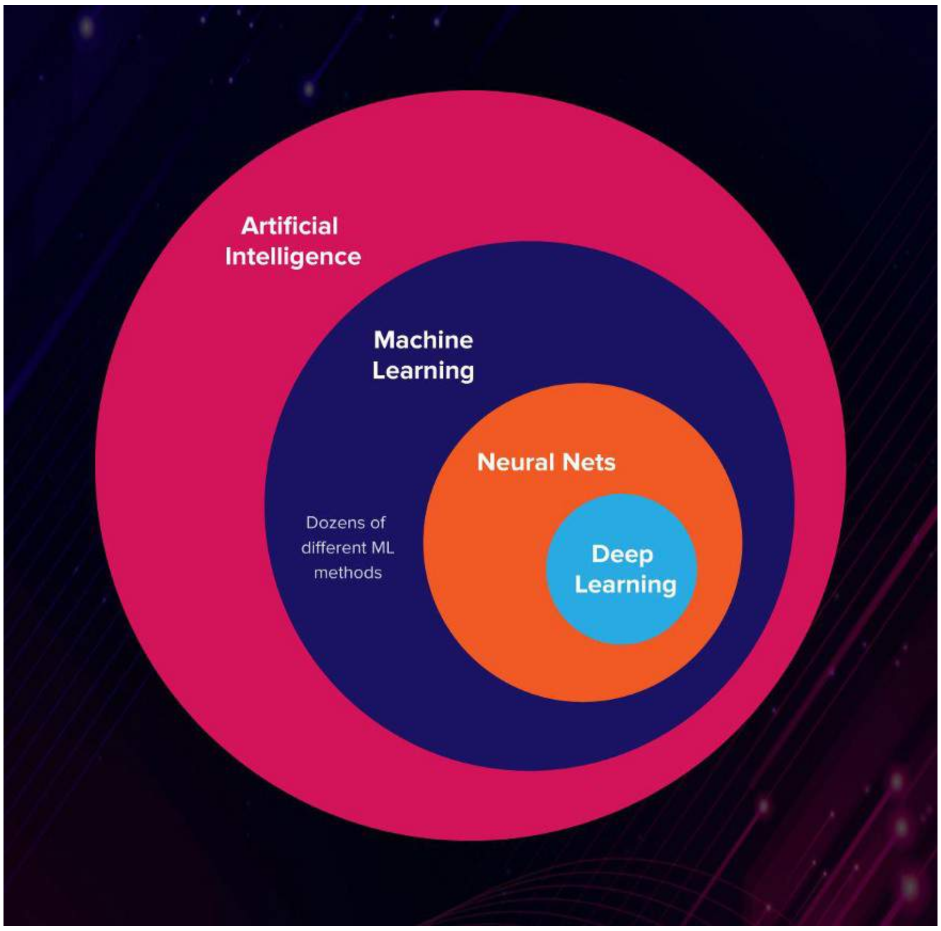


Figure 2.1: Difference between AI, ML and DL [17]

2.1.1 AN OVERVIEW OF MACHINE LEARNING

Machine Learning is the science (and art) of programming computers so they can learn from data. “A computer program is said to learn from experience E with respect to some task T and some performance measure P, if its performance on T, as measured by P, improves with experience E” by Tom Mitchell, 1997.

- Supervised learning
- Unsupervised learning
- Semi-supervised learning
- Reinforcement learning

Supervised Learning is the most common form of ML applied to business processes. These algorithms find a good approximation of the function that is mapping inputs and outputs. To accomplish that, it is necessary to provide both input and output values to the algorithm and it will find a function that minimizes the errors between predictions and actual output. The learning phase is called training [18]. After a model is trained, it can be used to predict the output from unseen data. This phase is commonly regarded as scoring or predicting, which is depicted in the following diagram:

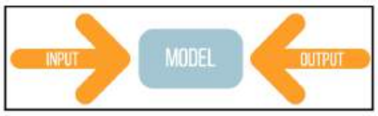


Figure 2.2: Supervised Learning [18]

Unsupervised learning works with unlabeled data, so the actual output is not required, only the input. It tries to find patterns in the data and reacts based on those commonalities, dividing the input into clusters: Usually, unsupervised learning is often used in conjunction

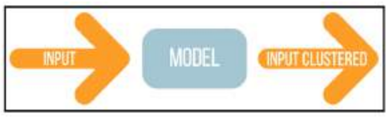


Figure 2.3: Unsupervised Learning [18]

with supervised learning to reduce the input space and focus the signal in the data on a smaller number of variables, but it has other goals as well [19]. From this point of view, it is more applicable than supervised learning as sometimes tagging the data is expensive or not reliable. Common unsupervised learning techniques are clustering and principal component analysis (PCA), independent component analysis (ICA), and some neural networks such as Generative Adversarial Networks (GANs) and Autoencoders (AEs).

Semi-supervised learning is a technique in between supervised and unsupervised learning and the aim is to reduce the cost of gathering labeled data by extending a few labels to similar unlabeled data. Some generative models are classified semi-supervised approaches.

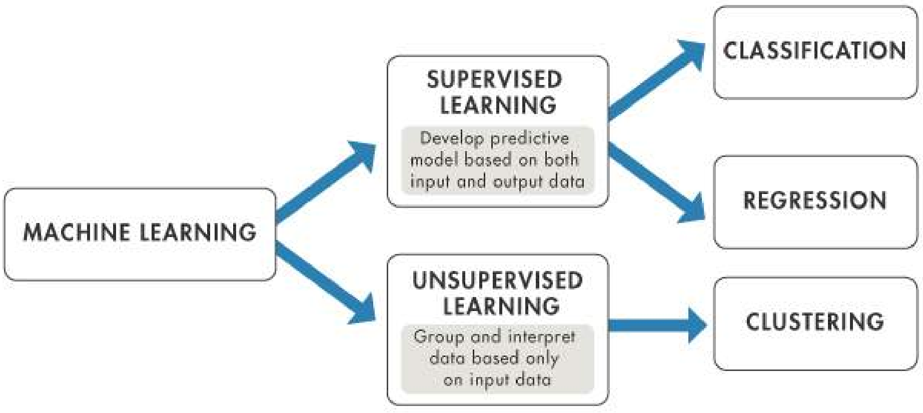

MACHINE LEARNING
SUPERVISED LEARNING
Develop predictive model based on both input and output data
UNSUPERVISED LEARNING
Group and interpret data based only on input data
CLASSIFICATION
REGRESSION
CLUSTERING

Figure 2.4: Supervised vs. Unsupervised Learning [19]

Semi-supervised learning can be divided into transductive and inductive learning. Transductive learning is when we want to infer the labels for unlabeled data. The goal of inductive learning is to infer the correct mapping from inputs to outputs [19].

Reinforcement learning is the most distinct category, with respect to others. The policy is learned by an agent who uses it to take actions in an environment. The environment then returns feedback, which the agent uses to improve its policy [19]. The feedback is the reward for the action taken and it can be a positive, null, or negative number, as shown in the following diagram:

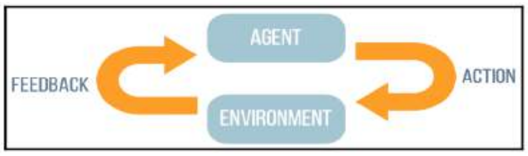


Figure 2.5: Reinforcement Learning [19]

To summarize, Machine Learning is suitable for:

- Problems for which existing solutions require a lot of hand-tuning or long lists of rules: one Machine Learning algorithm can often simplify code and perform better.
- Complex problems for which there is no good solution at all using a traditional approach: the best Machine Learning techniques can find a solution.
- Fluctuating environments: a Machine Learning system can adapt to new data.

- Getting insights about complex problems and large amounts of data.

Deep Learning or deep neural learning, is a subset of machine learning, which uses the neural networks to analyze different factors with a structure that is similar to the human neural system. “Deep” in deep learning refers to a neural network comprised of more than three layers—which would be inclusive of the inputs and the output—can be considered a deep learning algorithm [16].

“Deep” machine learning can leverage labeled datasets, also known as supervised learning, to inform its algorithm, but it does not necessarily require a labeled dataset. It can ingest unstructured data in its raw form (e.g. text, images), and it can automatically determine the hierarchy of features which distinguish different categories of data from one another. Unlike machine learning, it does not require human intervention to process data.

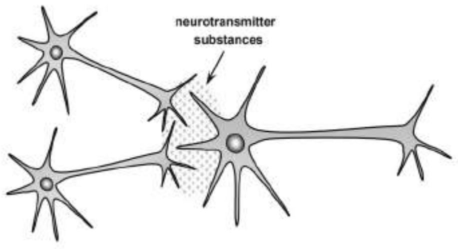


Figure 2.6: Synaptic connection between neurons [16]

Many kinds of tasks can be solved with the knowledge of ML and DL, namely (from[20] and [21]):

- **Classification:** In a classification problem, those outcomes are labels that could be applied to data: All classification tasks depend upon labeled datasets; that is, humans must transfer their knowledge to the dataset in order for a neural network to learn the correlation between labels and data. This is known as supervised learning. Few types of classification problems:
  - Detect faces, identify people in images, recognize facial expressions (angry, joyful)
  - Identify objects in images (stop signs, pedestrians, lane markers. . .)

  - Recognize gestures in video
  - Detect voices, identify speakers, transcribe speech to text, recognize sentiment in voices
  - Classify text as spam (in emails), or fraudulent (in insurance claims); recognize sentiment in text (customer feedback).

- **Classification with missing inputs:** Classification becomes even more challenging if the computer program is not guaranteed that every measurement in its input vector will always be provided. This kind of situation arises frequently in medical diagnosis, because many kinds of medical tests are expensive or invasive.

- **Predictive Analytics: Regressions:** Given a time series, deep learning may read a string of number and predict the number most likely to occur next.
  - Hardware breakdowns (data centers, manufacturing, transport)
  - Health breakdowns (strokes, heart attacks based on vital stats and data from wearables)
  - Customer churn (predicting the likelihood that a customer will leave, based on web activity and metadata)
  - Employee turnover (ditto, but for employees)

- **Clustering:** Clustering or grouping is the detection of similarities. Deep learning does not require labels to detect similarities. Learning without labels is called unsupervised learning. Unlabeled data is the majority of data in the world [20]. One law of machine learning is: **the more data an algorithm can train on, the more accurate it will be. Therefore, unsupervised learning has the potential to produce highly accurate models**.
  - Search: Comparing documents, images or sounds to surface similar items.
  - Anomaly detection: The other side of detecting similarities is detecting anomalies, or unusual behavior. In many cases, unusual behavior correlates highly with things to detect and prevent, such as fraud.

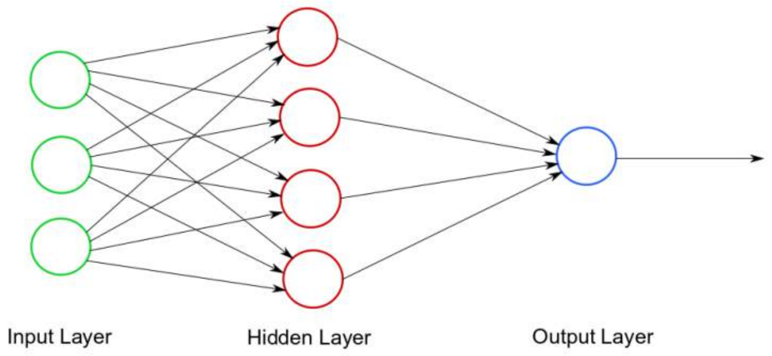


Figure 2.7: Artificial Neural network inspired from human neuron

Feature Engineering is the process of generating fresh features by transforming existing ones. In traditional ML, this feature is plays a role but not important in deep learning. Traditionally, the data scientists or the researchers would apply their domain knowledge and come up with a smart representation of the input that would highlight the relevant feature and make the prediction task more accurate [16]. For example, before the advent of deep learning, traditional computer vision required custom algorithms that were extracting the most relevant features, such as edge detection or **Scale-Invariant Feature Transform (SIFT)**.

One of the great advantages of using deep learning is that is not necessary to hand craft these features, but the network will do the job. A glimpse of working of a Neural Network is presented in Figure 2.8.

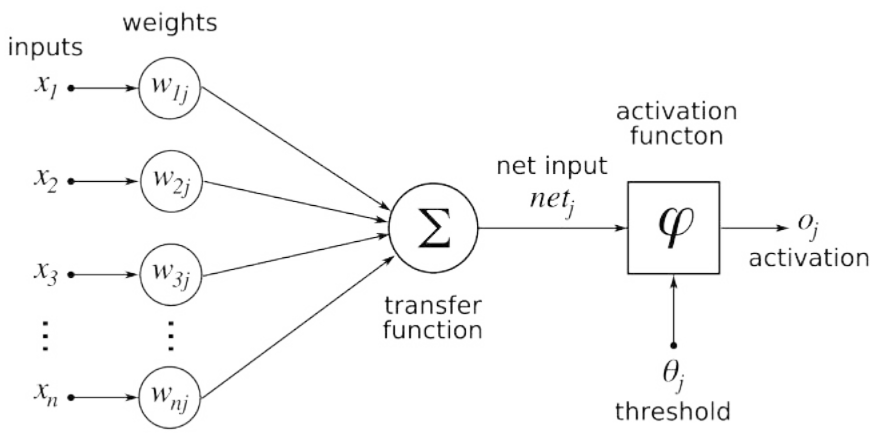


Figure 2.8: Artifical Neuron–Source: Chrislb Wikipedia

2.2 NEURAL NETWORKS: ARCHITECTURE

Neural Networks interprets sensory data through a kind of machine perception, labeling or clustering raw input. The patterns they recognize are numerical, contained in vectors, into which all real-world data, be it images, sound, text or time series, must be translated.

Deep learning maps and finds correlation of inputs to outputs. It is considered as a "universal approximator", because it can learn to approximate an unknown function *f(x)* $= y$ between any input x and any output y, assuming they are related at all (by correlation or causation, for example). In the process of learning, a neural network approximates a suitable function *f*, or the correct manner of transforming x into y.

In summary, **Classification** is a task where the output variable can assume a finite amount of elements, called categories. An example of classification would be classifying different types of flowers (output) given the sepal length (input). Classification can be further categorized in more sub types:

- **Binary classification:** The task of predicting whether an instance belongs either to one class or the other.
- **Multiclass classification:** The task (also known as multinomial) of predicting the most probable label (class) for each single instance.
- **Multilabel classification:** When multiple labels can be assigned to each input.

Regression is a task where the output variable is continuous. Here are some common regression algorithms:

- **Linear regression:** This finds linear relationships between inputs and outputs.
- **Logistic regression:** This finds the probability of a binary output.

Perceptron is a mathematical model of a biological neuron. While in actual neurons the dendrite receives electrical signals from the axons of other neurons, in the perceptron these electrical signals are represented as numerical values. As in biological neural networks, this output is fed to other perceptrons [16] and [22]. The simplicity of the Perceptron network is due to its condition of being constituted by just one neural layer, hence having a single artificial neuron in this layer. Even though the Perceptron was a simple network, at the time it was proposed, it attracted several researchers who aspired to investigate this promising field, receiving special attention of the scientific community.

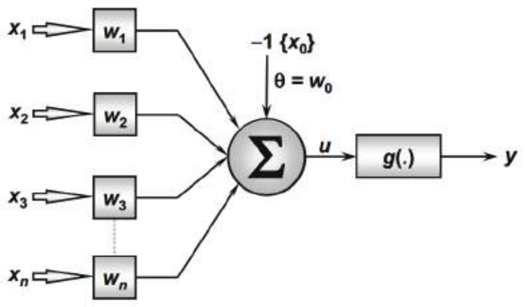


Figure 2.9: Illustration of Perceptron Network [16]

Single layer perceptron or a neural network belongs to the class of single-layer feed forward architectures, because the information that flows in its structure is always from the input layer to the output layer, without any feedback from the output produced by its single output neuron. From the analysis of Figure 2.9, it is possible to see that each one of the x_i inputs is initially pondered by synaptic weights in order to quantify the importance of the inputs on the functional goals of the neuron, whose purpose is to map the input/output behavior of the process in question. In sequence, the value resulting from the composition of all inputs pondered by weights, added to the activation threshold h, is used as an argument for the activation function, whose value is the output y produced by the Perceptron, where x_i is a network input, w_i is the weight (pondering) associated with the i^{th} input, h is the activation threshold (or bias), $g(.)$ is the activation function and u is the activation potential [16].

Multilayer Perceptron Networks (MLP) features, at least, one intermediate (hidden) neural layer, which is placed between the input layer and the respective output layer. Consequently, MLP networks have at least two neural layers, and their neurons are distributed among the intermediate and output layers. MLP networks are also known for their wide range of application in several problems and are also considered one of the most versatile architectures regarding applicability [16]. Among these potential areas, the most important are the following:

- Universal function approximation (curve fitting).
- Pattern recognition.
- Process identification and control.
- Time series forecasting (prediction).
- System optimization.

MLP networks belong to the multiple layer feed forward architecture, whose training is performed with a supervised process.

Training Process of the Perceptron The adjustment of weights and threshold in the Perceptron is made through supervised training, meaning that the respective desired output (response) must exist for each sample of the input signal. Since the Perceptron is usually used on pattern recognition problems, and considering that its output can assume just two possible values, then each output is associated with one of the two classes that are being identified [16].

The adjustment of Perceptron's weights and thresholds, in order to classify patterns that belong to one of the two possible classes, is performed by the use of Hebb's learning rule (Hebb 1949). In short, if the output produced by the Perceptron coincides with the desired output, its synaptic weights and threshold remain unchanged (inhibitory condition); otherwise, in the case the produced output is different from the desired value, then its synaptic weights and threshold are adjusted proportionally to its input signals (excitatory condition). This process is repeated sequentially for all training samples until the output produced by the Perceptron is similar to the desired output of all samples. However, for computational implementation, it is more convenient to approach the previous equations in their vector form. As the same adjustment rule is applied to both the synaptic weights and threshold, it is possible to insert the threshold value h within the synaptic weight vector.

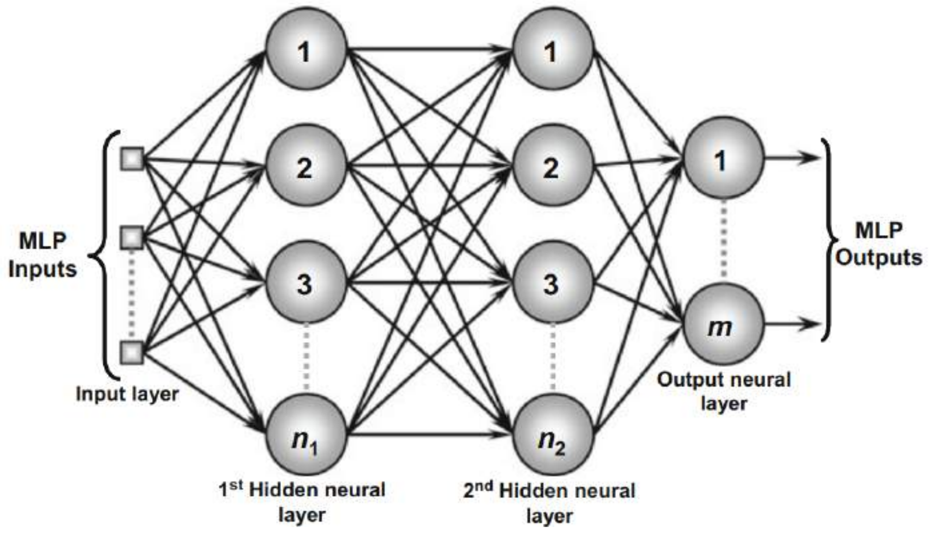


Figure 2.10: Illustration of MLP [16]

2.2.1 WORKING PRINCIPLE

By analyzing Figure 2.10, it is possible to see that each input of the network, which represents the signals from a given application, will be propagated layer-by-layer toward the output layer. In this case, the outputs of neurons from first neural layer will be the inputs of neurons from the second hidden neural layer. For the scenario illustrated in Figure 2.10, the neurons of second hidden neural layer will be the inputs to neurons of output neural layer.

Thus, the propagation of the input signals of an MLP, independently of the number of intermediate layers, always flows in one direction, that is, from the input layer to the output neural layer [16]. Besides the existence of hidden layers on the MLP topology that the output layer can be composed of several neurons, and each of these neurons would represent one of the outputs of the process being mapped. Thus, if such process consists of m outputs, then the MLP network would also have m neurons on its last neural layer.

The stimuli or signals are presented to the network on its input layer. The intermediate layers, on the other hand, extract the majority of the information related to the system behavior and codify them using the synaptic weights and thresholds of their neurons, thus forming a representation of the environment where the particular system exists. Finally, the neurons of the output layer receive the stimuli from the neurons of the last intermediate layer, producing a response pattern which will be the output generated by the network [16].

As mentioned, the adjustment of the weights and thresholds of each neuron of an MLP network is made by employing a supervised training process, that is, for each sample of input data, there must be the respective desired output (response). The learning algorithm used during the training process of an MLP is called backpropagation.

2.2.2 Training of an MLP

The training process of MLP networks using the backpropagation algorithm, also known as the generalized Delta rule, is usually done by the successive application of two specific stages. These stages are illustrated in Figure 2.11, which shows an MLP configuration composed of two hidden layers, n signals on its input layer, $n1$ neurons in its first hidden neural layer, $n2$ neurons in its second hidden neural layer, and $n3$ signals associated with the output neural layer (third neural layer) [16].

Forward Propagation is the first stage, where the signals $x1, x2, \ldots, xn$ of a given sample from the training set are inserted into the network inputs and are propagated layer-by-layer until the production of the corresponding outputs. Thus,this stage intends solely in obtaining the responses from the network, taking into account only the current values of the synaptic weights and thresholds of its neurons, which will remain unmodified during the execution of this stage [16].

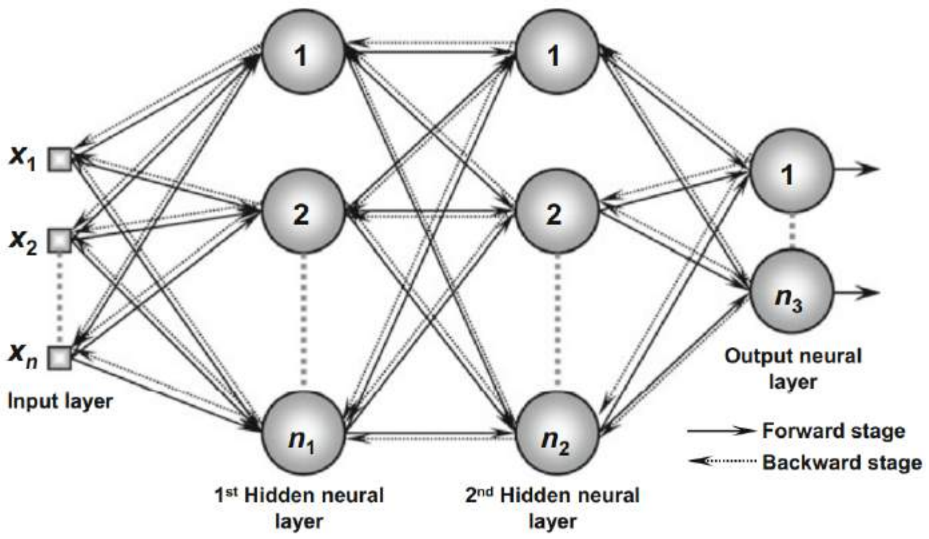


Figure 2.11: Training of an MLP [16]

Backward Propagation is the second stage. Unlike the first stage, the modifications (adjustments) of the synaptic weights and thresholds of all neurons of the network are executed during this stage. In summary, the successive application of the forward and backward stages allows the synaptic weights and thresholds of the neurons to be adjusted automatically in each iteration, also resulting in the gradual reduction of the sum of the errors produced by the network responses with respect to the desired responses [16].

Gradient Descent is the third step. An essential step in building a deep learning model is solving the underlying optimization problem, as defined by the loss function. This chapter covers Stochastic Gradient Descent (SGD), which is the most commonly used algorithm for solving such optimization problems. The relationship between network Error and each of those weights is a derivative, dE/dw, that measures the degree to which a slight change in a weight causes a slight change in the error. Each weight is just one factor in a deep network that involves many transforms; the signal of the weight passes through activations and sums over several layers, so we use the chain rule of calculus to march back through the networks activations and outputs and finally arrive at the weight in question, and its relationship to overall error [16].

The chain rule in calculus states that:

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx} \tag{2.1}$$

In a feedforward network, the relationship between the net's error and a single weight will look something like this:

$$\frac{\mathrm{d(Error)}}{\mathrm{d(weight)}} = \frac{\mathrm{d(Error)}}{\mathrm{d(activation)}} * \frac{\mathrm{d(activation)}}{\mathrm{d(weight)}} \tag{2.2}$$

That is, given two variables, Error and weight, that are mediated by a third variable, activation, through which the weight is passed, you can calculate how a change in weight affects a change in Error by first calculating how a change in activation affects a change in Error, and how a change in weight affects a change in activation.

The essence of learning in deep learning is nothing more than that: adjusting a model's weights in response to the error it produces, until you can't reduce the error any more.

Types of Gradient Descent [23]:

- **Batch Gradient Descent:** all available data is available at once and this version implies a high possibility of getting stuck. Since the gradient will be calculated using all the samples, and the variations will be minimal. As a general rule: for a neural network it is always positive to have an input with some randomness.

- **Stochastic Gradient Descent:** a single random sample is introduced on each iteration. The gradient will be calculated for that specific sample only, implying the introduction of the desired randomness, and making more difficult the possibility of getting stuck. The main problem with this version is its slowness, since it needs many more iterations. Additionally, it does not take advantage of the available computing resources.

- **Stochastic Mini-Batch Gradient Descent:** instead of feeding the network with single samples, N random items are introduced on each iteration. This preserves the advantages of the second version and also getting a faster training due to the parallelization of operations. This version of the algorithm is used and a value for N is chosen that provides a good balance between randomness and training time (and not too large for the available GPU memory).

Batch vs. Stochastic: In practice, stochastic approaches dominate over batch and are more commonly used. A seemingly non-intuitive fact about stochastic approaches is that not only is the gradient over few examples cheaper to compute but not getting the exact direction (using only a small number of examples) actually leads to better solutions. This is particularly true for large datasets with redundant information wherein the examples are not too different from each other [23]. Another reason for stochastic approaches performing better is the presence of multiple local minima with different depths. In a sense, the noise in the direction allows for jumping across the trenches while a batch approach will converge in the trench it started with.

Challenges with SGD [19]:

- **Local minima** are suboptimal solutions that trap any steepest descent approach and prevent the iterative procedure for making progress towards a better solution.
- **Saddle Points** are points where the gradient evaluates to zero but the point is not a local minimum. Saddle points are any steepest descent approach and prevent the iterative procedure for making progress towards a better solution. There is good empirical evidence that saddle points are much more common than local minima when it comes to optimization problems in a high number of dimensions (which is always the case when it comes to building deep learning models).
- **Learning Rate** is the amount that the weights are updated during training is referred to as the step size or the "learning rate." Specifically, the learning rate is a configurable hyperparameter used in the training of neural networks that has a small positive value, often in the range between 0.0 and 1.0. Too high a learning rate can cause the solution to bounce around and too low a learning rate means slow convergence (implying not

getting to a good solution in a given number of iterations). When it comes to loss functions with many parameters trained on sparse datasets, a single global learning rate for all parameters makes the problem of choosing a learning rate even more challenging. The learning rate controls how quickly the model is adapted to the problem. Smaller learning rates require more training epochs given the smaller changes made to the weights each update, whereas larger learning rates result in rapid changes and require fewer training epochs. A learning rate that is too large can cause the model to converge too quickly to a suboptimal solution, whereas a learning rate that is too small can cause the process to get stuck.

The challenge of training deep learning neural networks involves carefully selecting the learning rate. It may be the most important hyperparameter for the model.

2.2.3 REGRESSION

Logistic Regression is one of the most popular Machine learning algorithm that comes under Supervised Learning techniques. It can be used for Classification as well as for Regression problems, but mainly used for Classification problems. Logistic regression is used to predict the categorical dependent variable with the help of independent variables. The output of Logistic Regression problem can be only between the 0 and 1. Logistic regression can be used where the probabilities between two classes is required. Logistic regression is based on the concept of Maximum Likelihood estimation. According to this estimation, the observed data should be most probable. In logistic regression, we pass the weighted sum of inputs through an activation function that can map values in between 0 and 1. Such activation function is known as sigmoid function and the curve obtained is called as sigmoid curve or S-curve [23].

Linear Regression: is one of the most simple Machine learning algorithm that comes under Supervised Learning technique and used for solving regression problems. It is used for predicting the continuous dependent variable with the help of independent variables. The goal of the Linear regression is to find the best fit line that can accurately predict the output for the continuous dependent variable. If single independent variable is used for prediction then it is called Simple Linear Regression and if there are more than two independent variables then such regression is called as Multiple Linear Regression. By finding the best fit line, algorithm establish the relationship between dependent variable and independent variable. And the relationship should be of linear nature [23].

In this thesis, based on MLP, the Feed Forward Neural Networks is used for solving the inverse problem. The stepwise procedure for model fitting is explained in the below section.

2.2.4 ACTIVATION FUNCTIONS

The value of net input can be anything from -inf to +inf. The neuron does not really know how to bound to value and thus is not able to decide the firing pattern. Thus the activation function is an important part of an artificial neural network. It decides whether a neuron should be activated or not. Thus it bounds the value of the net input. The activation function is a non-linear transformation that we do over the input before sending it to the next layer of neurons or finalizing it as output.

$$\text{net input } = \sum(\text{ weight } * \text{ input}) \ + \text{ bias} \tag{2.3}$$

Types of Activation Function:

- Step function

$$\begin{aligned} f(x) &= 1, \text{ if x >= 0} \\ f(x) &= 0, \text{ if x < 0} \end{aligned} \tag{2.4}$$

- Sigmoid function

$$\frac{1}{(1+e^{-x})} \tag{2.5}$$

- ReLU

$$f(x) = \max(0, x) \tag{2.6}$$

- Leaky ReLU

$$\begin{aligned} f(x) &= ax, x < 0 \\ f(x) &= x, \text{ otherwise} \end{aligned} \tag{2.7}$$

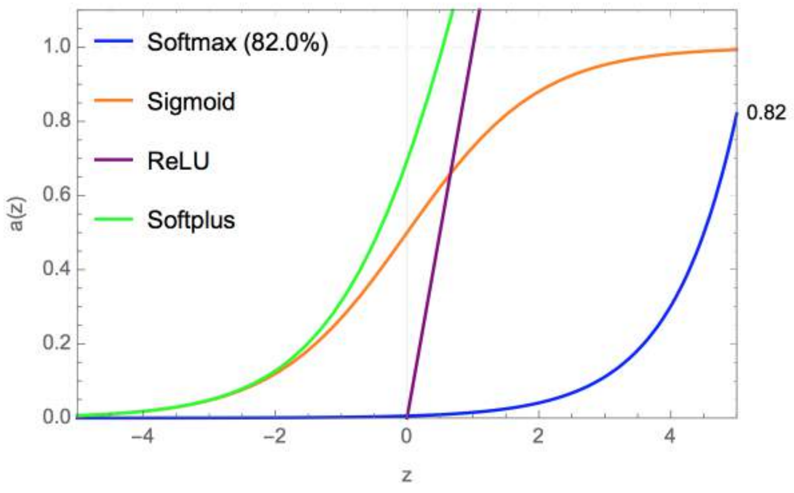


Figure 2.12: Activation Functions

Summary: In summary [23], it is possible to see that the artificial neuron is composed of seven basic elements, namely:

(a) *Input signals (x1, x2, . . . , xn)* are the signals or samples coming from the external environment and representing the values assumed by the variables of a particular application. The input signals are usually normalized in order to enhance the computational efficiency of learning algorithms.

(b) *Linear aggregator ($\sum$)* gathers all input signals weighted by the synaptic weights to produce an activation voltage.

(c) *Activation threshold or bias (θ)* is a variable used to specify the proper threshold that the result produced by the linear aggregator should have to generate a trigger value toward the neuron output.

(d) *Activation potential (u)* is the result produced by the difference between the linear aggregator and the activation threshold. If this value is positive, i.e. if u $\geq \theta$, then the neuron produces an excitatory potential; otherwise, it will be inhibitory.

(e) *Activation function (g)* whose goal is limiting the neuron output within a reasonable range of values, assumed by its own functional image.

(f) *Output signal (y)* consists on the final value produced by the neuron given a particular set of input signals, and can also be used as input for other sequentially interconnected neurons.

Batch size refers to the number of training examples utilized in one iteration. The batch size can be one of three options:

- Batch mode: The batch size is equal to the total dataset thus making the iteration and epoch values equivalent.
- Mini-Batch mode: The batch size is greater than one but less than the total dataset size. Usually, a number that can be divided into the total dataset size.
- Stochastic mode: The batch size is equal to one. Therefore the gradient and the neural network parameters are updated after each sample.

2.3 DATASETS

The data used to build the final model usually comes from multiple datasets. In particular, three datasets are commonly used in different stages of the creation of the model [24].

The model is initially fit on a training dataset, which is a set of examples used to fit the parameters (e.g. weights of connections between neurons in artificial neural networks) of the model. The model (e.g. a neural net or a naive Bayes classifier) is trained on the training dataset using a supervised learning method, for example using optimization methods such as gradient descent or stochastic gradient descent. In practice, the training dataset often consists of pairs of an input vector (or scalar) and the corresponding output vector (or scalar), where the answer key is commonly denoted as the target (or label). The current model is run with the training dataset and produces a result, which is then compared with the target, for each input vector in the training dataset. Based on the result of the comparison and the specific learning algorithm being used, the parameters of the model are adjusted. The model fitting can include both variable selection and parameter estimation. Successively, the fitted

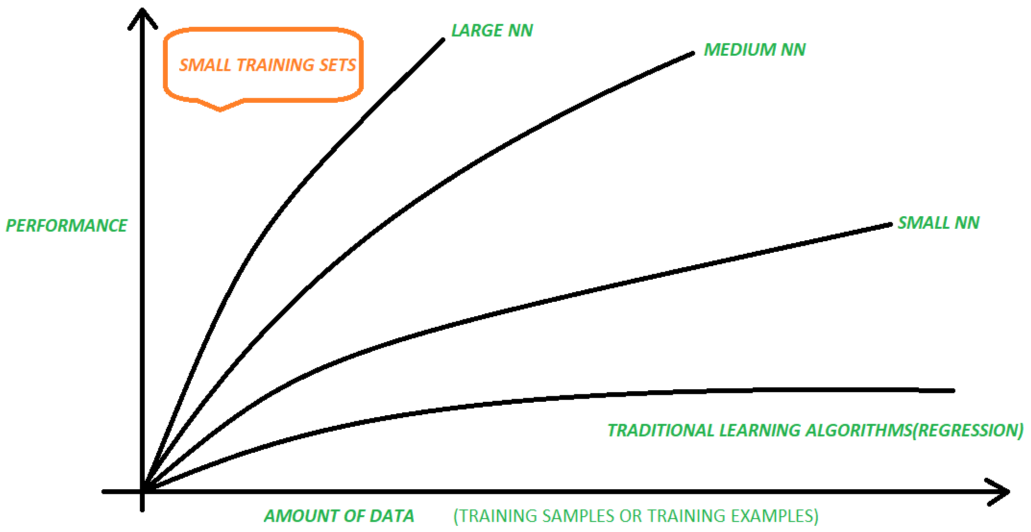


Figure 2.13: Performance vs. Training Data

model is used to predict the responses for the observations in a second dataset called the validation dataset. The validation dataset provides an unbiased evaluation of a model fit on the training dataset while tuning the model's hyperparameters (e.g. the number of hidden units—layers and layer widths—in a neural network). Validation datasets can be used for regularization by early stopping (stopping training when the error on the validation dataset increases, as this is a sign of overfitting to the training dataset). This simple procedure is complicated in practice by the fact that the validation dataset's error may fluctuate during training, producing multiple local minima. This complication has led to the creation of many ad-hoc rules for deciding when overfitting has truly begun.

Training Dataset A training dataset is a dataset of examples used during the learning process and is used to fit the parameters (e.g., weights) of, for example, a classifier. For classification tasks, a supervised learning algorithm looks at the training data set to determine, or learn, the optimal combinations of variables that will generate a good predictive model. The goal is to produce a trained (fitted) model that generalizes well to new, unknown data.

The fitted model is evaluated using “new” examples from the held-out datasets (validation and test datasets) to estimate the model’s accuracy in classifying new data. To reduce the risk of issues such as over-fitting, the examples in the validation and test datasets should not be used to train the model. Most approaches that search through training data for empirical relationships tend to over-fit the data, meaning that they can identify and exploit apparent relationships in the training data that do not hold in general [25].

Validation Dataset A validation dataset is a dataset of examples used to tune the hyper-parameters (i.e. the architecture) of a classifier. It is sometimes also called the development set or the ”dev set”. An example of a hyperparameter for artificial neural networks includes the number of hidden units in each layer. It, as well as the testing set (as mentioned above), should follow the same probability distribution as the training dataset [25].

In order to avoid overfitting, when any classification parameter needs to be adjusted, it is necessary to have a validation dataset in addition to the training and test datasets. The validation dataset functions as a hybrid: it is training data used for testing, but neither as part of the low-level training nor as part of the final testing.

Testing Dataset A testing dataset is used for evaluating the model. Untill the model is trained, it is verified if it is giving desired results using validation dataset. Testing is the final step of judging a model performance. For testing a model, a test dataset consiting a similar architecture as that of the input and output layers of training dataset is required. This is mostly useful for classification problems.

2.4 NORMALIZING/SCALING THE DATA

When training neural networks,the examples are fed in and the expected output are compared to the true output of the network [26].Then, the gradient descent is used to update the parameters of the model in the direction which will minimize the difference between expected (or ideal) outcome and the true outcome. In other words, the goal is to minimize the error observed in the model's predictions [25].

By normalizing all of the inputs to a standard scale, the algorithm is allowing the network to more quickly learn the optimal parameters for each input node.

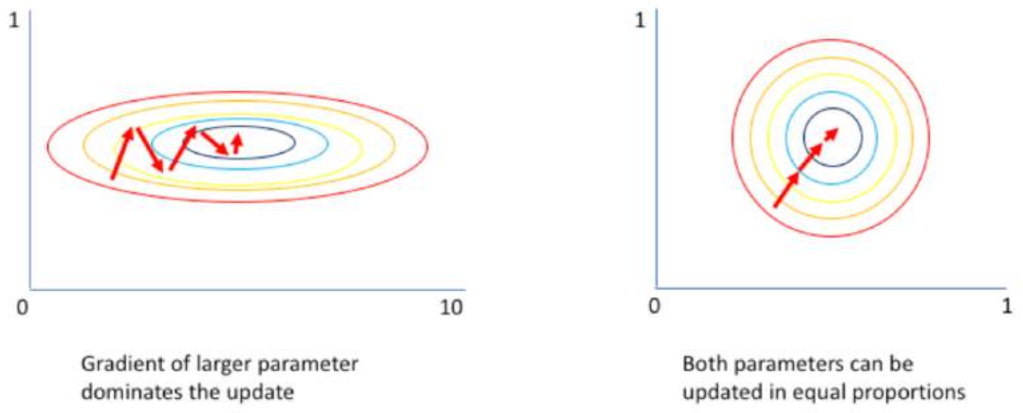

1
0
10
Gradient of larger parameter dominates the update
1
0
1
Both parameters can be updated in equal proportions

Figure 2.14: Why Normalize? by Jeremy Jordan [26]

Additionally, computers lose accuracy when performing math operations on really large or really small numbers. Moreover, if the inputs and target outputs are on a completely different scale than the typical -1 to 1 range, the default parameters for the neural network (ie. learning rates) will likely be ill-suited for the data.

Normalizing the input of the network is a well-established technique for improving the convergence properties of a network. In the following chapter *"Neural Network Model"*, the data is trained and fed into a Neural Network model and observed.

Chapter 3

NEURAL NETWORK MODEL ARCHITECTURE

3.1 BUILDING A NEURAL NETWORK: MAIN STEPS

At an abstract level, a Neural Network can be considered as a function which takes an input and produces an output, and whose behavior is parameterized by the weights and bias vector and controlled by the activation function to converge, gradually, using gradient descent algorithm.

A **unit** is the basic building of a neural network, it is a function that takes as input a vector $x \in \mathbb{R}^n$ and produces a scalar. It is parameterized by a weight vector $w \in \mathbb{R}^n$ and a bias term denoted by b. The output of the unit can be described as

$$\hat{y} = \sigma\left(w^T x + b\right), \text{ where } \sigma(z) = \frac{1}{1+e^{-z}} \tag{3.1}$$

where $f : \mathbb{R} \to \mathbb{R}$ is referred to as an activation function. Units are organized as layers, with every layer containing one or more units. The last layer is referred to as the output layer. All layers before the output layers are referred to as hidden layers [25]. The number of units in a layer is referred to as the width of the layer. The number of layers is referred to as the depth of the network. This is where the notion of deep (as in deep learning) comes from. Every layer takes as input the output produced by the previous layer, except for the first layer, which consumes the input. The output of the last layer is the output of the network and is the prediction generated based on the input. A neural network can be seen as a function$f_\theta : x \rightarrow y$,which takes as input $x \in \mathbb{R}^n$ and produces as output $y \in \mathbb{R}^m$ and whose behavior is parameterized by $\theta \in \mathbb{R}^n$.

3.1.1 MAIN STEPS

3.1.1.1 DEFINING MODEL ARCHITECTURE

Initially, define the size of input layer and the size of the output layer. Based on intuition, define the size of hidden layer after few iterations [25]. Let the x and y be the input training data and output training data repectively. In a training dataset, there are m training examples. the notation for a weight matrix of corresponding training example is $W^{[i]}$ and for a corresponding layer is $W^{[i](j)}$.

Given , $\left\{\left(x^{(1)}, y^{(1)}\right), \ldots, \left(x^{(m)}, y^{(m)}\right)\right\}$, want $\hat{y}^{(i)} \approx y^{(i)}$

$$\hat{y} = \sigma\left(w^T x + b\right), \text{ where } \sigma(z) = \frac{1}{1+e^{-z}} \tag{3.2}$$

The activation function described here is sigmoid.

3.1.1.2 RANDOM INITIALIZATION

When the neural network is implemented, it is important to initialize the weights randomly. They should not be initialized to "0". If the *weights* are intialized to 0, then all hidden units start off computing the same function. And both hidden the units have the same influence on the output unit, then after one iteration, that same statement is still true, the two hidden units are still symmetric. If *weights* are too large, it is more likely to end up at the very start of training, with very large values of "z". In return, activation functions will be saturated, thus slowing down learning [25]. This is called **Symmetry breaking problem**. And so in this case, there is really no point to having more than one hidden unit because they are all computing the same thing. Therefore, the weights and bias should be initialized randomly.

3.1.1.3 FORWARD PROPAGATION (CURRENT LOSS)

In Logistic Regression, the following **Loss Function** is used for a single training example [25]:

$$\mathscr{L}(\hat{y}, y) = -(y \log \hat{y} + (1-y)\log(1-\hat{y})) \tag{3.3}$$

If $y = 1$: the Loss function, $\mathscr{L}(\hat{y}, y) = -\log \hat{y} \longleftarrow$will push the parameter $\log \hat{y}$ large, want $\hat{y}$ large.
If $y = 0$: the Loss function, $\mathscr{L}(\hat{y}, y) = -\log(1-\hat{y}) \longleftarrow$will push the parameter $1-\hat{y}$ large, want $\hat{y}$ small.

Finally, the **Cost function**, which is the used as a reference for cost of the parameters. *The parameters: Weights and Bias (ω and b) try to minimize the overall Cost Function and loss function, over all training examples for the efficiency of the algorithm*:

$$J(\omega, b) = \frac{1}{m}\sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y^{(i)}\right) = -\frac{1}{m}\sum_{i=[}^{n}\left[y^{(1)}\log \hat{y}^{(i)} + \left(1-y^{(1)}\right)\log(1-\hat{y}(i))\right] \tag{3.4}$$

Note: The Loss function computes the error for a single training example, whereas the Cost function is the average of the Loss function of the entire training set.

$$\left.\begin{array}{c} x_1 \\ w_1 \\ w_2 \\ \mathrm{b} \end{array}\right] \longrightarrow \underbrace{z = w_1x_1 + w_2x_2 + b} \longrightarrow a = \sigma(z) \longrightarrow \mathcal{L}(\mathrm{a}, y) \tag{3.5}$$

Correct vectorized implementation of Forward Propagation for layer "l", $1 \leq l \leq$ Number of layers:

$$\begin{aligned} z^{[l]} &= w^{[l]}A^{[l-1]} + b^{[l]} \\ A^{[l]} &= g^{[l]}\left(z^{[l]}\right) \end{aligned} \tag{3.6}$$

$A^[0] = x$

3.1.1.4 BACKWARD PROPAGATION (CURRENT GRADIENT)

Correct vectorized implementation of Backward Propagation for layer "l", $1 \leq l \leq Numberoflayers$:

$$\begin{aligned} dz^{[l]} &= dA^{[l]} - Y \\ A^{[l]} &= g^{[l]}(z^{[l]}) = Y \\ d\omega^{(l)} &= \frac{1}{m} dz^{[l]} \cdot A^{[l-1]^T} \\ db^{[l]} &= \frac{1}{m}(dz^{[l]}, axis = 1, keepdims = True) \\ dz^{[l-1]} &= w^{[l]^T} dz^{[l]} * g^{[l-1]'}(z^{[l-1]}) \end{aligned} \tag{3.7}$$

The activations for each layer are computed from the output from the previous layer and is used to update weights and bias vectors in order to update the values for the next layer. If there are multiple hidden units then the mean of all the units is calculated and used to update the corresponding values. One step of backward propagation on a computation graph yields derivative of final output variable.

3.1.1.5 UPDATE PARAMETERS (GRADIENT DESCENT)

$$\begin{aligned} \omega^{[l]} &:= \omega^{[l]} - \alpha d\omega^{[l]} \\ b^{[l]} &:= b^{[l]} - \alpha db^{[l} \end{aligned} \tag{3.8}$$

The gradient descent is updated using update rule. In this way, the updated weights and bias can be supplied to the layer ahead.

3.1.2 VECTORIZING THE NEURAL NETWORK

Vectorization is a key tool for dramatically improving the performance of code running on modern CPUs. It is the process of converting an algorithm from operating on a single value at a time to operating on a set of values at one time. Modern CPUs provide direct support for vector operations where a single instruction is applied to multiple data (SIMD). Modern CPUs provide direct support for vector operations where a single instruction is applied to multiple data (SIMD). It is observed that vectorization improves the performance of the code by 300 times [25].

Note: Explicit “for” loops should be avoided when programming Neural Networks as they increase computation time. In this regard, *Numpy* and *keras* play a very crucial role.

3.1.3 ACTIVATION FUNCTION

In the forward propagation, from the input layer to the first hidden layer and then to second hidden layer, ReLU is used as activation funtion. From second hidden layer to output layer, LINEAR is used as activation function.

TANH function is almost superior to SIGMOID, whereas ReLU is almost superior to TANH [25]. The one exception is for the output layer because if y is either zero or one, then it is reasonable for $\hat{y}$ to be a number that should output between zero and one rather than between -1 and 1. So, the output layer requires linear activation function employed on previous layer.

3.2 SOFTWARE TOOLS

For modelling the Neural Network, *python* modules namely, *Tensorflow*, *keras* and *scikit learn* are used. **TensorFlow** is a Python library for fast numerical computing created and released by Google. It is a foundation library that can be used to create deep learning models directly or by using wrapper libraries that simplify the process built on top of TensorFlow. It was created and released under the Apache 2.0 open source license. The API is nominally for the Python programming language, although there is access to the underlying C++ API. Unlike other numerical libraries intended for use in Deep Learning like Theano, TensorFlow was designed for use both in research and development and in production systems. It can run on single CPU systems, GPUs as well as mobile devices and large scale distributed systems of hundreds of machines. **Keras** is a minimalist Python library for deep learning that can run on top of Theano or TensorFlow. It was developed to make developing deep learning models as fast and easy as possible for research and development. It runs on Python 2.7 or 3.5 and can seamlessly execute on GPUs and CPUs given the underlying frameworks. It is released under the permissive MIT license. Keras is implemented in this thesis because of its modularity, which represents the model architecture as a series of graphs; minimalism, which provides clear outputs with ease of readability; and extensibility, which allows adding new layers or elements in the framework allowing to test for newer possibilities.

The summary for construction of deep learning models in Keras is as follows:

- **Define your model:** Create a Functional API model and add configured layers.
- **Compile your model:** Specify loss function and optimizers and call the compile() function on the model.
- **Fit your model:** Train the model on a sample of data by calling the fit() function on the model.
- **Make predictions** Use the model to generate predictions on new data by calling functions such as evaluate() or predict() on the model.

Sequential vs Functional API A Sequential model is appropriate for a plain stack of layers where each layer has exactly one input tensor and one output tensor. The Keras functional API is a way to create models that are more flexible than the Sequential API. The functional API can handle models with non-linear topology, shared layers, and even multiple inputs or outputs. The main idea is that a deep learning model is usually a directed acyclic graph (DAG) of layers. So the functional API is a way to build graphs of layers.

In this model, there are around 48 inputs and 7 outputs, that there is a necessity for an API that can handle multiple inputs and outputs. Therefore, Functional API is used for creating the models.

In summary, known parameters for building a Neural Network:

- Input neurons
- Output neurons
- Hidden layers should be more than 1

- ReLU is used as activation function for hidden layers and LINEAR for output layer

The parameters to be considered and used for optimization, also called hyperparameters:

- Number of hidden layers
- Number of hidden units
- Loss Function
- Learning rate
- Number of epochs
- Batch size

The above mentioned parameters will be implemented in the next chapter and their inference is obtained from the model performance.

Chapter 4

GENERATING VIRTUAL TRAINING DATA

4.1 SENSOR CONCEPT

The method for measuring stress intensity factors using a piezoelectric material is presented in [3] and [9]. They explain the electrode arrangement where they are bonded directly next to crack on the specimen surface. Although the current proposition for the crack tip measurement is based on Nano strain gauge sensor, the concept of sensor arrangement and measuring their location with respect to the specimen is inspired from a PVDF based piezoelectric sensor [27]. The only difference is that in the latter a potential difference between electrodes is used for estimating strain field whereas in the former a strain gauge is used for the same. When the concept was tested, using the example of a crack in an infinite plate made of an isotropic material, it turns out that the positions of the electrodes have to be closer to the crack tip with shorter distances to each other.

Through the literature, it has been observed that in order to determine the loading situation at the crack tip based on the sensor outputs like strain form strain gauge sensors or potential difference from piezoelectric sensors, the solution of an inverse boundary value problem of the theory of elasticity is required. However, the practical application of the sensor concept is focused on the detection of unknown crack positions and SIF's. This means that the crack position with respect to the sensor arrangement is not known and the coordinates of the measuring points *(r_i, ϕ_i)* in the crack coordinate system *(x, y)* thus are not available.

Introducing a local coordinate system *(x',y')* related to the sensor arrangementin Figure 4.1, the position of any measuring point can be calculated by coordinate transformation to the crack coordinate system (x, y) according to [27]. As the coordinates of the electrodes in a PVDF sensor with respect to the film are known, the fracture quantities are complemented by three more unknowns (x_0, y_0, β) describing the film position with respect to the crack faces. This concept in the applied in the current thesis, with an exemption of a film. The sensors position is referred using a global coordinate system of that of the crack. The sensors are arranged in different layouts which are discussed below.

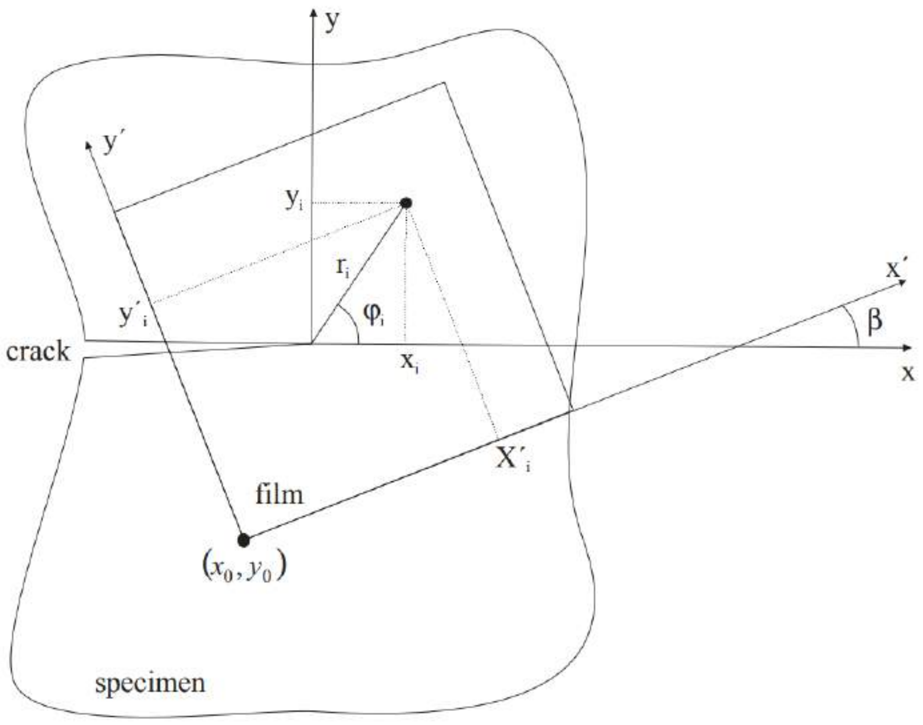


Figure 4.1: Crack coordinate system *(x,y)* and film coordinate system *(x',y')* [2]

In order to obtain the stress and strain fields from the crack location, the local coordinates of the sensors with varying origin with respect to crack are obtained from the Figure 4.1 in accordance with [2].The (local) coordinate system is considered with its origin at the left bottom end of the sensor arrangement and the corresponding coordinates of the sensors is evaluated (x',y'). The following equations give a relation between two, local and global, coordinate systems:

$$x_i = x'_i \cos\beta - y'_i \sin\beta + x_0 \tag{4.1}$$

$$y_i = x_i' \sin\beta + y_i' \cos\beta + y_0 \tag{4.2}$$

$$r_i = \sqrt{x_i^2 + y_i^2} \tag{4.3}$$

$$\begin{aligned} \varphi_i &= \arccos \tfrac{x_i}{r_i} 0 \le \varphi_i \le \pi \\ \varphi_i &= -\arccos \tfrac{x_i}{r_i} - \pi \le \varphi_i < 0 \end{aligned} \tag{4.4}$$

4.2 GENERATING VIRTUAL TRAINING DATA

The sensor concept is verified in [2] at a plate specimen holding the dimensions width=10mm, height= 250mm, length=750mm featuring an edge crack of length 25 mm, which is loaded in different ways. In [2] the film has the dimensions width=0.1mm, height= 50mm, length=50mm exhibiting 49 sensors with an area $0.5 \times 0.5mm$ each, being arranged at regular distances of 5 mm to each other. In this thesis, a new approach with a similar arrangement of sensors without a film is implemented on a 4×4 regular rectangular arrangement of sensors (16 sensors) with a distance of 10mm between sensors as observed in Figure 4.2.

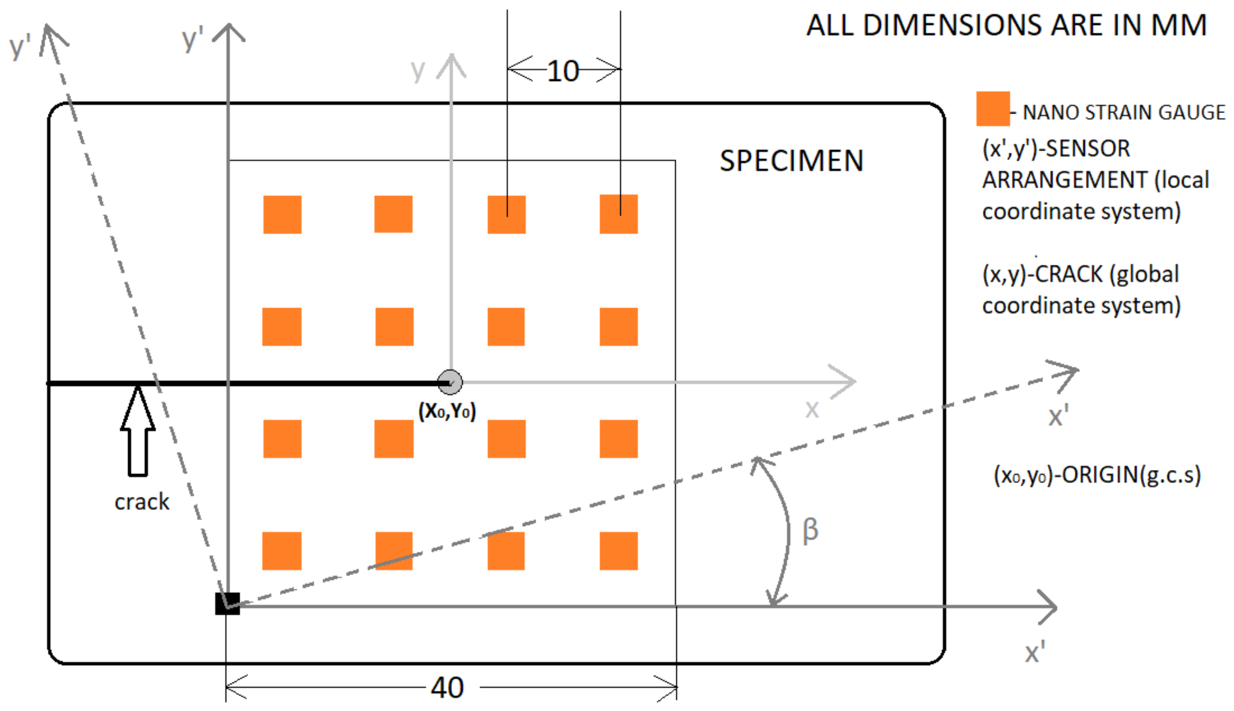


Figure 4.2: Rectangular 4x4 Sensor Arrangement [2]

The local coordinate system for 4x4 rectangular layout is provide in Table 4.1 The practical application of the sensor concept is aimed for detection of unknown crack paths. This

Table 4.1: 4x4 Rectangular Sensor Arrangement (Local Coordinate System)

(X', Y') [mm]	1	2	3	4
1	(5,5)	(15,5)	(25,5)	(35,5)
2	(5,15)	(15,15)	(25,15)	(35,15)
3	(5,25)	(15,25)	(25,25)	(35,25)
4	(5,35)	(15,35)	(25,35)	(35,35)

Table 4.2: Range for SIF-I, SIF-II, T, Crack angle and location

Parameter	Training Range
K_I[MPa$\sqrt{mm}$]	0 to 1200
K_{II}[MPa$\sqrt{mm}$]	-110 to 130
T[MPa]	-60 to 22
Beta[rad]	0 to 2 π
Y[mm]	-35 to -6
X[mm]	-35 to -6

means that the crack position with respect to the sensor is calculated for different positions of sensor layout and the corresponding loading fields are estimated and then the model is trained to solve the inverse problem. Before calculating the stress and strain fields, the range for parameters like "Stress Intensity Factors K_I, K_{II} and T-stress" are defined based on the lowest and highest values used in several research papers, along with Young's modulus and Poisson's ratio taken from [2].

In Linear Elastic Fracture Mechanics (LEFM), the asymptotic fields (r→0) of displacements, stresses and strains near the crack tip are expressed in terms of stress intensity factors (K-factors) [28] and [29]. At finite distance to the crack tip higher order terms are taken into account, e.g. a non-singular constant stress named T-stress acting parallel to the crack. The stress tensor components are:

$$\begin{aligned}
\sigma_{11} =& \frac{K_I}{\sqrt{2\pi r}} \cos\left(\frac{\varphi}{2}\right)\left(1-\sin\left(\frac{\varphi}{2}\right)\sin\left(\frac{3\varphi}{2}\right)\right) - \\
& \frac{K_{II}}{\sqrt{2\pi r}} \sin\left(\frac{\varphi}{2}\right)\left(2+\cos\left(\frac{\varphi}{2}\right)\cos\left(\frac{3\varphi}{2}\right)\right) + T \\
\sigma_{22} =& \frac{K_I}{\sqrt{2\pi r}} \cos\left(\frac{\varphi}{2}\right)\left(1+\sin\left(\frac{\varphi}{2}\right)\sin\left(\frac{3\varphi}{2}\right)\right) + \\
& \frac{K_{II}}{\sqrt{2\pi r}} \sin\left(\frac{\varphi}{2}\right)\cos\left(\frac{\varphi}{2}\right)\cos\left(\frac{3\varphi}{2}\right) \\
\sigma_{12} =& \frac{K_I}{\sqrt{2\pi r}} \sin\left(\frac{\varphi}{2}\right)\cos\left(\frac{\varphi}{2}\right)\cos\left(\frac{3\varphi}{2}\right) + \\
& \frac{K_{II}}{\sqrt{2\pi r}} \cos\left(\frac{\varphi}{2}\right)\left(1-\sin\left(\frac{\varphi}{2}\right)\sin\left(\frac{3\varphi}{2}\right)\right)'
\end{aligned} \tag{4.5}$$

with -$\pi \leq \pi$. The K-I and K-II represent the two in-plane crack opening modes I and II. The polar coordinate system (r, ϕ) has its origin at the crack tip and on the ligament $\phi = 0$.

Initially, a PVDF film was used in [2] and [27] which required a transformation of strain values from the specimen to the film using below equations. However, in the current thesis, estimations through nanosensors does not use any film between specimen and the sensors. Therefore, the strain field could be calculated directly using below equations

$$\begin{aligned}
\varepsilon_{11} &= (\sigma_{11} - (\nu * \sigma_{22}))/E, \\
\varepsilon_{22} &= (\sigma_{22} - (\nu * \sigma_{11}))/E, \\
\varepsilon_{12} &= (1+\nu) * \sigma_{12}/E
\end{aligned} \tag{4.6}$$

The program is looped over $K_I, K_{II}, T, \cos\beta, \sin\beta, Y_0, X_0$ for different values mentioned in Table 4.2. The data is generated for number of points within the range like 4,5,...,9 points in each parameter. Larger data is computed to compare the model performance. All the corresponding values within the range of each parameter is estimated and stored in individual arrays as columns. In this 4×4 rectangular layout, the sensors are arranged around the crack tip like a square in 4 rows with 4 sensors in each row. This design is inspired from [2]. With a total of 16 sensors, there would be 3 strain values for each sensor with a total of 48 columns or neurons as the input data. The corresponding $K_I, K_{II}, T, \cos\beta, \sin\beta, Y_0, X_0$ will be the output data, 7 values as output for each loop.

Stacking the training data The column arrays of data are gathered together or stacked to form the input and output values of training data and validation/testing data. The strain field from all sensors is considered as the input data and K_I, K_{II}, T, $\cos\beta$, $\sin\beta$ and crack positions through coordinates are considered as output data and stored. All the generated data is used instead of experimental data to train the model, hence, the name virtual training data.

4.3 OTHER LAYOUTS OF SENSORS

The virtual training data is generated in the above mentioned procedure for layouts of sensor arrangement.

4.3.1 RECTANGULAR (8x4) SENSOR LAYOUT

In this layout, the sensors are arranged around the crack tip like a rectangle in 4 rows with 8 sensors in each row. The sensor layout can rotate in any angle β. With a total of 32 sensors, there would be 3 strain values for each sensor with a total of 96 columns or neurons as the input data. The corresponding $K_I, K_{II}, T, \cos\beta, \sin\beta, Y_0, X_0$ will be output data, 7 values as output for each example. In this arrangement, maximum advantage is obtained if the sensors are placed in such a way around the crack tip that x-axis of sensor arrangement is parallel to the crack direction. Otherwise, it is similar to a regular rectangular arrangement. The arrangement of the sensors can be observed in the figure:4.3: The local coordinate system for 8x4 this layout is provided in the table:4.3.

Table 4.3: 8x4 Rectangular Sensor Arrangement (Local Coordinate System)

(X', Y') [mm]	1	2	3	4	5	6	7	8
1	(5,5)	(15,5)	(25,5)	(35,5)	(45,5)	(55,5)	(65,5)	(75,5)
2	(5,15)	(15,15)	(25,15)	(35,15)	(45,15)	(55,15)	(65,15)	(75,5)
3	(5,25)	(15,25)	(25,25)	(35,25)	(45,25)	(55,25)	(65,25)	(75,25)
4	(5,35)	(15,35)	(25,35)	(35,35)	(45,35)	(55,35)	(65,35)	(75,35)

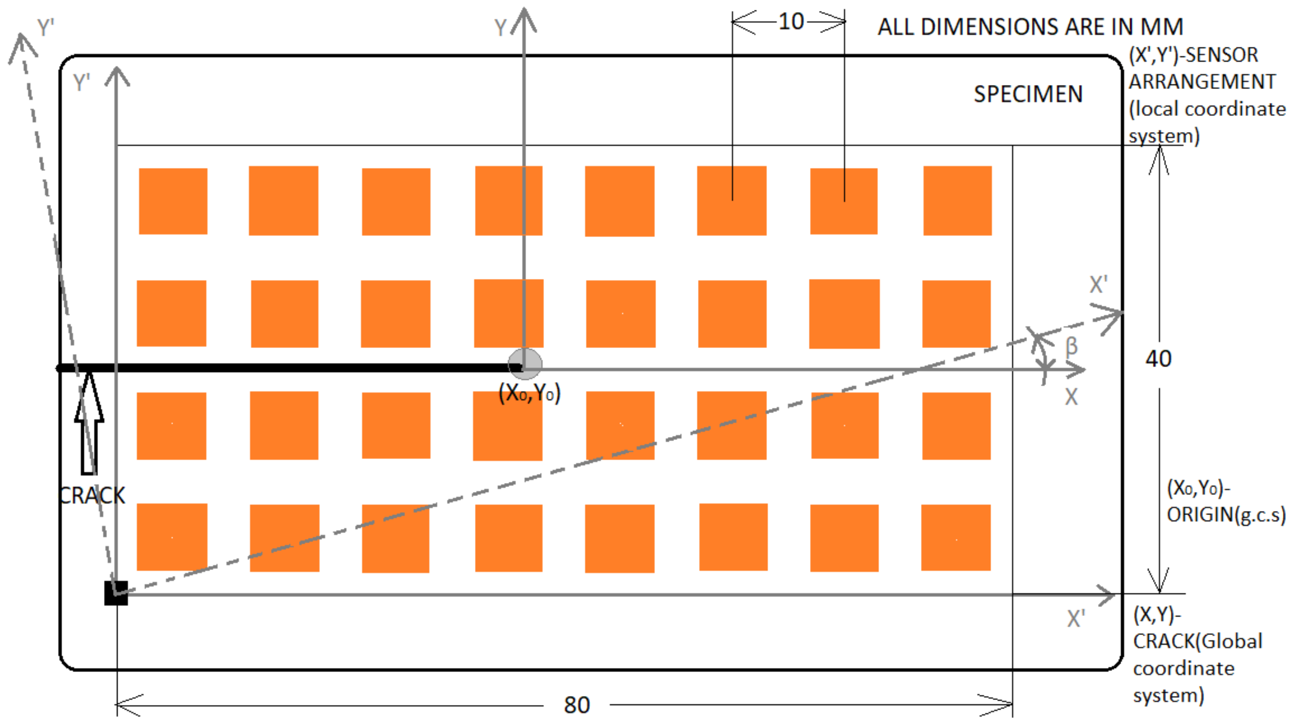

ALL DIMENSIONS ARE IN MM
10
Y'
Y'
Y
SPECIMEN
(X',Y')-SENSOR ARRANGEMENT (local coordinate system)
X'
β
X
40
(X0,Y0)
CRACK
(X0,Y0)-ORIGIN(g.c.s)
X'
(X,Y)-CRACK(Global coordinate system)
80

Figure 4.3: Rectangular 8x4 Sensor Arrangement

4.3.2 HEXAGONAL SENSOR LAYOUT

In this layout, the sensors are arranged around the crack tip like a equilateral hexagon. Each vertex of the hexagon holds a sensor. The hexagon consists of 6 equilateral triangles.The centroid of each equilateral triangle of the hexagon holds another sensor. The centre of the Hexagon holds another sensor forming a robust distribution of the sensor arrangement. The sensor layout can rotate in any angle β. With a total of 13 sensors, there would be 3 strain values for each sensor with a total of 39 columns or neurons as the input data. The corresponding $K_I, K_{II}, T, \cos\beta, \sin\beta, Y_0, X_0$ will be output data, 7 values as output for each example. The arrangement of the sensors can be observed in the Figure 4.4: The local coordinate system for the sensors in hexagon are based on "a" distance between sensors, which is 20mm, and angle α. First layer of sensors are positioned on the vertices of the hexagon and the second layers are positioned at an α=$\pi/6$ to α=11x$\pi/6$ in six positions at $a/\sqrt{3}$ from the centre.

The virtual training data is generated for different layouts of sensor arrangement.

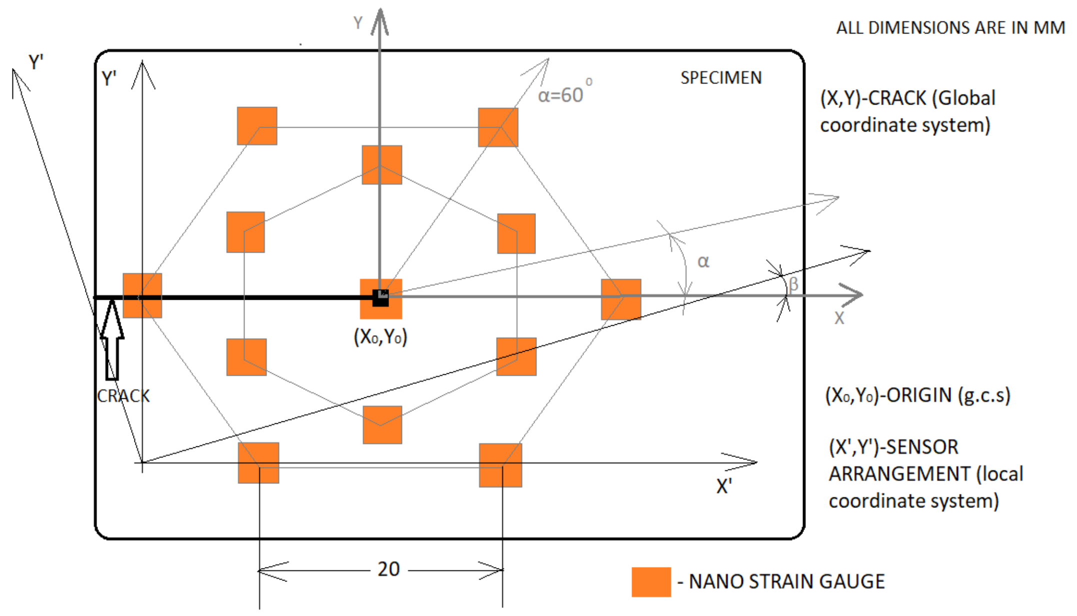


Figure 4.4: Hexagon Sensor Arrangement

Chapter 5

MODEL: STRATEGY AND INFERENCE

In summary, as explained in the previous chapters, the initial steps for building a reliable Neural Network are configuring a training data, modeling the neural architecture (input and output node strategy) and type of analysis(regression, classification). These steps have been implemented successfully.

In this chapter, initially, the Neural Network model for a rectangular (4×4) sensor layout is discussed with focus on its performance, reliability, optimization, evaluation, sampling strategy of the training data and limitations.

In the next chapter, the comparison of different layouts of sensor arrays, namely,

- Rectangular (4×4) Sensor Layout.
- Rectangular (8×4) Sensor Layout.
- Hexagonal Sensor Layout.

is performed.

In order to finalise a particular strategy, a performance measure using "loss vs. epochs" curves are plotted. It is observed in the applications that it is ideal to plot loss across epochs rather than iteration. During an epoch, the loss function is calculated across every data items and it is guaranteed to give the quantitative loss measure at the given epoch. But plotting curve across iterations only gives the loss on a subset of the entire dataset. Therefore, to estimate the performance of a model, from herein the loss vs epoch graphs are plotted.

Another most used curves to understand the progress of Neural Networks is an Accuracy curve. The gap between training and validation accuracy is a clear indication of overfitting. The larger the gap, the higher the overfitting.

5.1 RECTANGULAR (4x4) SENSOR LAYOUT

Given that,
Input Layer: Number of sensors $\times 3 \rightarrow (\epsilon_{11}, \epsilon_{12}, \epsilon_{22})$
Output Layer: $x_0, y_0, K_I, K_{II}, T, \cos\beta, \sin\beta$

In the current 4×4 regular rectangular model, there are 16 sensors and a strain field for each sensor, therefore, $16 \times 3 = 48$ Neurons as the Input Layer and the 7 parameters mentioned above are given as Output layer neurons.

Artificial Neural Networks have two main hyperparameters that control the architecture or topology of the network: the number of hidden layers and the number of nodes in each hidden layer. The most reliable way to configure these hyperparameters for a specific predictive modeling problem is via systematic experimentation with a robust test harness because this is the first attempt to address such a specific inverse problem which has not been configured or solved before. As in any case of solving a stochastic gradient descent problem, there must a suitable optimizer, here, being a regression problem, "adam" is considered, as it is suggested and widely accepted by experts. Adam has a default learning rate (lr) of 0.01.

5.1.1 CHOOSING NUMBER OF HIDDEN LAYERS

A "deep Neural Network" has more than two hidden layers. In this case, a study is performed between 2,3, 4 and 5 hidden layers with standard number of nodes (700) in each hidden layer to identify which architecture performs better : less computation time and better performance.

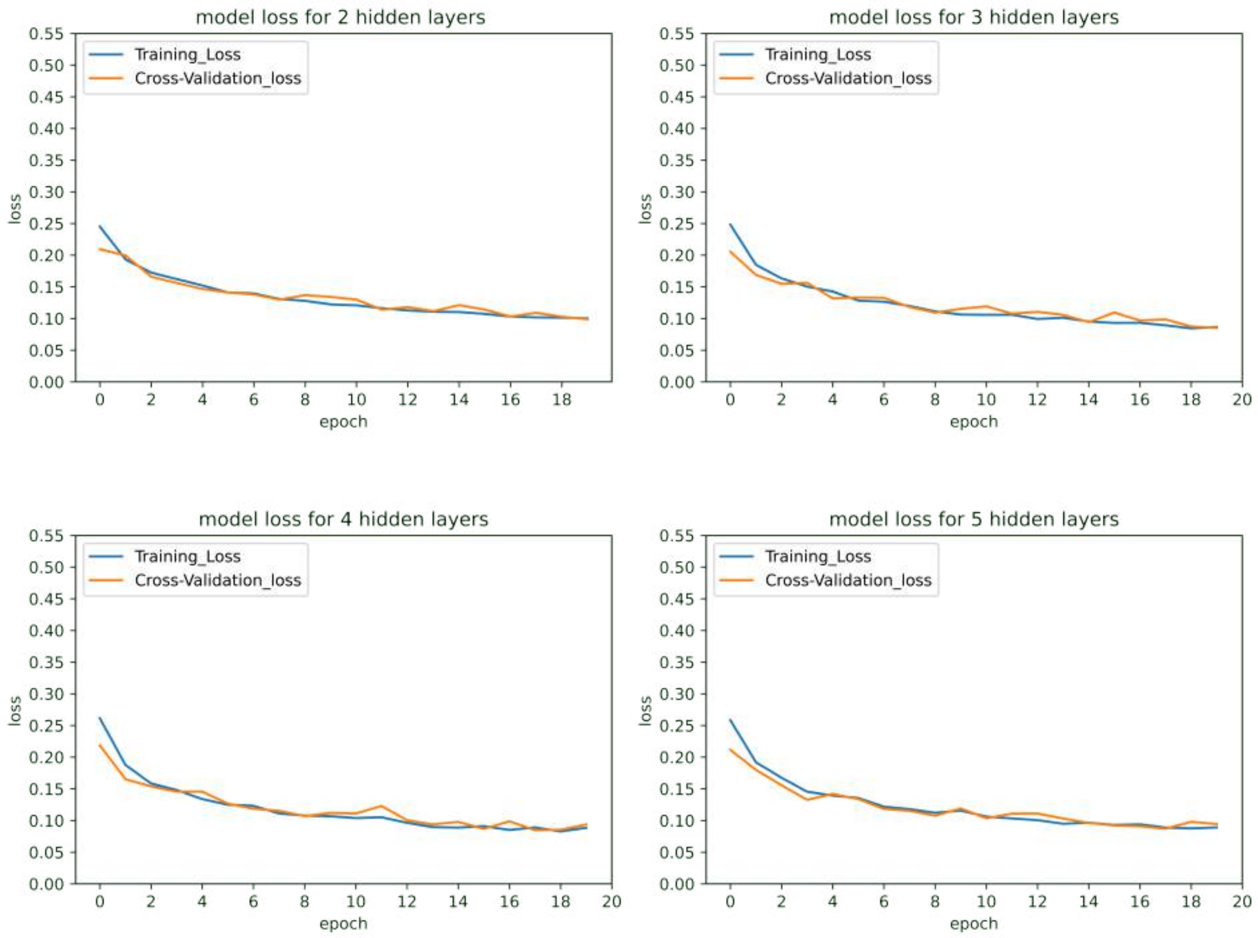


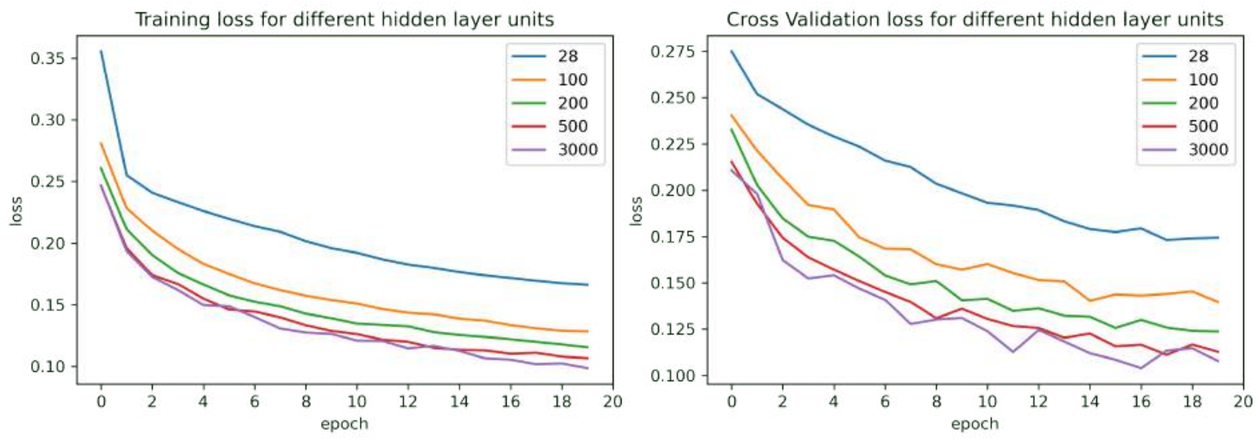
Figure 5.1: Model performance with different hidden layers

Inference from Figure 5.1: The hidden layers 3, 4, 5 perform similarly with a tiny improvement compared to 2 hidden layers. More hidden layers increases the computation time and is only reasonable if the loss decreases consistently. The model performance remains steady from 2 hidden layers till 5 hidden layers with the loss reaching 0.10 approximately, at the end of 20 epochs without any significant improvement over 2 hidden layers. Therefore,

$$\boxed{\text{Number of Hidden layers} = 2}$$

5.1.2 CHOOSING NUMBER OF UNITS FOR EACH HIDDEN LAYER

There are some empirically derived rules of thumb for choosing the number of hidden units. The commonly followed rule is that the optimal size of the hidden layer is usually between the size of the input and size of the output layers as suggested by [30]. In general, for many approximation problems, a decent performance is obtained (even without a second optimization step) by setting the hidden layer configuration using two rules, namely, number of hidden layers equals one and the number of neurons in that layer is the mean of the neurons in the input and output layers. In this case, the mean is close 27.5 rounding off to 28. But, it is observed in practice that each training set requires a different optimized architecture to extract best performance. In this regard, systematic experimentation have been performed with hidden layer units for each hidden layer ranging from 28 to 3000 units.



(a) Training Loss for different hidden units **(b)** Validation Loss for different hidden units

Figure 5.2: Model Loss for different hidden units

Inference from Figure 5.2: It has been observed that when hidden units are 500 for each hidden layer, the model gives better performance than that of 28. And when hidden layers are from 1000 to 3000, the performance remains similar to 500 hidden units but with longer computation time. Therefore, hidden units from 500 to 700 for each hidden layer perform efficiently.

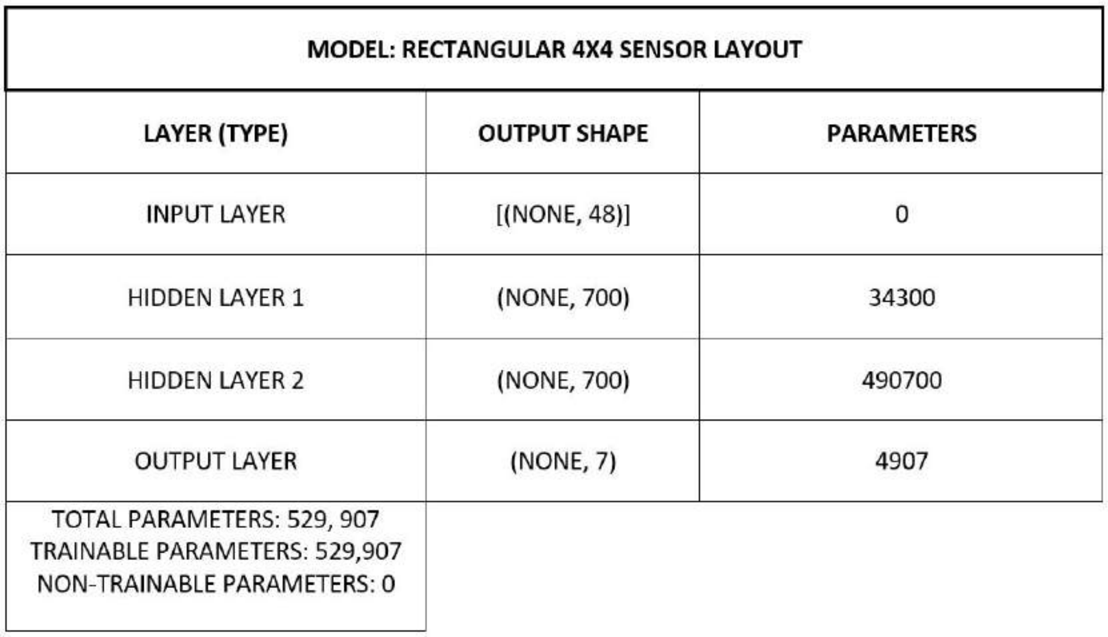
$$\boxed{\text{Number of units for each Hidden layer} = 700}$$

5.1.3 CHOOSING THE TRAINING DATASET

The performance of a Neural Network is influenced by the correct amount of training data considered. If the amount of training data is less, then there is a general possibility of poor approximation. An over-constrained model will underfit the small training dataset, whereas an under-constrained model, in turn, will likely overfit the training data, both resulting in poor performance. Too little test data will result in an optimistic and high variance estimation of model performance.

After finalizing the input layer, output layer, hidden layers and the corresponding units, the Model is built for an initial dataset consisting of (3072,48) and (3072,7) as training data size and (1024,48) and (1024,7) as validation data size referred as "4p". The performance is measured using "Mean Absolute Error" as loss function and the optimizer is *adam* derived from "adaptive moment estimation". Adam is an optimization algorithm that can be used instead of the classical stochastic gradient descent procedure to update network weights iterative based in training data.

$$\boxed{\text{Loss Function = Mean Absolute Error; Optimizer = adam}}$$



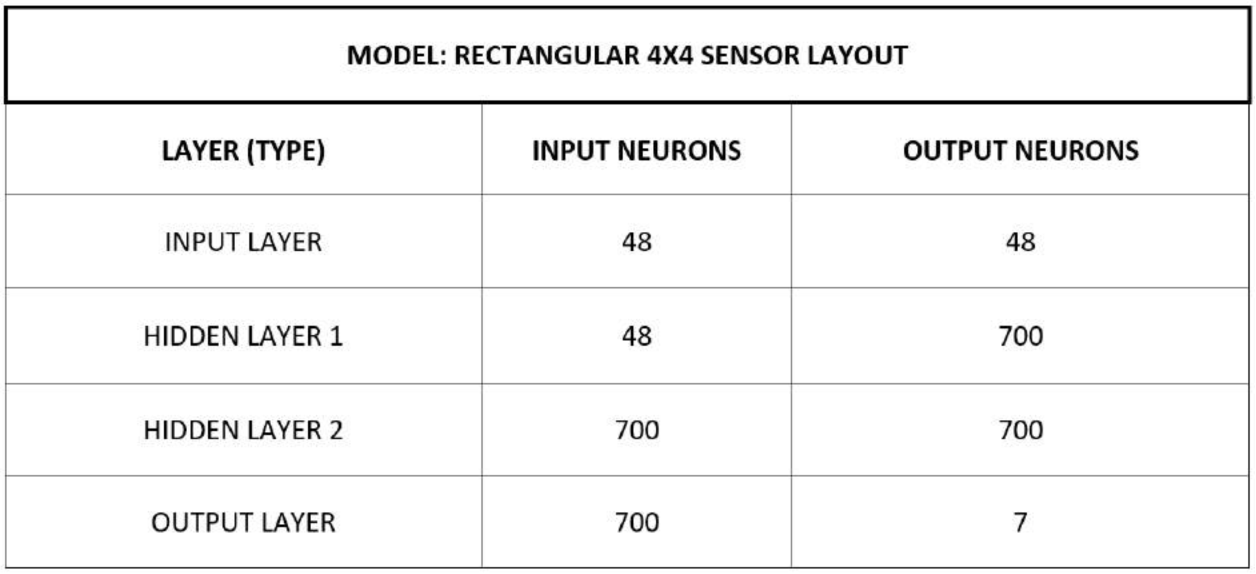
MODEL: RECTANGULAR 4X4 SENSOR LAYOUT

LAYER (TYPE)	OUTPUT SHAPE	PARAMETERS
INPUT LAYER	[(NONE, 48)]	0
HIDDEN LAYER 1	(NONE, 700)	34300
HIDDEN LAYER 2	(NONE, 700)	490700
OUTPUT LAYER	(NONE, 7)	4907

TOTAL PARAMETERS: 529, 907
TRAINABLE PARAMETERS: 529,907
NON-TRAINABLE PARAMETERS: 0

Figure 5.3: Model Summary

As observed in Figure 5.3, the string summary of the Neural Network with shapes of input, output and hidden layers is observed. This is obtained using "Functional API" in *keras* module. The functional API can handle models with non-linear topology, shared layers, and even multiple inputs or outputs. The main idea is that a deep learning model is usually a directed acyclic graph (DAG) of layers. So the functional API is a way to build graphs of layers. From the Figure 5.4, the arrangement of different layers with multiple inputs and



MODEL: RECTANGULAR 4X4 SENSOR LAYOUT		
LAYER (TYPE)	**INPUT NEURONS**	**OUTPUT NEURONS**
INPUT LAYER	48	48
HIDDEN LAYER 1	48	700
HIDDEN LAYER 2	700	700
OUTPUT LAYER	700	7

Figure 5.4: Model Architecture

outputs can be observed. "Dense" implements the operation: output = activation(dot(input, kernel) + bias) where activation is the element-wise activation function passed as the activation argument. In this case, "ReLU" activation function is used for both the hidden layer implementations and "linear" is used for output layer calculations. Kernel is a weights matrix created by the layer, and bias is a bias vector created by the layer. These are all attributes of "Dense". From the plot in Figure 5.5, it is observed that the model could probably be trained a little more as the trend for learning on both datasets is still rising for the last few epochs. Also, the model has not yet over-learned the training dataset, showing comparable skill on both datasets. In this regard, few larger datasets are generated for training the Neural Network and a comparison of the performance of the model through these data sets is observed. Here, Implementing with an appropriate loss function will result in better approximation avoiding an under or over-fitting or a fitting which can be tuned later for better predictions.

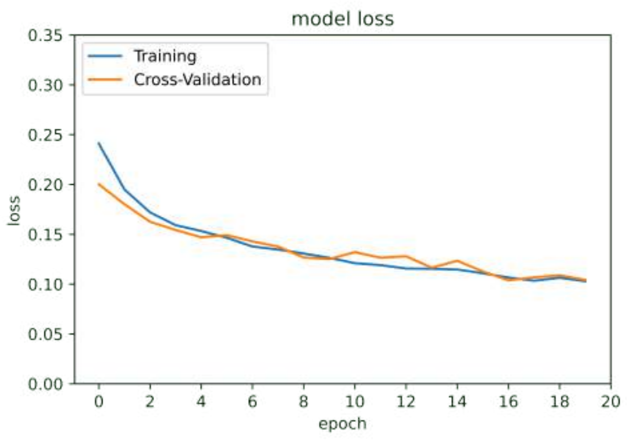


Figure 5.5: Model Loss for 4p

Table 5.1: Training and validation examples

Dataset	Training data	Validation data
4p	3072	1024
5p	11718	3907
6p	34992	11664
7p	88236	29413
8p	196608	65536
9p	398580	132861

5.1.3.1 CHOOSING THE LOSS FUNCTION

As discussed in Chapter 2, in Figure 2.13, the size of the generated training data plays a very crucial role in Model performance. This theory is validated with the help of Training data ranging from [4000 × (Input units(48) + output units(7))] data points to [500000 × (Input units(48) + output units(7))] datapoints, approximately from 220,000 (identified as "4p") datapoints to 27,500,000 (identified as "9p") datapoints respectively. The Neural Network model is built for these data sets 4p,5p, 6p, 7p, 8p and 9p with their training and validation examples as described in Table 5.1, their performance is compared. The validation dataset is 33.33% of Training dataset. Number of neurons for input layer is 48 and that of output layer is 7 which remains constant throughout the model.

MAE and MSE MAE is derived from "Mean Absolute Error", MSE is derived from "Mean Squared Error". The MSE, MAE, RMSE, and R-Squared metrics are mainly used to evaluate the prediction error rates and model performance in regression analysis. Among them, MAE and MSE are most commonly preferred as Loss Functions. In the below equations, y is origina value and $\hat{y}$ is predicted value of y.

- **MAE** (Mean absolute error) represents the difference between the original and predicted values extracted by averaged the absolute difference over the data set.

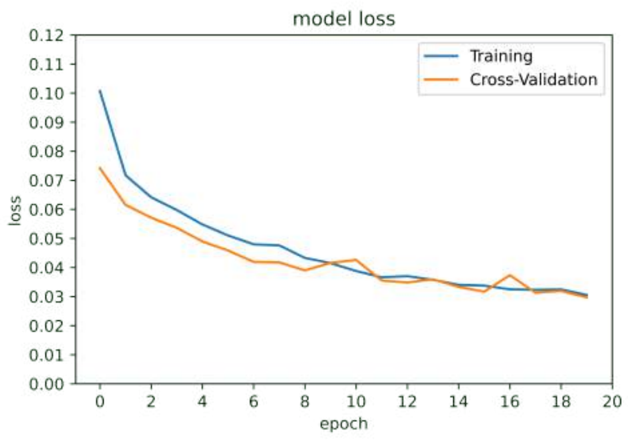
$$MAE = \frac{1}{N} \sum_{i=1}^{N} |y_i - \hat{y}| \tag{5.1}$$

- **MSE** (Mean Squared Error) represents the difference between the original and predicted values extracted by squared the average difference over the data set.

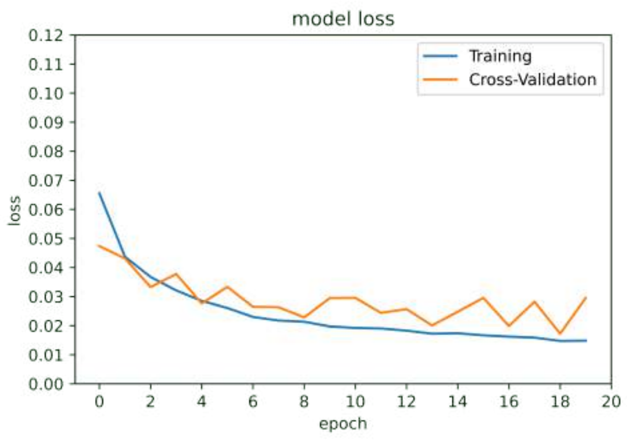
$$MSE = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y})^2 \tag{5.2}$$

Unlike classification, model accuracy cannot be used to evaluate the predictions made by a regression model. Instead, error metrics are specifically designed for evaluating predictions made on regression problems. Predictive modeling can be described as the mathematical problem of approximating a mapping function (f) from input variables (X) to output variables (y). This is called the problem of function approximation.

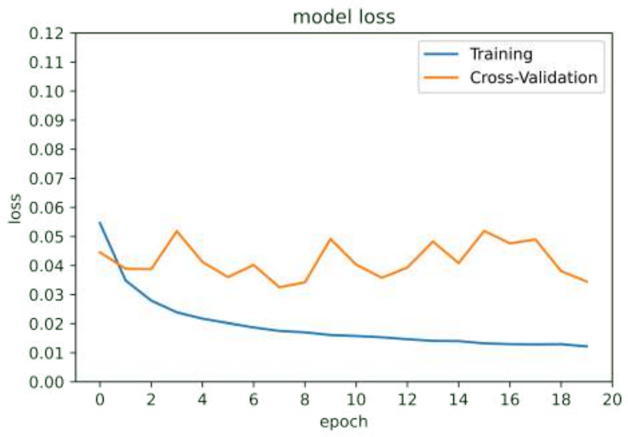
The job of the modeling algorithm is to find the best mapping function for given time and resources. MSE is also an important loss function for algorithms fit or optimized using the least squares framing of a regression problem. Here “least squares” refers to minimizing the mean squared error between predictions and expected values.



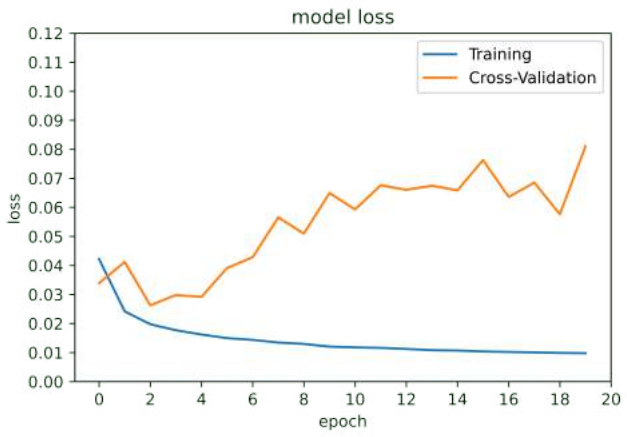
(a) Training size: 4p



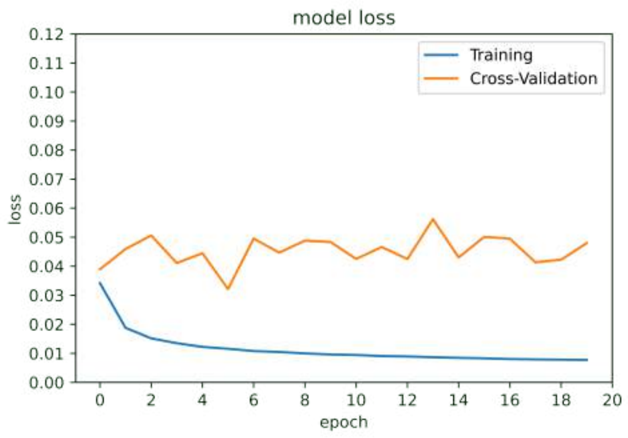
(b) Training size: 5p



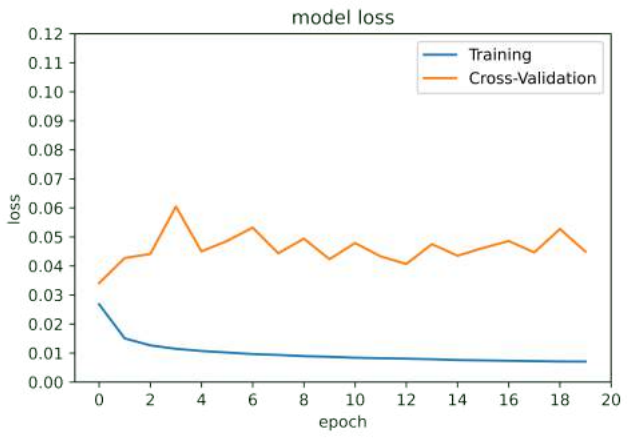
(c) Training size: 6p



(d) Training size: 7p



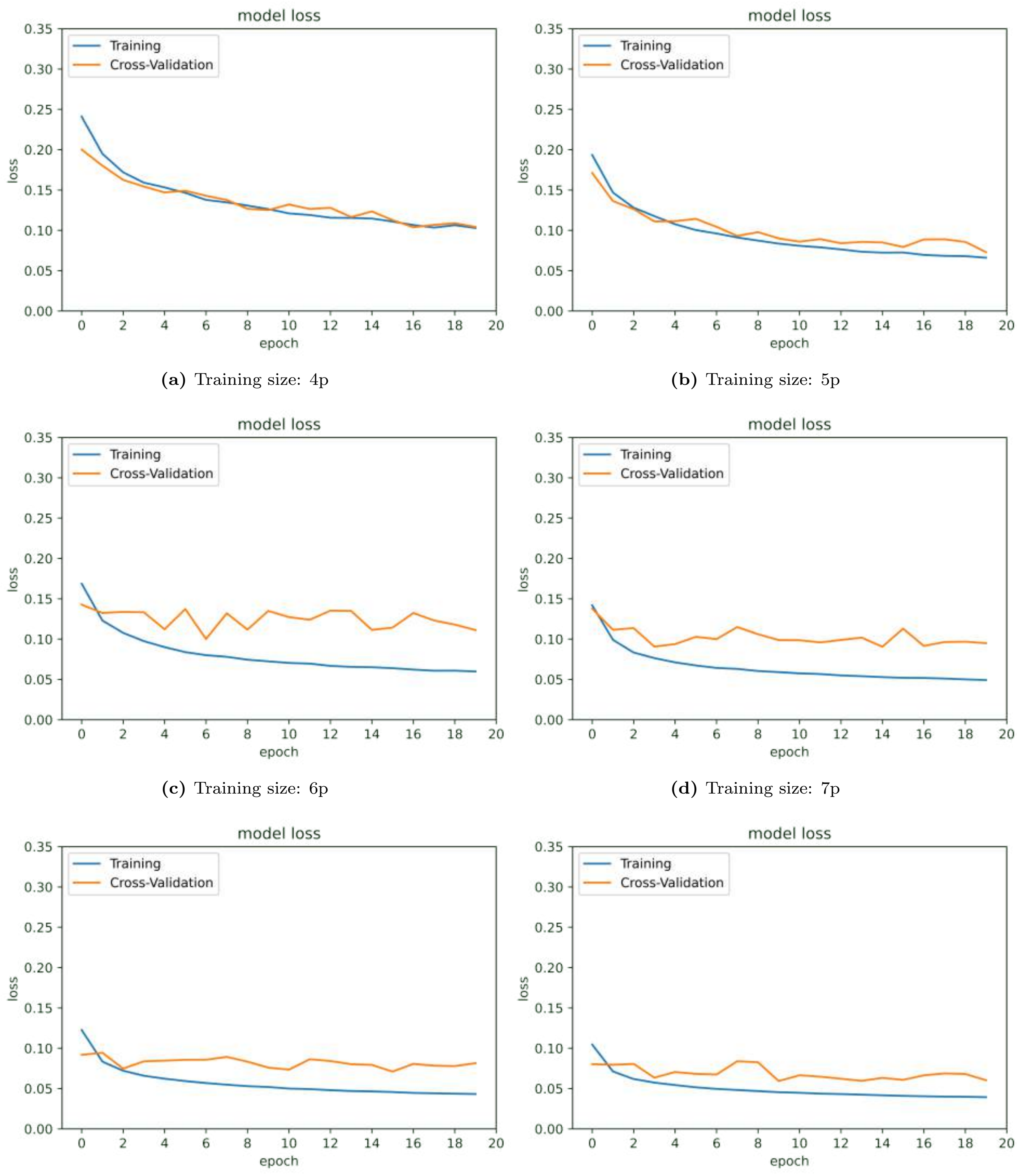
(e) Training size: 8p



(f) Training size: 9p

Figure 5.6: Comparison of Model Loss using MSE

Inference from Figure 5.6 is that the squaring also has the effect of inflating or magnifying large errors. That is, the larger the difference between the predicted and expected values, the larger the resulting squared positive error. This has the effect of "punishing" models more for larger errors when MSE is used as a loss function. It also has the effect of "punishing" models by inflating the average error score when used as a metric.



(a) Training size: 4p

(b) Training size: 5p

(c) Training size: 6p

(d) Training size: 7p

(e) Training size: 8p

(f) Training size: 9p

Figure 5.7: Comparison of Model Loss using MAE

Inference from Figure 5.6 and 5.7 is that the Mean Absolute Error, or MAE, is a popular metric because, like MSE, the units of the error score match the units of the target value that is being predicted.That is, MSE punish larger errors more than smaller errors, inflating or magnifying the mean error score. This is due to the square of the error value. The MAE does not give more or less weight to different types of errors and instead the scores increase linearly with increases in error. The mean absolute error between the expected and predicted values can be calculated using the “mean absolute error” function from the “keras” library, if the value is closer to 0, better the performance.

In conclusion, the performance of the model using MAE is more reliable than that of MSE. With increase in training size, the performance improves using MAE rather than MSE as in the former the validation loss tries to to mimic the training loss as it is supposed to do. Whereas, the validation loss in MSE, moves far away from training loss instead of converging closer.

$$\boxed{\text{Loss Function} = \text{MAE}}$$

For an appropriate training set, from Figure 5.7 it can be inferred that all datasets perform well as the loss gradually reduces towards 0. In this regard, the largest dataset, 9p performs very well compared to others as both loses start close to 0.1 and decrease reliably.

$$\boxed{\text{Training Dataset} = \text{5p and 9p}}$$

5.1.4 OPTIMIZATION OF NEURAL NETWORK

In the context of Neural Networks, Optimization refers to“non-convex optimization”. Non-convex optimization involves a function which has multiple optima, only one of which is the global optima. Depending on the loss surface, it can be very difficult to locate the global optima. Finding the global minimum on the loss surface — this is the aim of neural network training. Challenges in optimizing a Neural Network:

- Reasonable Learning Rate: If it is too small, it takes too long to converge and if too large, it may never converge.

- Learning rate scheduler
- Number of epochs
- Batch size
- Momentum

5.1.4.1 TUNING LEARNING RATE

The learning rate is a hyperparameter that controls how much to change the model in response to the estimated error each time the model weights are updated. Choosing the learning rate is challenging as a value too small may result in a long training process that could get stuck, whereas a value too large may result in learning a sub-optimal set of weights too fast or an unstable training process.

Two popular methods to tune learning rate are as follows:

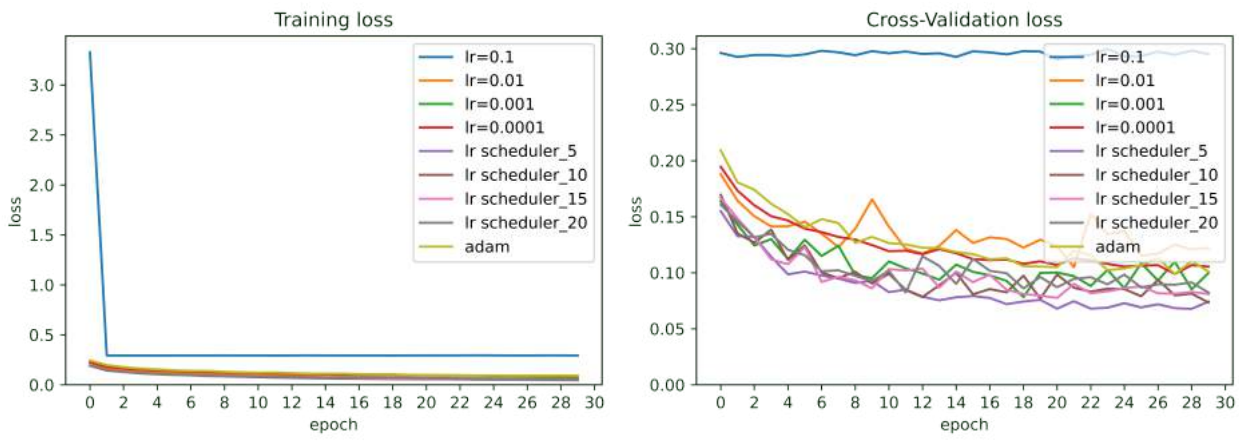
- Decrease the learning rate gradually based on the epoch.
- Decrease the learning rate using punctuated large drops at specific epochs which is called “learning rate scheduler”.

In this regard, for this Neural Network Model, **Hyperparameter Tuning** is performed by changing the “learning rate” in steps and observing the behaviour of the model. The learning rate is abbreviated as “lr” and is gradually varied throughout the model in the following sequence:

- lr=0.1
- lr=0.01
- lr=0.001
- lr=0.0001

- lr=learning rate scheduler for 5 epochs
- lr=learning rate scheduler for 10 epochs
- lr=learning rate scheduler for 15 epochs
- lr=learning rate scheduler for 20 epochs
- adam

and its performance is observed. Learning rate scheduler means to systematically drop the learning rate at specific times during training. Often this method is implemented by dropping the learning rate by half every fixed number of epochs. For example, in the case of lr scheduler_5, the model has an initial learning rate of 0.1 and drops it by 0.1 every 5 epochs. The first 5 epochs of training would use a value of 0.1, in the next 5 epochs a learning rate of 0.01 would be used, and so on. From Figure 5.8, it is inferred that the learning rate 0.1,

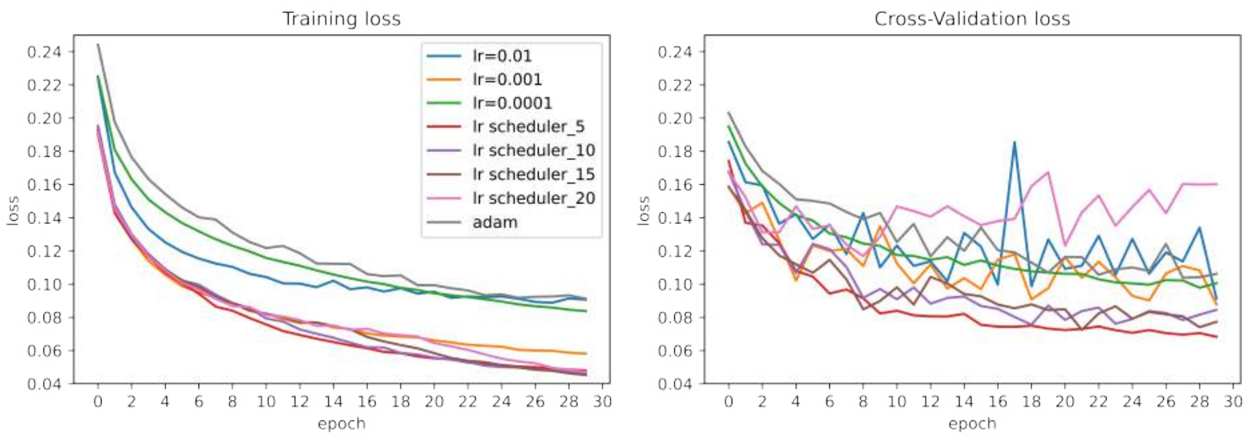


(a) Training Loss for different Learning Rates

(b) Validation Loss for different Learning Rates

Figure 5.8: Model Loss for different Learning Rates

does not improve the performance of the model, rather remains stagnant through most of the epochs in both training and validation datasets. However, other learning rates perform relatively close to each other and suggests better contribution to model performance. From Figure 5.9, a clear understanding of the analysis is obtained. Adapting the learning rate for your stochastic gradient descent optimization procedure can increase performance and reduce training time. Adam utilizes this feature and combines the best properties of the



(a) Training Loss for different Learning Rates **(b)** Validation Loss for different Learning Rates

Figure 5.9: Model Performance with different Learning Rates

AdaGrad and RMSProp algorithms to provide an optimization algorithm that can handle sparse gradients on noisy problems. It's learning rate is maintained for each network weight (parameter) and separately adapted as learning unfolds.

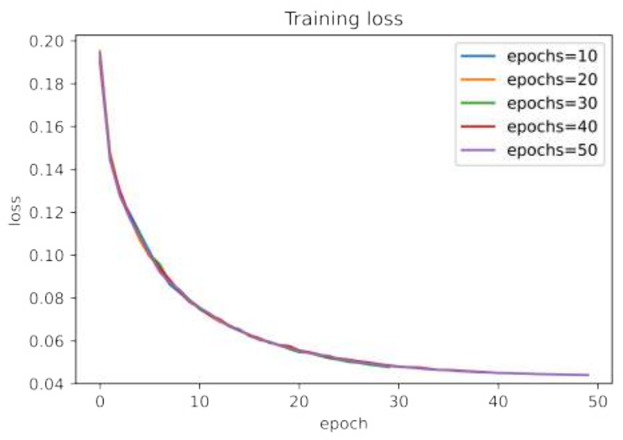
It is observed that lr=0.01, 0.0001 and lr scheduler_15, perform a similar approximation although lr=0.01 drops slower than other. Remaining lr's mimic each other and perform in close approximations in both datasets. However, lr scheduler_5 performs reliably and performs better than "adam" which is used in the initial model.

$$\boxed{\text{Learning rate= lr scheduler\_5}}$$

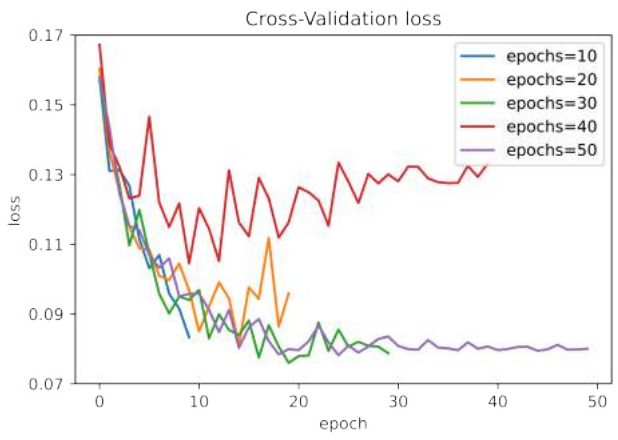
5.1.4.2 TUNING NUMBER OF EPOCHS

Like in the previous section, here, number of epochs are varied and corresponding model behaviour is recorded. The number of epochs is a hyperparameter that defines the number times that the learning algorithm will work through the entire training dataset. One epoch means that each sample in the training dataset has had an opportunity to update the internal model parameters. An epoch is comprised of one or more batches.

From this Figure 5.10, it is observed that, in agreement with theory, with increase in epochs, the model fit improves but with a few exceptions. It is suggested to always be vigilant for validation loss behaviour. If it increases far away from training loss, it means that too many



(a) Training Loss for different set of epochs



(b) Validation Loss for different set of epochs

Figure 5.10: Model Performance for different epochs

epochs may have caused the model to over-fit the training data. This behaviour is observed for 40 epochs. It means that the model does not learn the data, it memorizes the data. But, the model perform exceptionally well until 30 epochs. Therefore, out of 10, 20 and 30 epochs, the last is more reliable as it mimics the training loss.

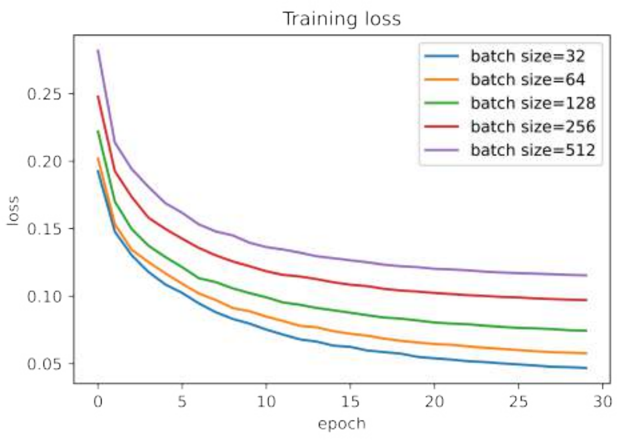
$$\boxed{\text{Number of epochs} = 30}$$

5.1.4.3 TUNING BATCH SIZE

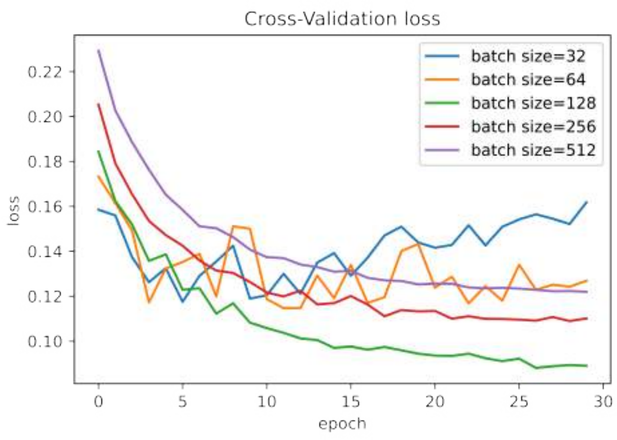
Here, the effect of batch size on training dynamics is investigated. Batch size refers to the number of training examples utilized in one iteration. The size of a batch must be more than or equal to one and less than or equal to the number of samples in the training dataset. With this knowledge, the model is fitted using different batch sizes and the model behaviour is observed. The batch sizes vary from 32, 64, 128, 256 and 512. Since, lr scheduler_5 and ADAM prove to reasonably efficient, model performance is compared for both for different batch sizes, observed in Figure 5.11.

From Figure 5.11, it is inferred that the with decrease in batch size, the model performance improved in training loss in both adam and lr scheduler_5. Validation loss in batch size 32 performs well in both adam and lr scheduler_5, batch size 64 performs well in adam as well.

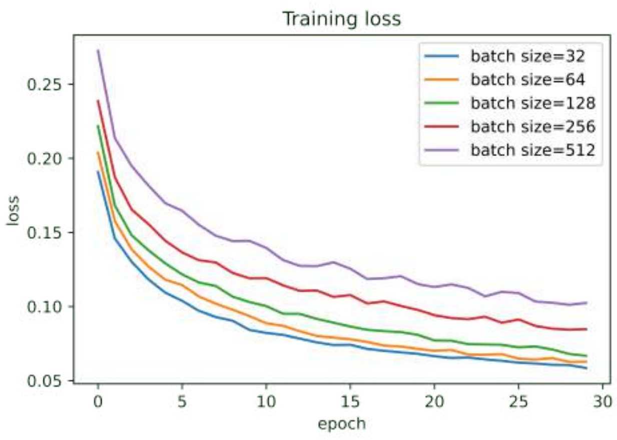
$$\boxed{\text{Batch Size=32}}$$



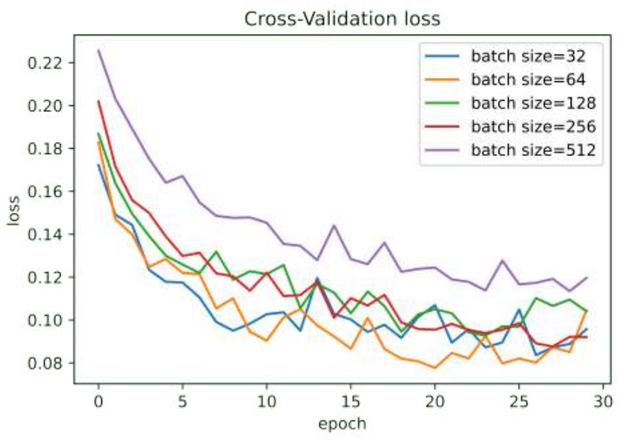
(a) Using lr scheduler_5



(b) Using lr scheduler_5



(c) Using adam



(d) Using adam

Figure 5.11: Comparison of Model Loss for different batch sizes

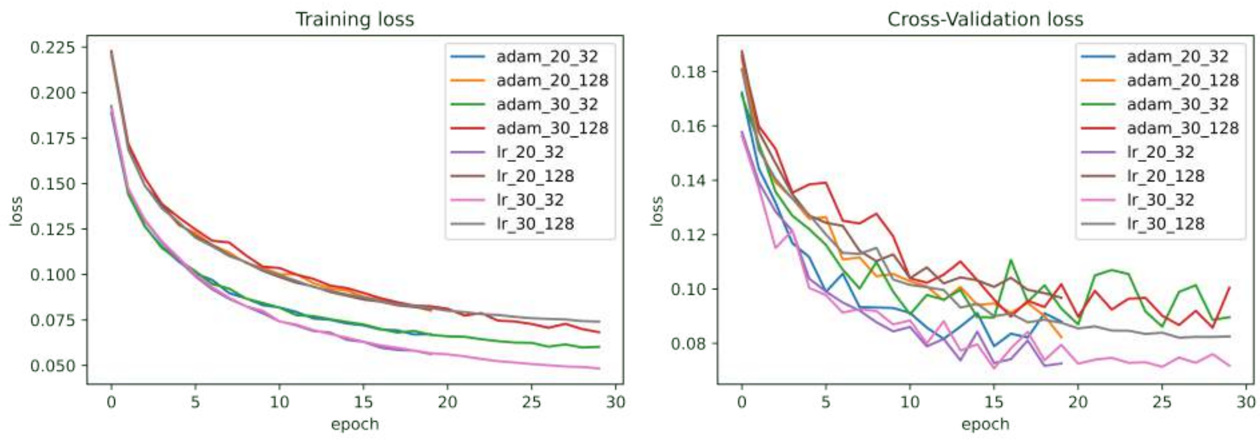
5.1.4.4 COMPARISON BETWEEN OPTIMIZED PARAMETERS

From above sections, the best set of parameters are chosen and interchanged over the model to see the best performance over datasets 5p and 9p. The selected set of tuned parameters are described below:

- adam_20_32
- adam_20_128
- adam_30_32
- adam_30_128
- lr_20_32

- lr_20_128
- lr_30_32
- lr_30_128

In the above set, the notation "a_b_c" is described as a=learning rate, b=epochs, c=batch size. Therefore, if a = adam, it means adam optimizer with default learning rate and if lr, it means learning rate scheduler with 5 epochs; if b=20, it means 20 epochs and if c = 32, it means the batch size is 32. It can be observed from both the Figures 5.12 and 5.13, that the set *lr_30_32* performs better than expected in both the training sets 5p and 9p. However, if there could be some element which could stop the performance in 9p between 6 and 7 epochs, because the Model Loss is at a lowest point before gradually drifting towards "overfitting". This stopper element defined as *Early stopping* is implemented in next description, which will hold the Model's best performance before it deviates any further.

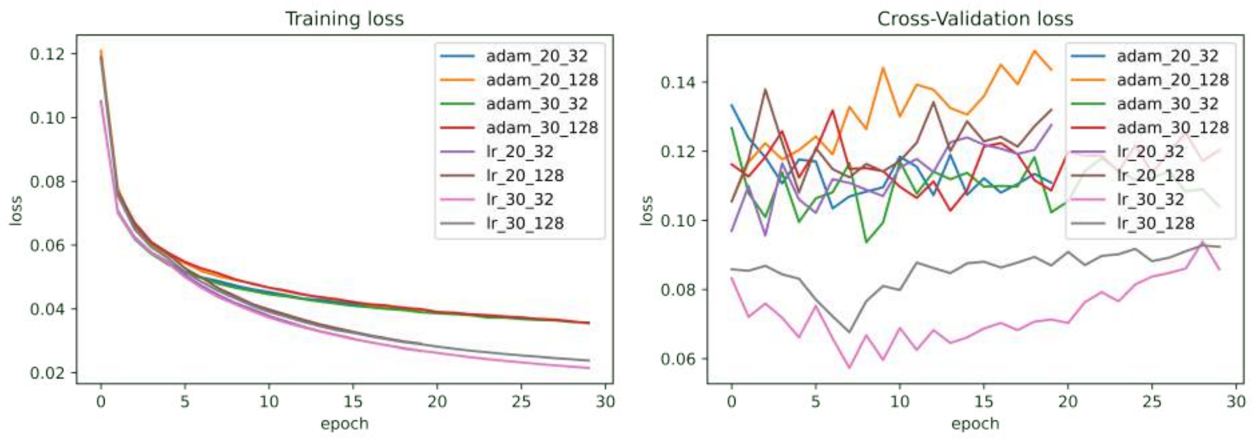


(a) Training Loss for tuned parameters **(b)** Validation Loss for tuned parameters

Figure 5.12: Model Performance for tuned parameters on 5p

5.1.4.5 EARLY STOPPING

From the Figure 5.13, it is inferred that the set "lr_30_32" performs well untill 7 epochs, but slowly drifts away. This could be stopped and should be stopped as the model is at it's best performance.



(a) Training Loss for tuned parameters **(b)** Validation Loss for tuned parameters

Figure 5.13: Model Performance for tuned parameters on 9p

When training a large network, there will be a point during training when the model will stop generalizing and start learning the statistical noise in the training dataset. This overfitting of the training dataset will result in an increase in generalization error, making the model less useful at making predictions on new data.

A major challenge in training neural networks is to train on the training dataset but to stop training at the point when performance on a validation dataset starts to degrade. This simple, effective, and widely used approach to training neural networks is called "early stopping".

"Early stopping" stops training when a monitored metric has stopped improving. The goal of this training is to minimize the validation loss. With this, the metric to be monitored would be 'val_loss', and mode would be 'min'. This is implemented in the current model so that a model training loop will check at end of every epoch whether the loss is no longer decreasing, considering the minimum and patience (number of epochs with no improvement). Once it is observed that val_loss no longer decreases, the training terminates.

5.1.4.6 DROP REGULARIZATION

Dropout is a technique where randomly selected neurons are ignored during training. They are "dropped-out" randomly. This means that their contribution to the activation of downstream neurons is temporally removed on the forward pass and any weight updates are not applied to the neuron on the backward pass. This is implemented on the model for checking if its performance improves.

As a neural network learns, neuron weights settle into their context within the network. Weights of neurons are tuned for specific features providing some specialization. Neighboring neurons become to rely on this specialization, which if taken too far can result in a fragile model too specialized to the training data. This reliant on context for a neuron during training is referred to complex co-adaptations. This is also implemented for two hidden layers.

5.2 EVALUATION OF THE MODEL

In this section, the model is evaluated using virtual experiments.

- Adding noise to the virtually measured strain data;
- Testing levels of loading outside training range;
- If $\sin(\beta)$ and $\cos(\beta)$ can serve as an error indicator.

5.2.1 ADD NOISE TO THE VIRTUALLY MEASURED STRAIN DATA

Add noise to the virtually measured strain data and test until which level of noise the predicted output remains reliable. In order to check until which level of noise the predicted output remains reliable, the virtually measured strain data are perturbed with small values in terms of percentage of range which is (maximum − minimum) value in strain data. A uniform distribution is used to sample the noise levels. The performance is observed for different values of noise : 5% of range, 10% range, 20% range, 30% range and 40% range. The following performance is observed:

From the Figure 5.14, it can be observed that, when compared to the cross-validation loss of model without noise, for 5% of noise, the model performs reasonably, for 10% of noise, the

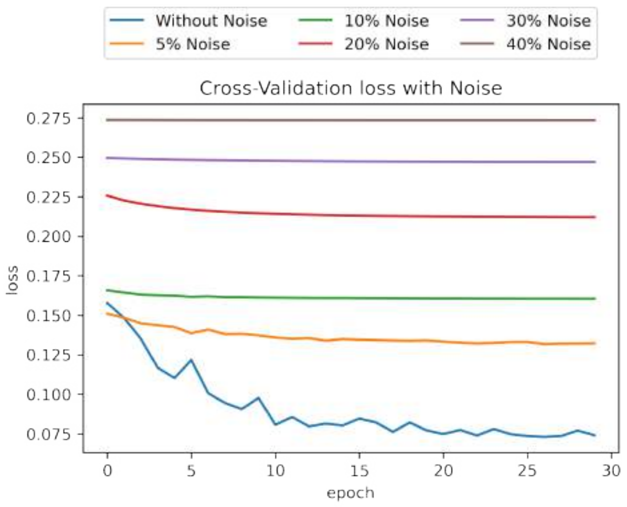


Figure 5.14: Model Performance with Noise

Table 5.2: Model Performance with noise

NOISE PERCENTAGE	5	10	20	30	40
LOSS/ERROR	0.1323	0.1603	0.2120	0.2470	0.2735

loss at last epoch increases by approximately 0.1, for 20% of noise, the loss increases by 0.15 and for 30% of noise, the loss crossed over to 0.25 and reached almost twice the initial loss observed in model without noise . Considering that reliable performance is when loss is below 0.25 or preferably closer to 0, 30% noise is the last reliable noise because the performance after that remains consistent without any convergence. Therefore, it is definitely not reliable to use the data set which has noise more than 30 percent of the range. The evaluated model loss for different noise levels can be observed in Table 5.2. Therefore, the model sustains the performance until or below 30% noice.

5.2.2 TESTING LEVELS OF LOADING OUTSIDE TRAINING RANGE

Initially, the minimum and maximum values of K_I, K_{II} and T are obtained from [2], the same values are perturbed by increasing their range in terms of percentage of (maximum − minimum) value in stress data and checking if the model maintains its performance. The test data is generated for different cases, namely, case 1, case 2, case 3 and case 4 where the

Table 5.3: Loading range for different cases

Parameter	Training Range	Evaluated Range (Case 1)	Evaluated Range (Case 2)	Evaluated Range (Case 3)	Evaluated Range (Case 4)
K_I[MPa$\sqrt{mm}$]	0 to 1200	-10 to 1210	-120 to 1320	-240 to 1440	-360 to 1560
K_{II}[MPa$\sqrt{mm}$]	-110 to 130	-120 to 140	-134 to 154	-158 to 178	-182 to 202
T[MPa]	-60 to 22	-70 to 32	-68 to 30	-76 to 38	-84 to 46
Beta[rad]	0 to 2 π	0 to 2 π	0 to 2 π	0 to 2 π	0 to 2 π
Y[mm]	-35 to -6	-35 to -6	-35 to -6	-35 to -6	-35 to -6
X[mm]	-35 to -6	-35 to -6	-35 to -6	-35 to -6	-35 to -6

Table 5.4: Loss for different loading range

LEVELS OF LOADING	CASE 1	CASE 2	CASE 3	CASE 4
LOSS/ERROR	0.4135	0.4274	0.4499	0.4672

increase in percentage of range is 1, 10, 20 and 30 respectively. The model is evaluated for the test data. The varied loading values are presented in Table 5.3:

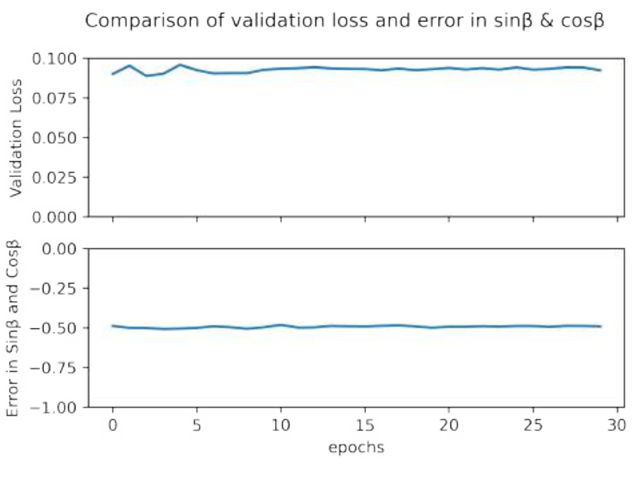
As observed in Table 5.4, the test loss for all cases is very high, almost 4 times the model loss. It can be inferred that if the loading range of K_I, K_{II} and T are outside the training region, the model fails to perform as expected. The reason is the very ideology of Neural network which claims that the test data or the validation data that is evaluated using the model should have the same signature of the model training data. In other words, the range of the test data set should lie within the range of the training data set. Since, the evaluated data is higher than the range of the trained data, the model is supposed to fail. Therefore, the remedy is, if the values outside the training range are desirable then those values should also be included in the training data while modeling a Neural Network.

5.2.3 SINE AND COSINE: ERROR INDICATOR

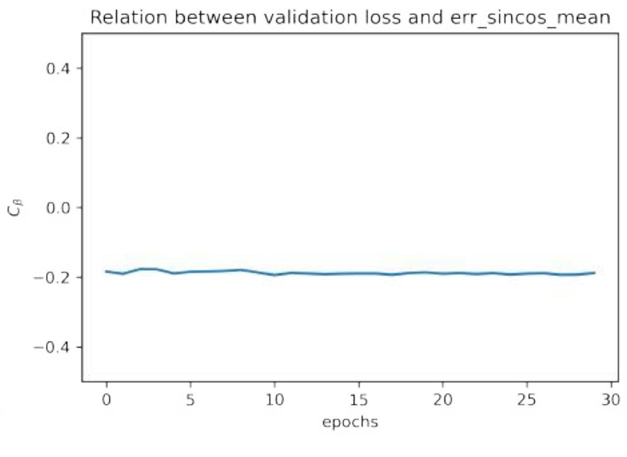
The model is evaluated in the context whether the error in prediction of $\sin\beta$ and $\cos\beta$ can serve as an error indicator in the model loss. The concept behind this idea is that, in the correct regime, $\sin^2\beta+\cos^2\beta = 1$ should hold, following pythagorean trignometric. In order to check if the idiom, $\sin^2\beta+\cos^2\beta = 1$ has any relation to the validation loss, the predicted values between epochs are considered to calculate "err_sincos= $(\sin^2\beta + \cos^2\beta) - 1$". The mean of these values for each epoch is calculated and appended to "err_sincos_mean" . The err_sincos_mean and validation loss from model are plotted to see if there is any similarity between them and if could be related using a constant C_β called as "err". It is observed that err_sincos_mean can actually be used as an error indicator in the prediction of model loss. The following relation holds well:

$$\boxed{\text{Validation loss of the model} = C_\beta * \text{ err\_sincos\_mean}}$$

The Figure 5.15 describes the dependence of prediction error in $\sin\beta$ and $\cos\beta$ over the validation loss which would be calculated using the relation. The constant is observed as -0.1879245066014118 , it is the mean which is maintained through epochs when Validation loss is divided by err_sincos_mean as shown in Figure 5.15b. Therefore, $\sin^2\beta + \cos^2\beta$ can be used as an indicator. This concept is applied for the model within the training region without any noise.



(a) Comparison of validation loss and error in Sin and Cos



(b) Relation between validation loss and err_sincos_mean

Figure 5.15: Sin and Cos as error indicator

This feature helps the user of this model while evaluating the experimental values of strains generated from nanosensor. With the relation between the error in prediction of $\sin\beta$, $\cos\beta$ and validation loss, the user can check if the idiom, $\sin^2\beta + \cos^2\beta = 1$ holds, if not, the error observed is related to the error that can be expected while running the model on experimental data. This feature confirms if the model is suitable for the experimental data or if it should be trained with new values.

The Neural Network Model is built and is successfully solving the inverse problem with a maximum absolute error in readings less than 0.1. This means that the maximum loss will be in the range of +0.1 or -0.1 as MAE is used as a loss function.

In the next chapter, a comparison of different sensor arrangement layouts is performed.

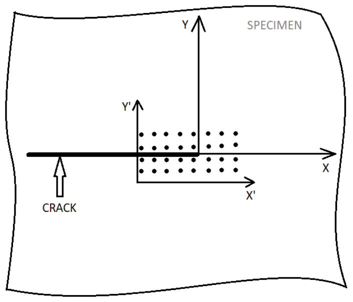
Chapter 6

COMPARISON OF DIFFERENT SENSOR LAYOUTS

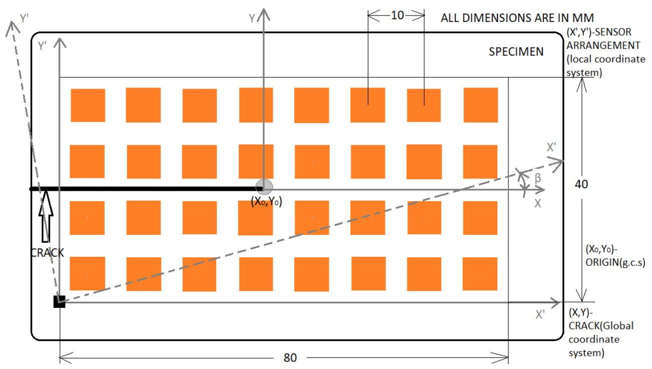
In this chapter, apart from a regular Rectangular (4×4) sensor layout, two other sensor layouts, namely, Rectangular (8×4) layout and Hexagonal layout are described, fitted to a model and their performance is compared with the regular layout.

6.1 RECTANGULAR (8X4) LAYOUT

In this layout, there are 4 rows of sensors with 8 sensors arranged equidistant with each other in each row. The sensor layout has a dimension of 80mm x 40mm and the sensors are placed at a distance of 10mm from each other vertically and horizontally. The total number of sensors are 32 and each sensor has an output of the strain field measured at a distance from the crack tip using Near crack tip solutions by Griffith. The virtual training data is generated similar to the regular 4×4 layout as mentioned in Chapter 3. The schematic of the layout is mentioned in the Figure 6.1. This study or the regression analysis of the data generated is performed to study the behavior of the model and the performance is observed. There a total of 96 input and 7 output neurons with a similar number of parameters like number of hidden layers and hidden units as that of a regular layout. Their performance is compared with model loss and different optimization parameters. The model architecture is described in the Figure 6.2

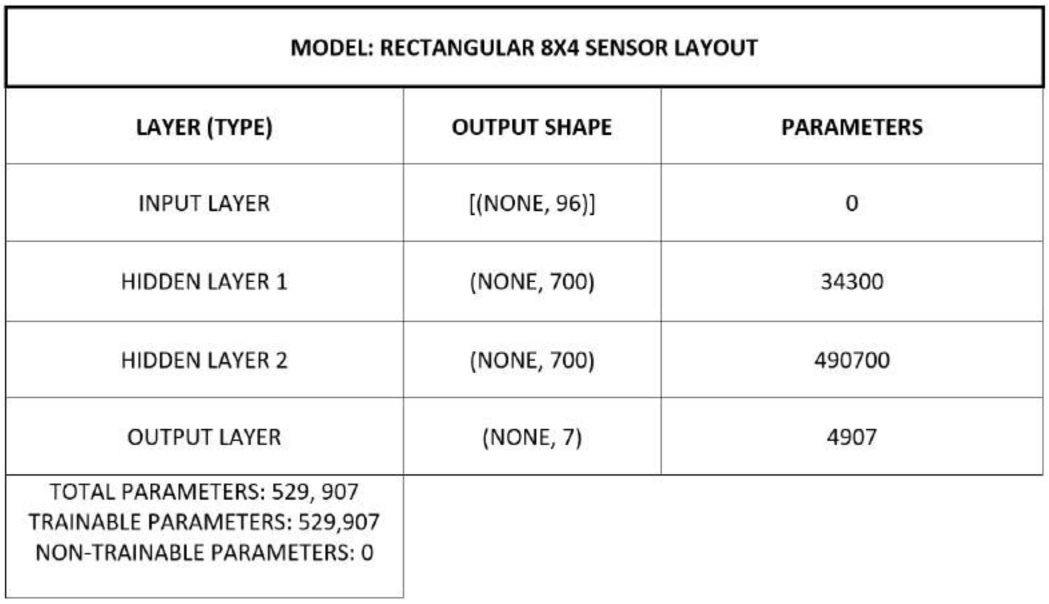


(a) A sketch of 8x4 Rectangular Layout


ALL DIMENSIONS ARE IN MM
10
(X',Y')-SENSOR ARRANGEMENT (local coordinate system)
SPECIMEN
Y
Y'
X
X'
β
40
(X₀,Y₀)
CRACK
(X₀,Y₀)-ORIGIN(g.c.s)
(X,Y)-CRACK(Global coordinate system)
80

(b) Dimensions of 8x4 Rectangular Sensor Layout

Figure 6.1: 8x4 Rectangular Sensor Layout

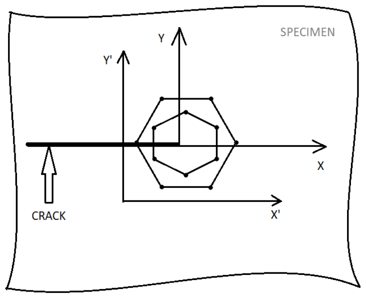


MODEL: RECTANGULAR 8X4 SENSOR LAYOUT		
LAYER (TYPE)	OUTPUT SHAPE	PARAMETERS
INPUT LAYER	[(NONE, 96)]	0
HIDDEN LAYER 1	(NONE, 700)	34300
HIDDEN LAYER 2	(NONE, 700)	490700
OUTPUT LAYER	(NONE, 7)	4907
TOTAL PARAMETERS: 529, 907 TRAINABLE PARAMETERS: 529,907 NON-TRAINABLE PARAMETERS: 0		

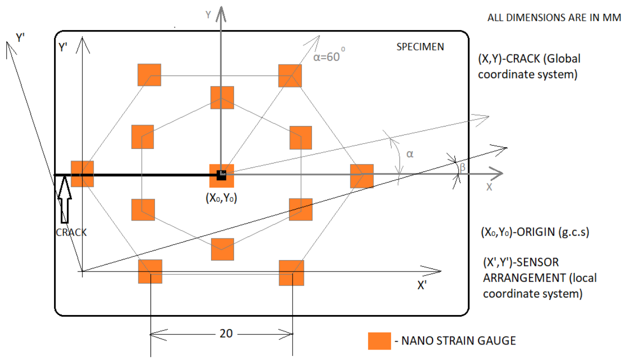
Figure 6.2: Model Architecture of 8x4 Rectangular Layout

6.2 HEXAGONAL LAYOUT

In this layout, there are 13 sensors arranged equidistant with each other and in an angle of $\pi/3$ and $\pi/6$ throughout the layout. The longest diagonal in the sensor layout has a dimension of $40mm$ and the sensors are placed at a distance of 20mm from the centre of the hexagon on all the vertices of the hexagon. A sensor is placed on the centroid of each equilateral triangle in the hexagon along with a sensor at the centre of the hexagon. The total number of sensors are 13 and each sensor has an output of the strain field measured at a distance from the crack tip using Near crack tip solutions by Griffith. The virtual training data is generated similar to the regular 4×4 layout as already mentioned. The schematic of the layout is mentioned in the Figure 6.3. This study or the regression analysis of the data generated is performed to study the behaviour of the model and the performance is observed.



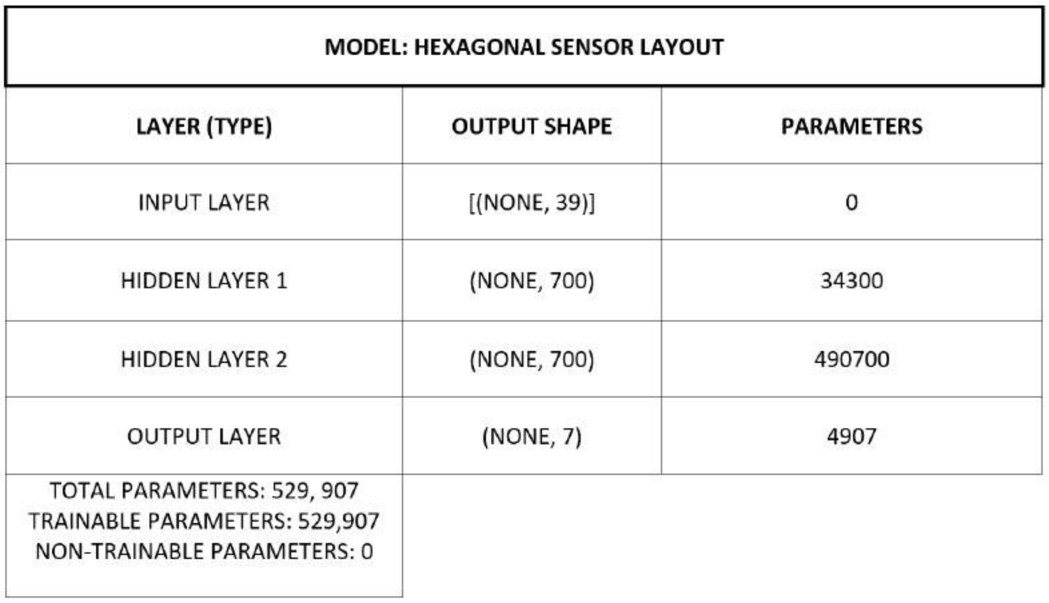
(a) A sketch of Hexagonal Layout



(b) Dimensions of Hexagonal Sensor Layout

Figure 6.3: Hexagonal Sensor Layout

There a total of 39 input and 7 output neurons with a similar number of parameters like number of hidden layers and hidden units as that of a regular layout. Their performance is compared with model loss and different optimization parameters. The model architecture is described in the Figure 6.4

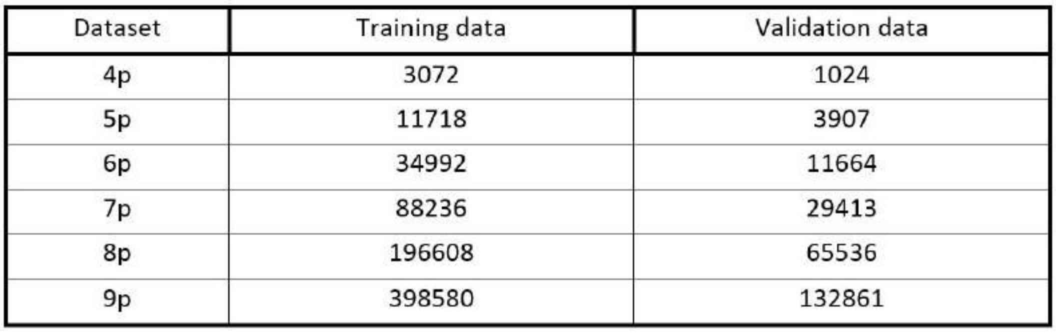


MODEL: HEXAGONAL SENSOR LAYOUT

LAYER (TYPE)	OUTPUT SHAPE	PARAMETERS
INPUT LAYER	[(NONE, 39)]	0
HIDDEN LAYER 1	(NONE, 700)	34300
HIDDEN LAYER 2	(NONE, 700)	490700
OUTPUT LAYER	(NONE, 7)	4907
TOTAL PARAMETERS: 529, 907 TRAINABLE PARAMETERS: 529,907 NON-TRAINABLE PARAMETERS: 0		

Figure 6.4: Model Architecture of Hexagon Layout

Table 6.1: Training and validation examples



Dataset	Training data	Validation data
4p	3072	1024
5p	11718	3907
6p	34992	11664
7p	88236	29413
8p	196608	65536
9p	398580	132861

6.3 COMPARISON OF SENSOR LAYOUTS

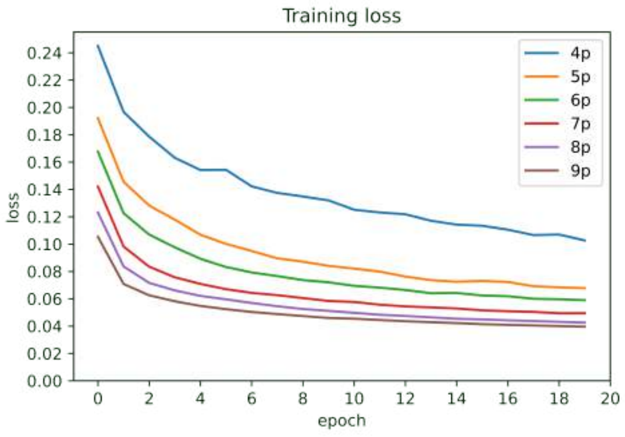
To compare various parameters of the layouts, initially, the performance of different training datasets over different layouts is compared. The datasets range from 4p,6p,7p,8p and 9p as mentioned in Table 6.1 .The loss function is MAE for all layouts.

The comparison plots are plotted for parameters in the below sequence respectively:

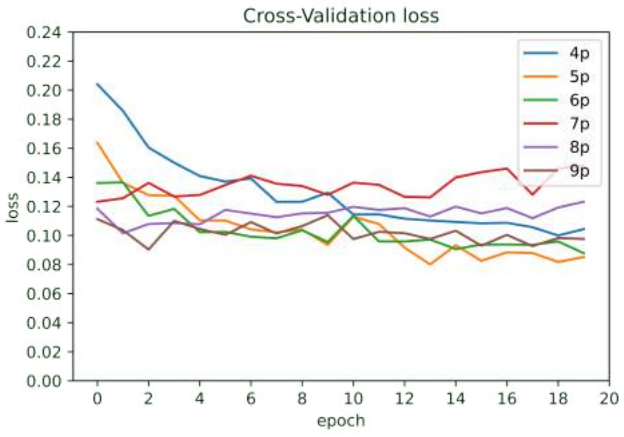
- Training size (4p, 5p, 6p, 7p, 8p, 9p)
- Learning rate (0.01, 0.001, 0.0001, lr scheduler for 5, 10, 15, 20 epochs, adam)
- Epochs (10, 20, 30, 40, 50)
- Batch Size (32, 64, 128, 256, 512)

First, the comparison of model loss for different datasets on each layout is performed. The training loss and validation loss are depicted side by side for a better representation and understanding.

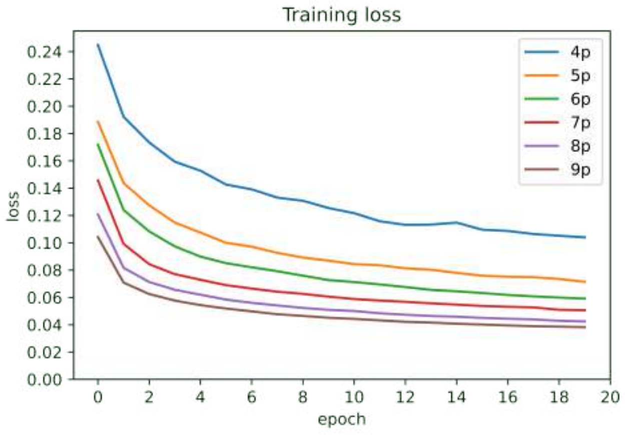
From Figure 6.5, it is inferred that the training loss for all the layouts through all the datasets is gradually decreasing with increasing epochs. The validation loss for $4 * 4$ layout using 5p is lower when compared to other datasets. Similarly, 8p for $8 * 4$ layout and 8p as well as 9p for hexagon layout converge better than others. Minimum loss among all the layouts is obtained in hexagonal and then 8×4 layouts with close to 0.05 and 0.06 respectively. However, lowest computation time is observed in $4 * 4$ layout for 5p.



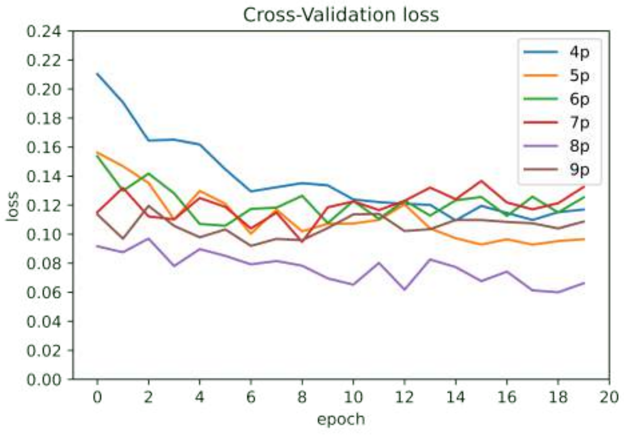
(a) Training Loss for 4x4



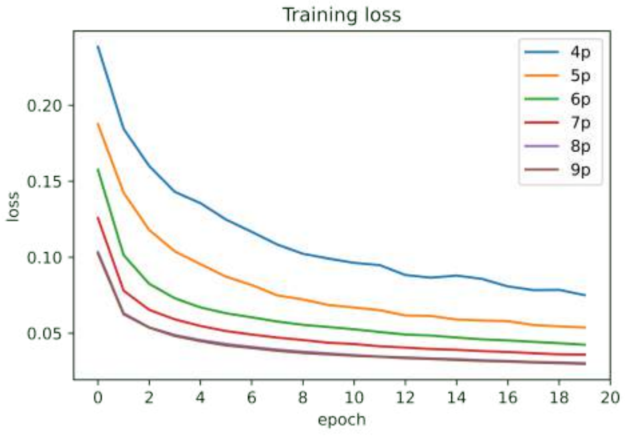
(b) Validation Loss for 4x4



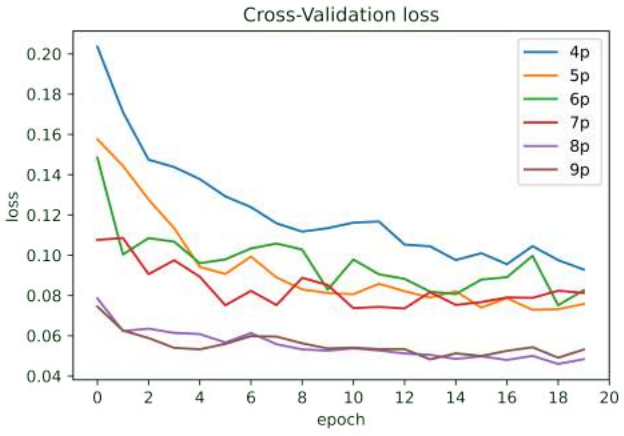
(c) Training Loss for 8x4



(d) Validation Loss for 8x4



(e) Training Loss for hexagon



(f) Validation Loss for hexagon

Figure 6.5: Comparison of model loss over different datasets for different layouts

Second, the comparison of model evolution with varied learning rates on each layout is performed. The rates depict approximation of the model either through an early or a gradual decrease in model loss on different layouts.

From Figure 6.6, it can be observed that the lr scheduler_5 performs exceptionally well on 4×4 and 8×4 layouts whereas lr scheduler_10 perform well on hexagonal as well as 8×4 layout.

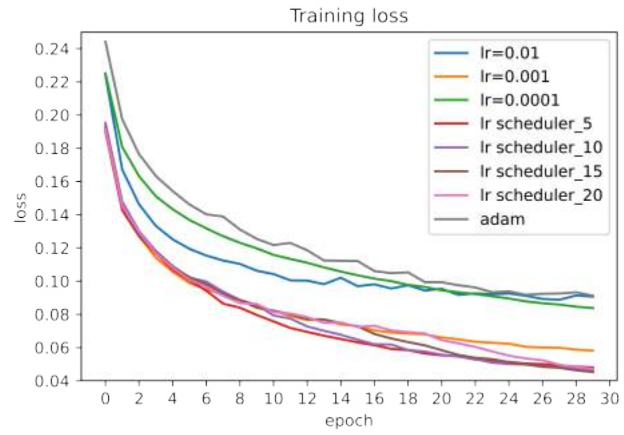
Third, the comparison of model performance at different epochs is studied on each layout. Sometimes, in order to avoid over-fitting of data, the model approximation should be stopped at a point where the validation loss climbs up or remains steady instead of decreasing gradually.

From Figure 6.7, it is studied that the performance of each layout is varied over a range of epochs. It is inferred that the model using 4×4 layout performs efficiently at 30 and 50 epochs with gradual convergence. However, as the loss remains same for 30 and 50 epochs, it is beneficial to stop at 30 epochs to save computation time and avoid over-fit. While at 20 epochs, 8×4 layout shows better performance. However, at 30 epochs, hexagonal layout approaches minimum loss when compared to other epochs and layouts.

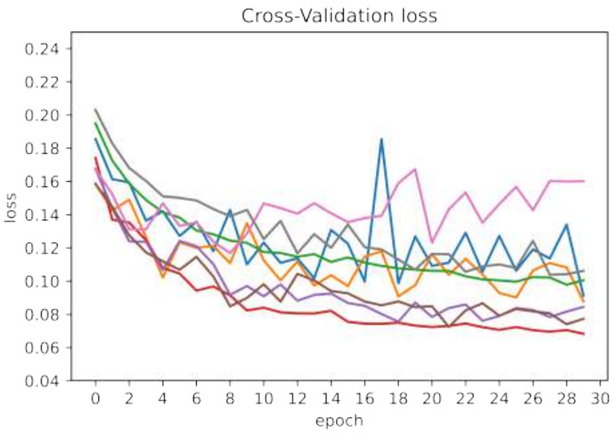
Fourth, the model performance is checked with different batch sizes for applying stochastic gradient descent.

From Figure 6.8, it is understood that lower batch sizes perform well in decreasing training loss in all layouts complying with theory. Whereas, batch size 128 shows good approximations in both 4×4 and $\times 4$ layouts with batch size 64 and 32 performing similarly in the latter. Batch size 64 agrees with hexagon layout reminding the presence of lower number of sensors in its layout. The inference is that for large data (from many sensors), large batch sizes perform well and vice-versa.

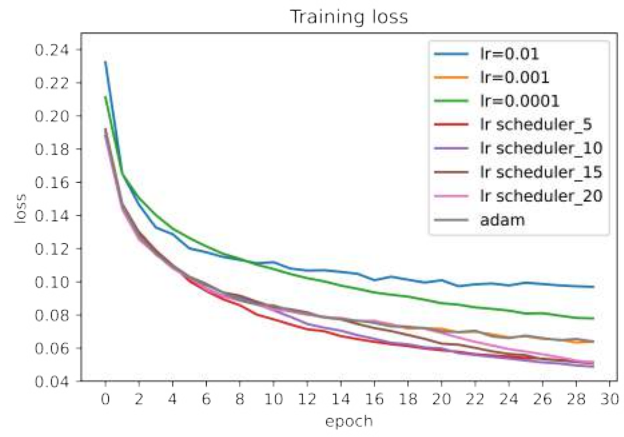
In summary, several parameters are used on different layouts of sensor arrangement and the results are compared to identify the best suitable parameters for each layout and their best performance.



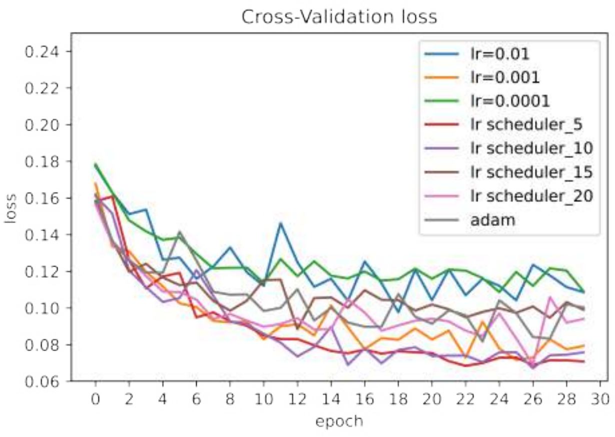
(a) Training Loss tuning Learning Rate for 4x4



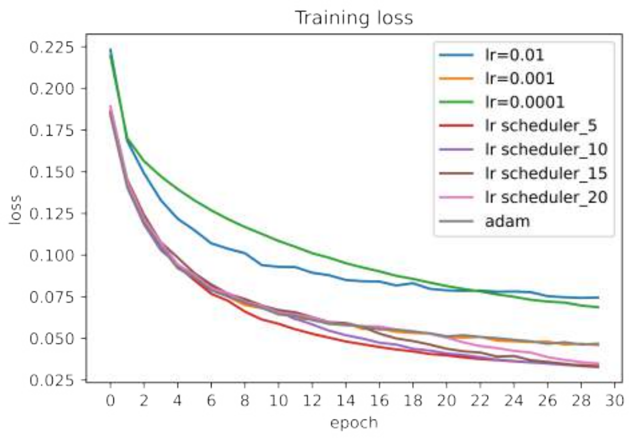
(b) Validation Loss tuning Learning Rate for 4x4



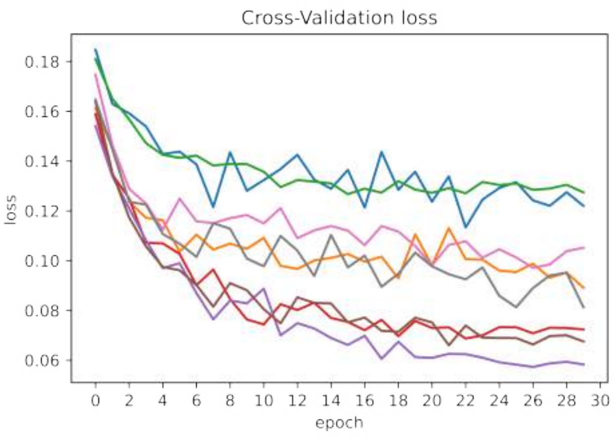
(c) Training Loss tuning Learning Rate for 8x4



(d) Validation Loss tuning Learning Rate for 8x4

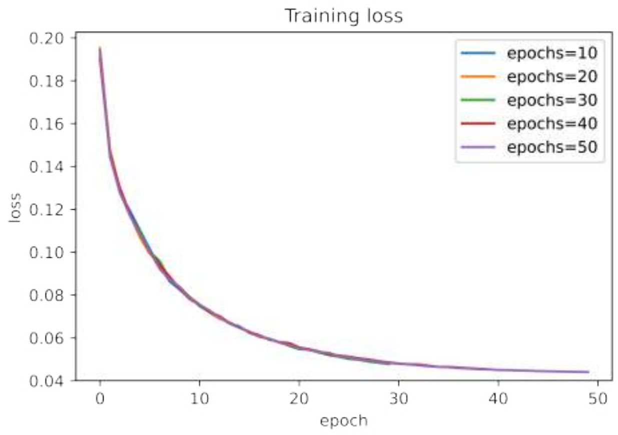


(e) Training Loss tuning Learning Rate for hexagon

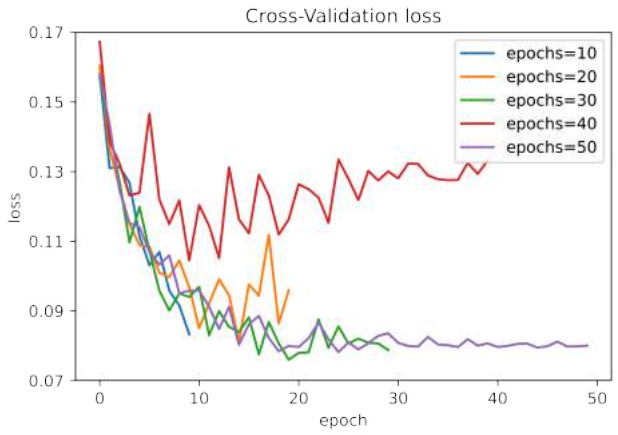


(f) Validation Loss tuning Learning Rate for hexagon

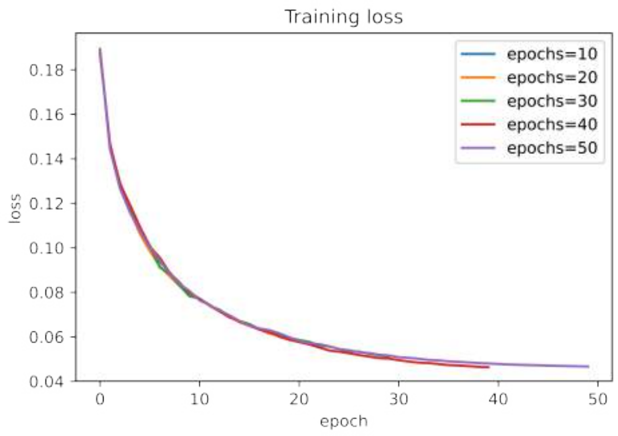
Figure 6.6: Comparison of model loss for different learning rates in different layouts



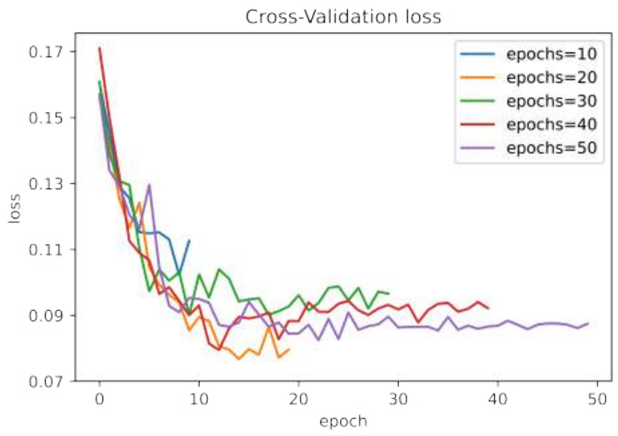
(a) Training Loss tuning epochs for 4x4



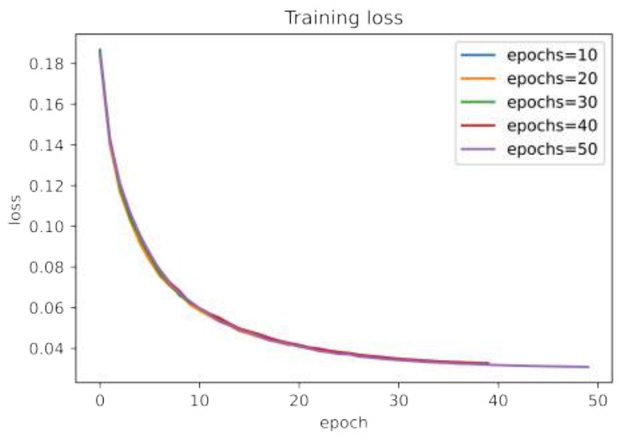
(b) Validation Loss tuning epochs for 4x4



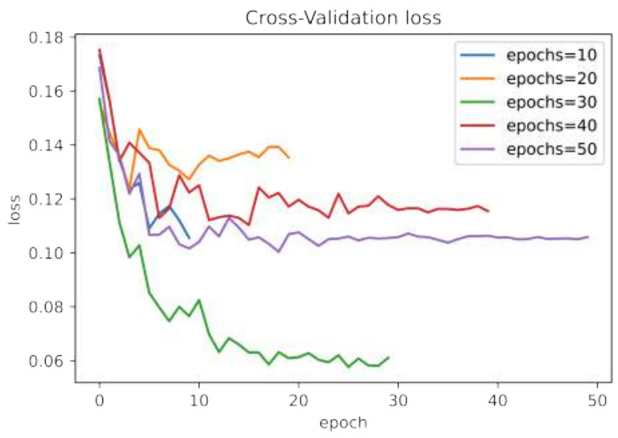
(c) Training Loss tuning epochs for 8x4



(d) Validation Loss tuning epochs for 8x4

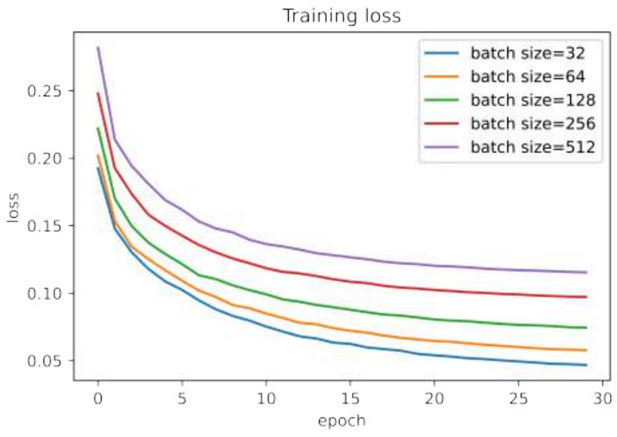


(e) Training Loss tuning Learning Rate for hexagon

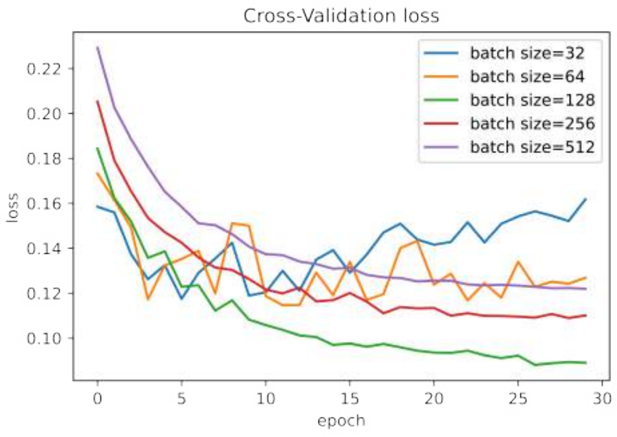


(f) Validation Loss tuning Learning Rate for hexagon

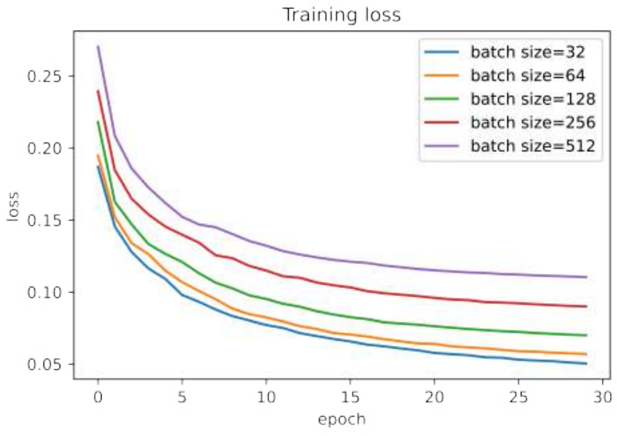
Figure 6.7: Comparison of Model Loss over different epochs for different layouts



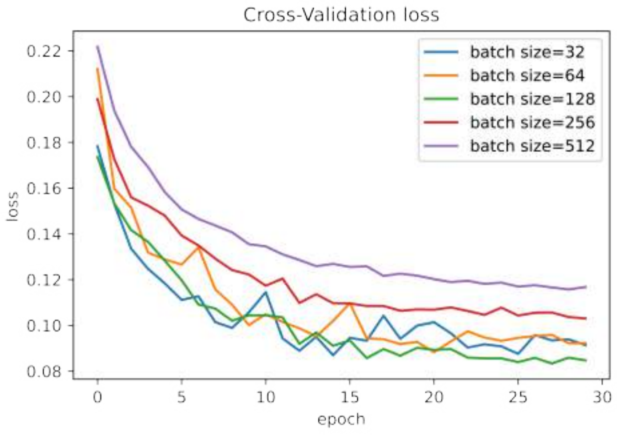
(a) Training Loss tuning batch size for 4x4



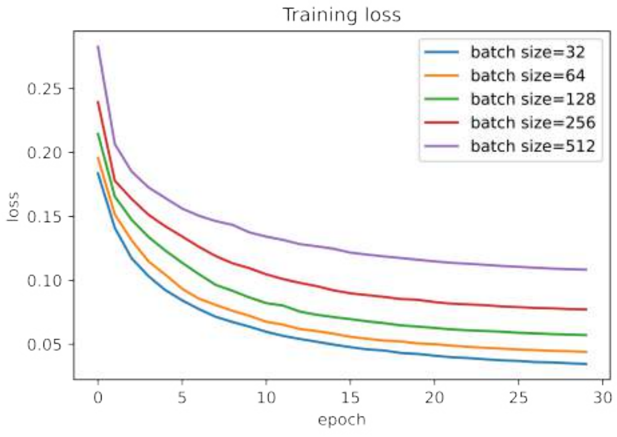
(b) Validation Loss tuning batch size for 4x4



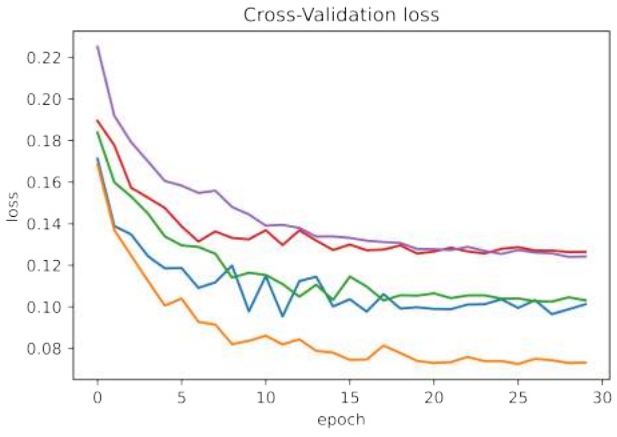
(c) Training Loss tuning batch size for 8x4



(d) Validation Loss tuning batch size for 8x4



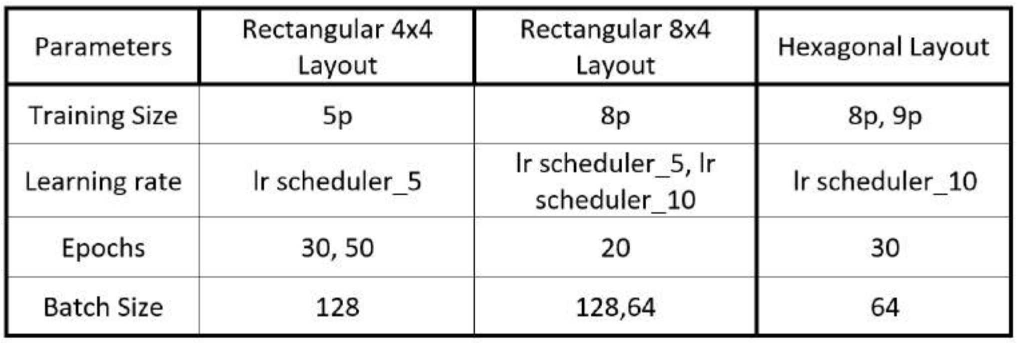
(e) Training Loss tuning batch size for hexagon



(f) Validation Loss tuning batch size for hexagon

Figure 6.8: Comparison of Model Loss over different batch size for different layouts

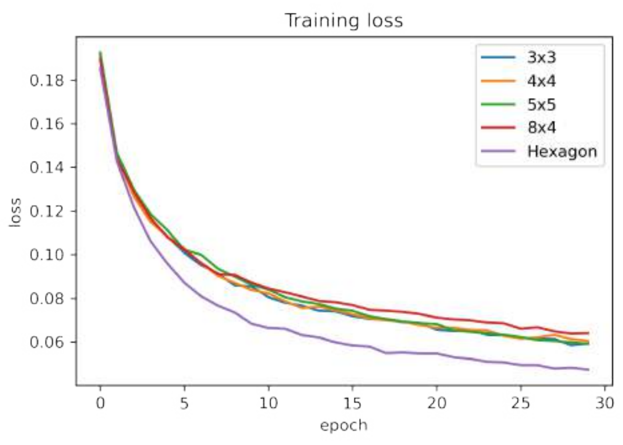
Table 6.2: Optimized parameters for different layouts



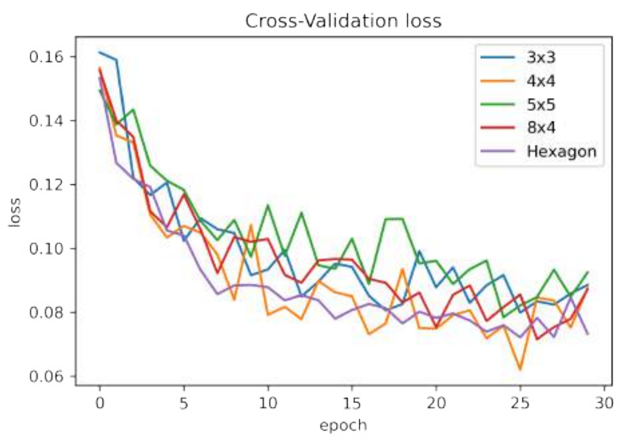
Parameters	Rectangular 4x4 Layout	Rectangular 8x4 Layout	Hexagonal Layout
Training Size	5p	8p	8p, 9p
Learning rate	lr scheduler_5	lr scheduler_5, lr scheduler_10	lr scheduler_10
Epochs	30, 50	20	30
Batch Size	128	128,64	64

Comparison of performance: Several comparisons are performed based on parameter tuning on different layouts. Although, each layout performed well at certain parameters, hyper parameter tuning helps mainly in saving computation time and better convergence. All the optimized parameters for each layout are mentioned in the Table 6.2 upon which desired model performance can be obtained.

However, comparison of tuned parameters gives only the optimized individual performance of the layouts. But, in order to simplify which layout performs better, these layouts should be compared given the same parameters like datasets, learning rate, epochs, batch size and loss function. In this regard, two other regular rectangular layouts, namely, 3×3 and 5×5 are also designed and corresponding training data is generated to compare alongside the other three layouts. Finally, the following layouts are compared for same hyper parameters and their performance is observed.



(a) Training Loss for different layouts



(b) Validation Loss for different layouts

Figure 6.9: Model Loss for different layouts

From Figure 6.9, the following points can be inferred, Rectangular (4×4) layout has the lowest cross-validation loss, approximately 0.06, when compared to other layouts. The layout performs as expected; the validation loss mimics training loss and runs in close range to the training loss depicting almost no over-fitting of the data. Both 3×3 and 5×5 model perform normally but with increasing gap between training loss and validation loss. They are relatively less reliable than 4×4. However, 8×4 performs slightly better than 3×3 and 5×5 keeping a close range between the validation and training loss. On the other Hexagon layout, even though with less number of sensors compared to others, performs exceptionally competing with 4×4 layout. Given that the training loss is minimum for hexagon, the model learns faster relative to other layouts and gives reliable performance.

The better performance of hexagonal layout with training loss when compared to other layouts is because of the placement of the sensors in close quarters to each other. The sensors on the inner hexagon are almost half the distance from those on the vertices of outer hexagon. This indicates that in other layouts, there is an equidistant gap between each sensor but in Hexagon, the sensors are closer to each other. This keeps the readings in close range and better predictions even with realistic data where noise is inevitable. Varied distance between inner and outer hexagon with larger layout perimeter, the hexagonal layout is highly desirable. However, 4×4 layout has a simple distribution and gives a similar performance as that of hexagonal layout.

Overall, among regular or irregular rectangular layouts and hexagonal layouts, the performance is graded as follows:

$$\boxed{\text{Hexagonal} \geq (4 \times 4) > (8 \times 4) > (3 \times 3) > (5 \times 5)}$$

Chapter 7

CONCLUSION

With an aim to study the behavior of Neural Networks over solving an inverse problem, the sensor concept was adapted and applied to the crack tip on the specimen surface. The sensor concept connected all the parameters together and provided a way to calculate the strain field and related loading characteristics virtually, compensating the lack of experimental data. But, the sensor concept provides many ways to explore the innate strength of Neural Networks in providing reliable predictions over various circumstances.

In this thesis, the sensor concept is applied through different layouts of sensor arrangement for which Neural Networks are applied to generate a model that predicts $K_I, K_{II}, T-stress, \cos\beta, \sin\beta, Y_0, X_0$ from strain field obtained from nanosensors. These models are optimized using different parameters to produce minimum model less avoiding over and under fitting of data. In order to understand which layout gives better predictions, they are compared over same hyper-parameters.

Among other rectangular layouts, 4×4 regular rectangular layout gives better model performance along with hexagonal layout which goes hand in hand with the former. Given that the hexagonal performs similar to 4×4 rectangular even though with lesser number of sensors, makes this layout more reasonable. However, the regular 4×4 rectangular layout is appreciated for its simplicity of arrangement. The optimized models of each are provided with proper validations to predict the loading characteristics over the information of strain field obtained from the nanosensors.

The 4×4 rectangular model is tested using virtual experiments to check the performance and limitations. Firstly, noise is added to virtually measured strain data using uniform distribution to sample the noise levels. The model performs reliably until 30% of noise. Secondly, the model is tested for loading levels outside training range. Conceptually, model proved that the range of the test data should lie within the range of the training data, otherwise the model fails to give reasonable predictions. If the test data is outside training range, the model tries to fit it within training range, which is highly undesirable. Thirdly, for the convenience of the user when running the model over experimental test data, a facility to comprehend how much error can be expected from the evaluation, knowing the error in predictions of $\sin\beta$ and $\cos\beta$ is implemented. The error in predictions of $\sin\beta$ and $\cos\beta$ can be extracted from the pythagorean trignometric, $\sin^2\beta + \cos^2\beta = 1$. A relation between validation loss and mean of error in predicting $\sin\beta$ and $\cos\beta$ is determined as C_β.

In [27], [2]and [1], the sensor concept was suggested with an aim to identify crack location from electric potentials from electrodes placed in a close radius arrangement on a film. The process produced results with an accuracy of 1% and 10% in several cases. The research mentioned that there were several limitations in the measurement method like the electrodes have to be placed within a radius [27]; an essential precondition for determining correct solution is the choice of suitable initial values for solving nonlinear equation system, which is not always possible [1]; and good results are obtained for pure mode-I loading and not for pure mode-II loading [2]. In this work, Neural Networks are implemented to fit a data set to a function to predict reliable solutions. All the limitations mentioned above are rectified through this implementation. Whether different types of loading, varied placements of sensors or initial value guessing are all nullified with the help of Neural Networks. Although, parameter selection and tuning plays a crucial role, the implementation is efficient at every step and provides many facilities to adapt the model for a suitable problem. The general limitation of Neural Networks is that it functions efficiently only in the range of training data. But, there is always a facility to train a model with desired output. It provides minimum loss and a scope for much desired scientific experimentation.

Bibliography

[1] M Kuna and D Bäcker. A pvdf sensor for the in-situ measurement of stress intensity factors during fatigue crack growth. *Procedia materials science*, 3:473–478, 2014.

[2] Dennis Bäcker, Andreas Ricoeur, and Meinhard Kuna. Sensor concept based on piezoelectric pvdf films for the structural health monitoring of fatigue crack growth. 2011.

[3] Y Fujimoto, G Liu, Y Tanaka, and E Im. Stress intensity factor measurement of cracks using a piezoelectric element. *Experimental mechanics*, 44(3):320–325, 2004.

[4] Nanolike. Nanosensors from nanolike, 2021-06-20. URL `https://www.sensor-test.de/ausstellerbereich/upload/mnpdf/de/Nanosensorik_Nanolike_Datenblatt_17.pdf`.

[5] Nestor Perez. Introduction to fracture mechanics. In *Fracture Mechanics*, pages 53–77. Springer, 2017.

[6] SS Kulkarni and Jan Drewes Achenbach. Structural health monitoring and damage prognosis in fatigue. *Structural Health Monitoring*, 7(1):37–49, 2008.

[7] TT Nguyen. Neural network architecture for solving nonlinear equation systems. *Electronics Letters*, 29(16):1403–1405, 1993.

[8] Youshen Xia and Gang Feng. A new neural network for solving nonlinear projection equations. *Neural Networks*, 20(5):577–589, 2007.

[9] Yukio Fujimoto, Eiji Shintaku, Gernot Pirker, and Yoshikazu Tanaka. Stress intensity factor measurement of two-dimensional cracks by the use of piezoelectric sensor. *JSME*

International Journal Series A Solid Mechanics and Material Engineering, 46(4):567–574, 2003.

[10] Wikipedia. Strain gauge by wikipedia, 2021-06-20. URL `https://en.wikipedia.org/wiki/Strain_gauge`.

[11] Ronald MacGregor. *Neural and brain modeling.* Elsevier, 2012.

[12] Marvin L Minsky and Seymour A Papert. Perceptrons: expanded edition, 1988.

[13] David E Rumelhart, James L McClelland, PDP Research Group, et al. *Parallel distributed processing*, volume 1. IEEE Massachusetts, 1988.

[14] John McCarthy. Roads to human level ai? *Keynote Talk at Beijing University of Technology, Beijing, China (September 2004)*, 2004.

[15] IBM. What is artificial intelligence?, 2021-06-20. URL `https://www.ibm.com/cloud/learn/what-is-artificial-intelligence`.

[16] Ivan Nunes Da Silva, Danilo Hernane Spatti, Rogerio Andrade Flauzino, Luisa Helena Bartocci Liboni, and Silas Franco dos Reis Alves. Artificial neural network architectures and training processes. In *Artificial neural networks*, pages 21–28. Springer, 2017.

[17] Serokell. Artificial intelligence vs. machine learning vs. deep learning: What's the difference, 2021-06-20. URL `https://ai.plainenglish.io/artificial-intelligence-vs-machine-learning-vs-deep-learning-whats`.

[18] Leonardo De Marchi and Laura Mitchell. *Hands-On Neural Networks: Learn how to build and train your first neural network model using Python.* Packt Publishing Ltd, 2019.

[19] DEEP.AI. Supervised vs. unsupervised learning, 2021-06-20. URL `https://deepai.org/machine-learning-glossary-and-terms/supervised-learning`.

[20] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[21] A.I.Wiki. Neural network, 2021-06-20. URL `https://wiki.pathmind.com/neural-network#:~:text=Neural%20networks%20are%20a%20set,labeling%20or%20clustering%20raw%20input`.

[22] Laurene V Fausett. *Fundamentals of neural networks: architectures, algorithms and applications*. Pearson Education India, 2006.

[23] Nikhil Ketkar and Eder Santana. *Deep learning with python*, volume 1. Springer, 2017.

[24] Wikipedia. Datasets for training, validation and test, wikipedia, 2021-06-20. URL `https://en.wikipedia.org/wiki/Training,_validation,_and_test_sets`.

[25] Andrew Ng. Deep learning specialization, 2021-06-20. URL `https://www.coursera.org/specializations/deep-learning`.

[26] Jeremy Jordan. Why normalize by jeremy jordan, 2021-06-20. URL `https://www.jeremyjordan.me/batch-normalization/`.

[27] D Bäcker, C Häusler, and M Kuna. Piezoelectric sensor for in situ measurement of stress intensity factors. In *Sensors and Smart Structures Technologies for Civil, Mechanical, and Aerospace Systems 2011*, volume 7981, page 79812T. International Society for Optics and Photonics, 2011.

[28] Dietmar Gross and Thomas Seelig. *Fracture mechanics: with an introduction to micromechanics*. Springer, 2017.

[29] Meinhard Kuna. *Finite elements in fracture mechanics*, volume 10. Springer, 2013.

[30] Jeff Heaton. *Introduction to neural networks with Java*. Heaton Research, Inc., 2008.