A PROJECT REPORT (CS321)

ON

MALWARE DETECTION USING MACHINE LEARNING

A report submitted in partial fulfilment of the requirement for the award of

the degree of

BACHELOR OF TECHNOLOGY

In

COMPUTER SCIENCE AND ENGINEERING

Submitted to:

Dr. Atul Kumar Srivastava
Assistant Professor

Submitted by:

Anagh Sharma, CSE A
Kartikey Sharma, CSE A
Vishal Bora, CSE ML
Harendra Singh Bisht, CSE A

SCHOOL OF COMPUTING

DIT UNIVERSITY, DEHRADUN

(State Private University through State Legislature Act No. 10 of 2013 of Uttarakhand and approved by UGC)

Mussoorie Diversion Road, Dehradun, Uttarakhand - 248009, India.

2022

DECLARATION

We here by certify that the work, which is being presented in the second phase of the project, entitled **Malware Detection Using Machine Learning**, in partial fulfilment of the requirement for the award of the Degree of **Bachelor of Technology** and submitted to the DIT University is an authentic record of our work carried out during the period from August 2021 to April 2022 under the guidance of Dr. Atul Kumar Srivastava**.**

Date: Monday, 11 April 2022

Signature of the Candidates

X	X
Anagh Sharma	Kartikey Sharma
X	X
Vishal Bora	Harendra Singh Bisht

Signature of Guide

X

Dr. Atul Kumar Srivastava

ACKNOWLEDGEMENT

This project is a culmination of invaluable guidance and encouragement from various people at DIT University.

We would first like to thank our guide, Dr. Atul Kumar Srivastava for his encouragement and guidance throughout the project. We are wholeheartedly thankful to him for giving us his valuable time & attention and for providing us with regular feedback to help us in progressing our project in time. Then we would also like to thank our friends and family for their support.

Anagh Sharma,	Roll Number: 190102260
Kartikey Sharma,	Roll Number: 190102261
Vishal Bora,	Roll Number: 190178033
Harendra Singh Bisht,	Roll Number: 190102041
Group ID:	CSE19-G58

ABSTRACT

With the rise in complexity of cybersecurity techniques and a simultaneous growth of sophistication with which cybercriminals write and obfuscate malware, the research and industry investment in exploring machine learning techniques to develop more robust security solutions have also increased exponentially. The goal of this study is to study the effectiveness of machine learning in recognizing malicious software.

This project has the following two goals: (1) to study the effectiveness of traditional machine learning methods along with deep learning approaches to identify malicious software, and (2) to identify light weight model(s) that produce minimal computational load on user devices.

The static features of a file are to be used to analyse a file's nature. The files are to be either classified as malware or safe, or further subclassified into various of malware types.

TABLE OF CONTENTS

List of Figures

Chapter 1 – Introduction

1.2. Malicious software

Malicious software, or malware for short, is a program that is designed to compromise a digital system. Malware specimens were initially made for experimentation purposes, to study the vulnerabilities in computer architecture and programming. Malware has rapidly evolved from targeting personal computers to almost every device used today, from mobile phones to ATMs. The diversity of malware and the complexity with which malware is launched will only increase in future, as shown by various studies.

Various organizations broadly classify malware in following categories:

1. Worms: A malware which self-replicates and spreads to other computers through a computer network. Worms can corrupt and steal data, install backdoors for hackers, and consume memory and bandwidth.
2. Virus: The term "virus" was coined by Frederick B. Cohen in his Ph.D. thesis in 1986. He defined the term as: "A program that can infect other programs by modifying them to include a, possibly evolved, version of itself.".
3. Trojan horse: A software which disguises itself as a legitimate program and when downloaded and installed, is used to disrupt or damage data or a network.
4. Spyware: A malware that installs itself on a computer to monitor the behaviour and to steal sensitive information from a user. Spyware can also be used to grant remote access of the compromised system to predators.
5. Adware: This type of software is designed to help companies generate more revenue by automatically displaying advertisement banners and pop-ups while another program is running.
6. Ransomware: A program which threatens to perpetually block or publish personal user data unless a ransom is paid. The attacker does so by using a disguised link to trick the user into downloading the malware file which then encrypts the user data. The data can then only be unlocked through a secret key which is usually promised to be revealed upon payment.

1.2. Machine learning

Machine learning is a part of the broader subject of artificial intelligence which is used to enable a machine to use real-world data in order to solve a problem. Machine learning is a combination of statistics, applied mathematics, and computer science that allows computers to automatically improve their performance from experiences and without any additional programming to make those improvements. A model training algorithm is used to generate a model which can generalise well on unseen data.

Machine learning is generally classified into the following three paradigms:

1. Supervised Learning: This methodology uses real-world datasets which consist of training features and associated labels to generate an inferred function which can then be used to label unseen and unlabelled data.
2. Unsupervised Learning: In unsupervised learning approach, an unlabelled and uncategorized dataset is used to train a machine learning model which must discover patterns to associate features with labels by itself.
3. Reinforcement Learning: This approach involves training through reward and punishment of the behaviour of an intelligent agent, which takes actions with the goal of maximizing cumulative reward.

Deep learning, is a subfield of machine learning wherein, features are extracted from raw data progressively, from lower to higher degree of abstraction. Deep learning algorithms are composed of multiple layers arranged in a hierarchical manner of increasing complexity.

Chapter 2 – Machine learning for malware detection

2.1. Introduction

Earlier, the methodology used for malware detection involved manual configuration of malware fingerprints by analysts and security experts. This signature-based malware detection was used extensively and involved detection through manually configured and regularly updated pre-execution rules. These rules were based on the features of executable files such as fragments of code and several other properties of software files.

But this methodology eventually became obsolete due to the substantial increase in the volume of malware produced that made manual configuration of fingerprints unrealistic since even a small change in a file's properties could render these rules ineffective.

This has led to the exploration of machine learning and deep learning-based malware detection approaches which are able to use intelligent solutions in order to keep up with the evolution of malware. The following methods are used in malware detection:

1. Static analysis: Features of a program are extracted and used to predict its nature without executing the code. Such features include but are not limited to file format, binary data of the file, text strings, etc. This method is least computationally expensive but could fail to detect malware if it uses effective code obfuscation.
2. Dynamic analysis: This method takes a behaviour-based approach in examining an executable file. The executable file is run and its behaviour is studied on either an air-gapped machine, a virtual machine or a sandbox. The behaviour includes API calls, memory writes, registry changes, etc.
3. Hybrid malware detection: This method combines both static and dynamic methodologies.

Deep learning-based approaches do not require expertly selected feature configurations based on domain knowledge. Instead, deep learning involves approaches that include:

1. Extracting a feature vector to represent the executable.

2. Examining grayscale image pixel intensity data that is produced using the binary content of a file.
3. Studying API call traces.
4. Examining binary representation of a file, etc.

2.2. Study of datasets

Following the machine learning pathway, the first step in this project is to explore various datasets suitable for this study. The data samples explored in this project are executable malware associated with different malware families. The executable files are studied without their execution as this project studies static malware detection.

This is a fundamental stage of the machine learning pathway and involves identifying appropriate data sources and available datasets. Some of the attributes of a dataset that are used to determine its usefulness are describe below:

1. The size of the dataset
2. The source and age of the dataset
3. Feature representation
4. The number of labelled samples

The following datasets were studied in this project:

1. Microsoft Malware Classification Challenge (BIG 2015) [3]: This dataset was published by Microsoft as part a malware classification challenge hosted on Kaggle in 2015. When uncompressed, this dataset contains over half a terabyte of data and consists of bytecode and disassembly code from over twenty thousand malware files.

   The malware samples which are represented in this dataset belong to nine malware families. These malware families, or classes, are shown in fig 1.

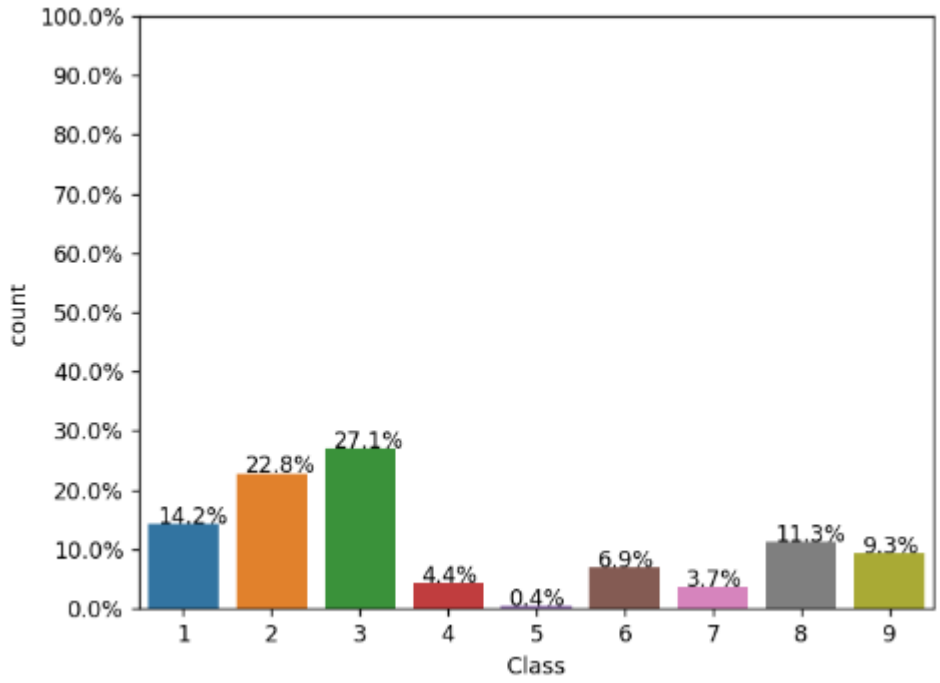

Fig 1: Malware classes in MMCC dataset
(Source: Rayudu Yarlagadda)

The MMCC malware samples have the following specifications:

a. A name: MD5 hash value to uniquely identify the file
b. A label: Integer representation of one of the nine malware families

Each malware sample has two files associated with it which are described in fig 2.

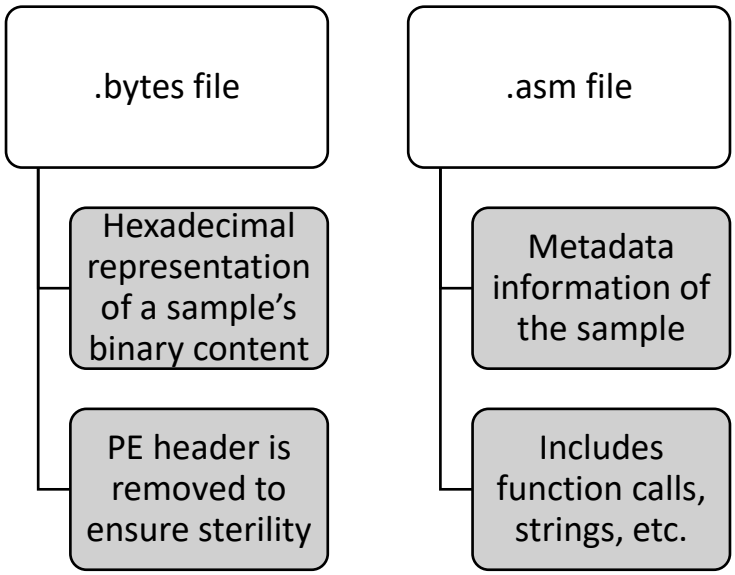

Fig 2: Malware sample files

2. Sophos-ReversingLabs 20 million dataset [4]: This dataset is hosted on Amazon S3 storage service and contains approximately over eight terabytes of data. This is a large-scale dataset comprised of almost twenty million files with pre-extracted features and metadata, labels derived from various sources, along with "tags" related to every malware file which serve as additional targets. Additionally, the dataset contains over ten million disarmed malware samples.

   The SoReL-20M dataset contains the following data for each malware sample:

   a. Features of the samples that are derived in accordance with the format of the EMBER 2.0 dataset.
   b. Labels for each sample which are obtained by using both external as well as internal Sophos sources.
   c. PE metadata of malware files obtained using pefile library.
   d. Binary files of malware samples.

3. Malimg (Nataraj et al., 2011) dataset [5] [6]: This dataset contains PNG image representation of nearly nine thousand malware files. Over 25 malware families are represented in this dataset.

   The malware families represented in this dataset are shown in fig 3.

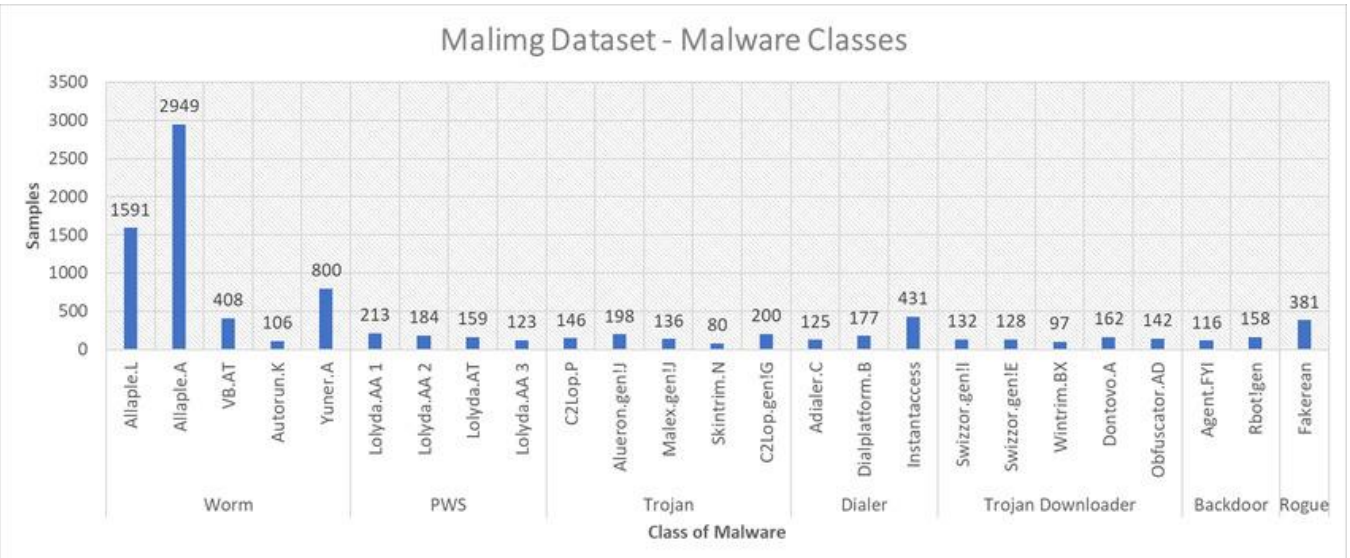

Fig 3: Malware classes in Malimg dataset
(Source: Abhijitt Dhavlle)

2.3. Portable Executable file format

The Portable Executable, or PE format is employed for executable/dll files in the Windows environments and was first used in Windows NT operating system. The contents of the file are composed in a linear manner. Features are extracted from this file to train a machine learning model which is used in static malware analysis.

2.4. Feature engineering

The raw data of the samples needs to be transformed into features which can be used to effectively train a machine learning model.

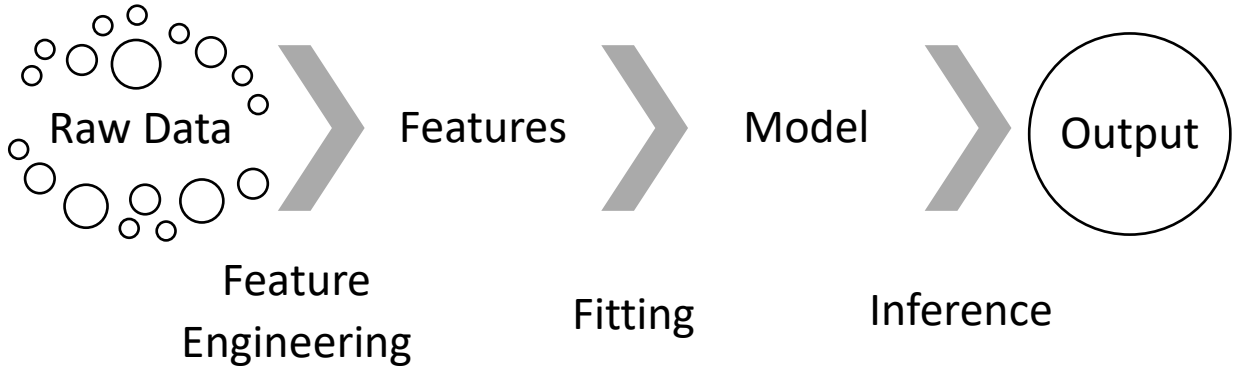

Fig 4: Feature engineering in machine learning pathway

In this instance of feature engineering, the hexadecimal content of the malware file samples is transformed to produce a PNG image representation of the malware file.

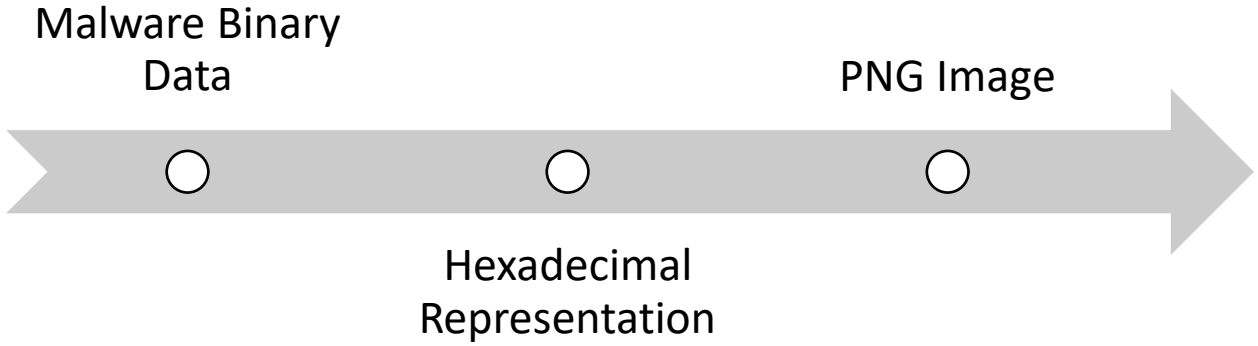

Fig 5: Binary file to PNG representation

Feature engineering is a continuous process in the machine learning pathway and image representation of malware files is only an example of possible features.

The PNG images as such cannot be directly fed to the model since there is a great variance in sizes of malware images. These images need to be processed first so that they are of a common scale. The ImageDataGenerator.flow_from_directory() method of the TensorFlow python library is used to generate this tensor image data as shown in fig 6. This demonstration uses only ten samples each from all the malware classes.

Generating normalized tensor image data using Keras

```
from keras.preprocessing.image import ImageDataGenerator

#Generating DataSet
the_data = ImageDataGenerator().flow_from_directory(directory=root,␣
 ↪target_size=(512,512), batch_size=100)
imgs, labels = next(the_data)
```

```
Found 90 images belonging to 9 classes.
```

Fig 6: Generation of tensor image data using Keras

A sample of normalized tensor image data created using the TensorFlow python library is shown in fig 7. This data is used to train a deep learning model.

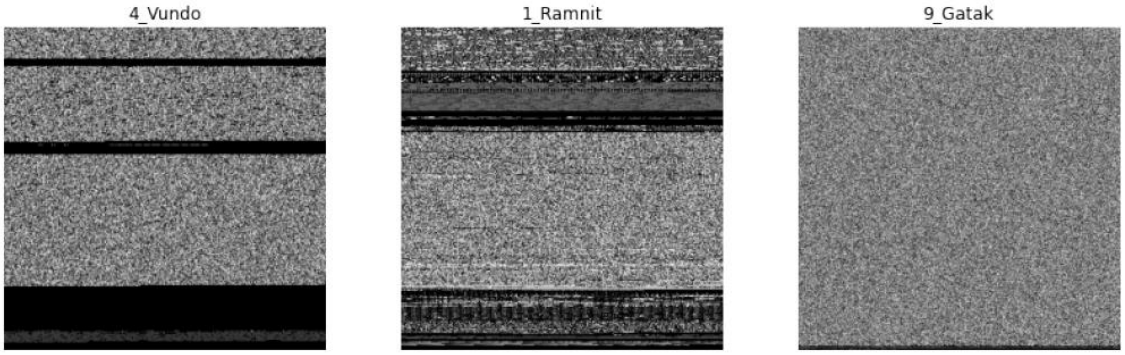

Fig 7: A sample of PNG malware image representation

Feature engineering is the most time-consuming stage of the machine learning pathway and involves deep statistical analysis of the data. An exhaustive study of the features on the basis of domain knowledge and with respect to specific machine learning models is required.

Finally, after feature engineering the resultant data is split into the following three randomly generated subsets using the train_test_split() method of the Scikit-learn python library:

1. Training set: Used to train a machine learning model to help it understand the relation between the features and labels.

2. Validation set: Training dataset is not used to entirety during training. Instead, a hold-out set, or validation set, is used to measure the performance of a model after each epoch. Different subsets of the training set are used to assess the performance of a model over different epochs.

3. Testing set: The performance of a model is assessed using this subset by the use of several evaluation metrics.

2.5. Generic machine learning models for malware detection

Various generic machine learning algorithms can be used to construct models that effectively classify malware samples. The use of n-opcode sequences is an example of one such strategy where the n opcodes are extracted from the binary data of the malware executables and algorithms such as random forests, k-nearest neighbours, chi-squared test, and support vector machines, etc are used to effectively classify the malware samples. Naman Bagga [6] in thesis has studied the performance of models using n-gram frequencies as the features of the MMCC dataset samples.

The fig 8. describes the feature selection used in the cited work [6].

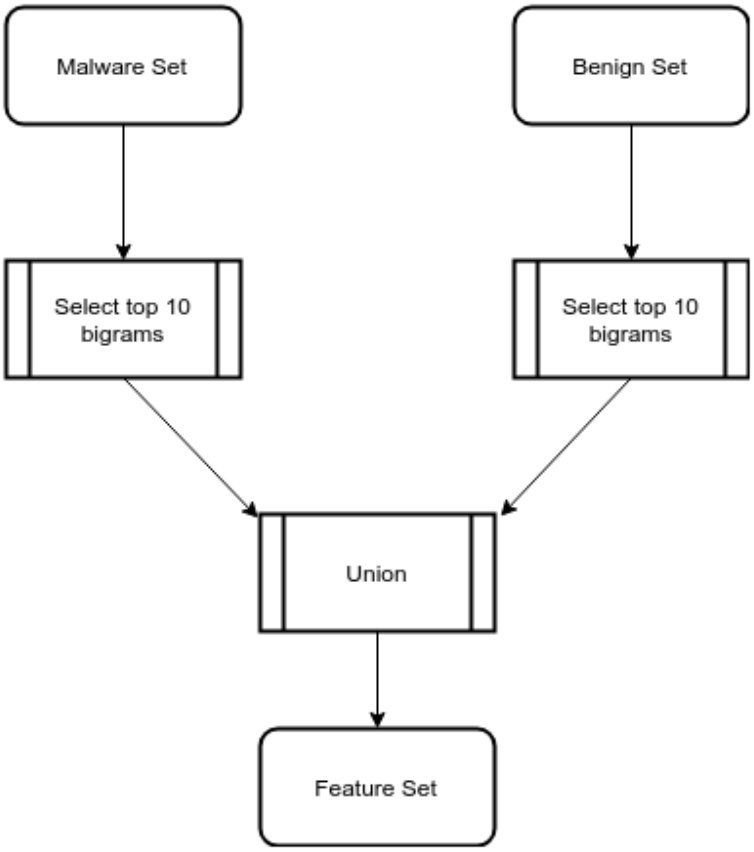

Fig 8: Feature Selection in MMCC
(Source: Naman Bagga)

The table 1 describes the performance of some of the most common machine learning models in classifying malware files.

Machine learning algorithm	Average accuracy
Support Vector Machine (SVM)	0.73
Chi-squared test	0.80
k Nearest Neighbours	0.86
Random Forests	0.88

Table 1: Performance of generic machine learning model in malware detection
(Source: Naman Bagga)

A more useful strategy for malware detection is dep learning. The following sections explore their effectiveness in malware classification.

Chapter 3 – Deep learning

3.1. Concept

Deep learning is a machine learning technique that utilizes neural networks to perform automatic feature extraction to classify these features in a hierarchical manner. The initial layers of a deep neural network train to classify the more fundamental aspects of the image and every subsequent layer generalizes these classes, with the very last SoftMax layer producing the class label of an input.

The accuracy of a deep learning model continues to increase with the size of the dataset unlike that of the generic machine learning models as highlighted by the pioneer of the field Andrew Ng on several occasions. The fig 9. used by Andrew Ng in his presentations illustrates this contrast between deep learning and the generic models.

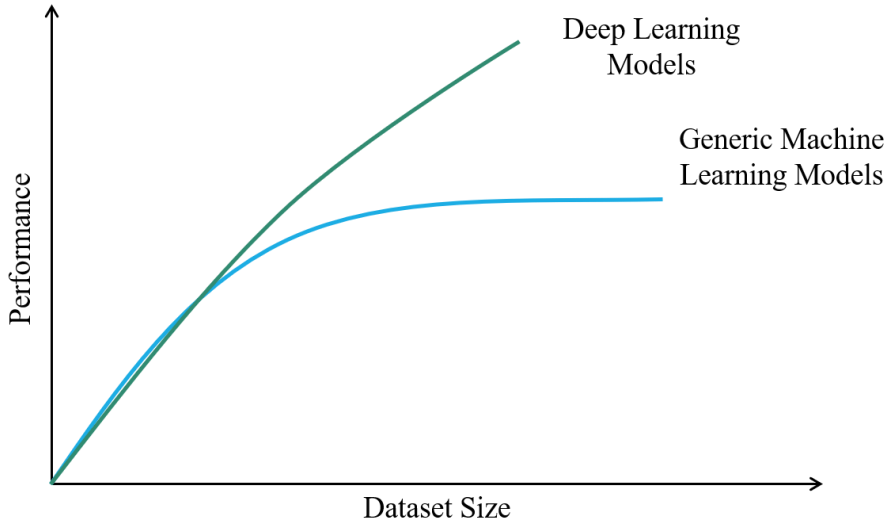

Fig 9: Relation between performance and dataset size in machine learning

The deep learning paradigm is marked by its ability to use non-pre-processed data to train models that achieve high prediction accuracy. The use of data that has not been pre-processed reduces the time required to build and maintain an A.I. software.

Deep learning models, or deep neural networks are composed of several layers made up of nodes that use "weights" and "biases" to function. These nodes called neurons are the most fundamental units of a neural network. Each neuron representing a hypothesis function can be thought of as a regression algorithm with weights for parameters that are updated during training.

A neuron, also called a perceptron illustrated in fig 10. is the basic unit in a deep learning model through which computations propagate. In the figure, $\boldsymbol{x}$ variables denote the input, $\boldsymbol{w}$ the weights, associated with each input stream, and $\boldsymbol{b}$ value denotes the bias associated with the neuron which is analogous to the y-intercept in the equation $\boldsymbol{y = mx + c}$. The bias helps us in customizing the output of the model to fulfil our model training goals.

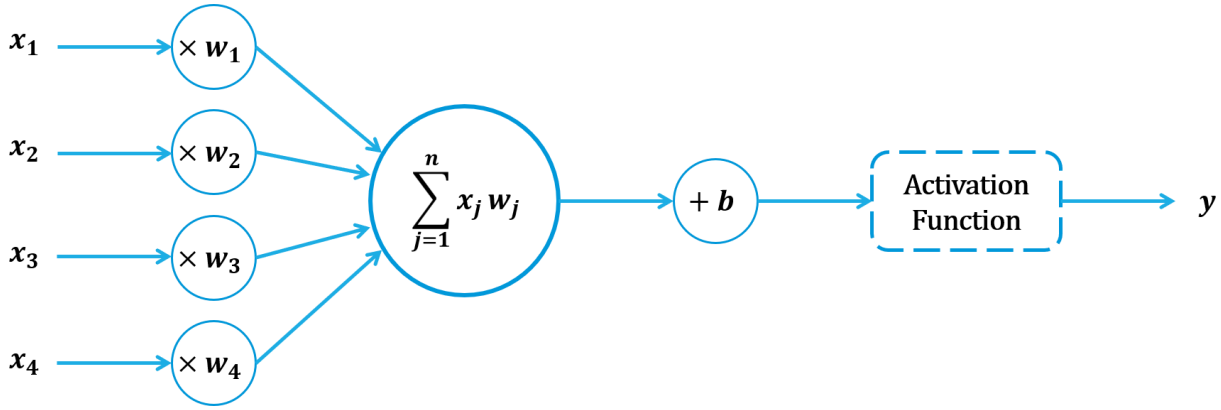

Fig 10: The structure of a perceptron

The following computation taking place in a neuron is as described in the equation below.

$$\boldsymbol{y = f(x_1 w_1 + x_2 w_2 + x_3 w_3 + x_4 w_4)}$$

The function $\boldsymbol{f}$ is the activation function of the neuron which is used to constrain, and determine the output of a neuron, for example between 0 and 1, i.e., it introduces non-linearity in the computations an restricts the output some functionally defined values. The most common activation function used in deep learning is the rectified linear unit, or ReLU. The ReLU function maps all negative inputs to zero and all positive inputs and zero inputs to itself.

$$\boldsymbol{ReLU(y) = \max(0, y)}$$

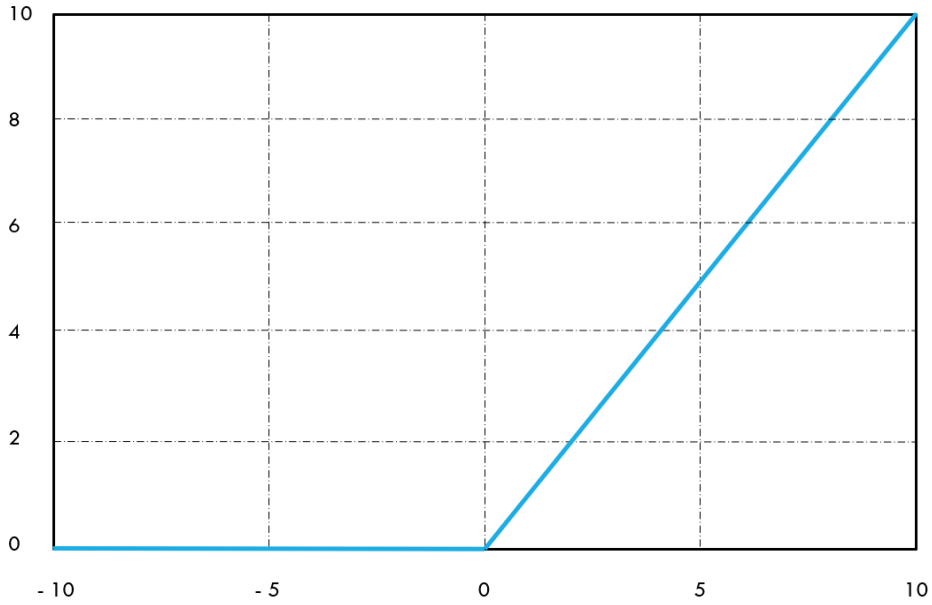

Fig 11: Activation function, ReLU

A neural network is composed of an input layer, output layer, and hidden layers between them that carry out the computations. It may be called deep, or shallow depending upon the number of hidden layers.

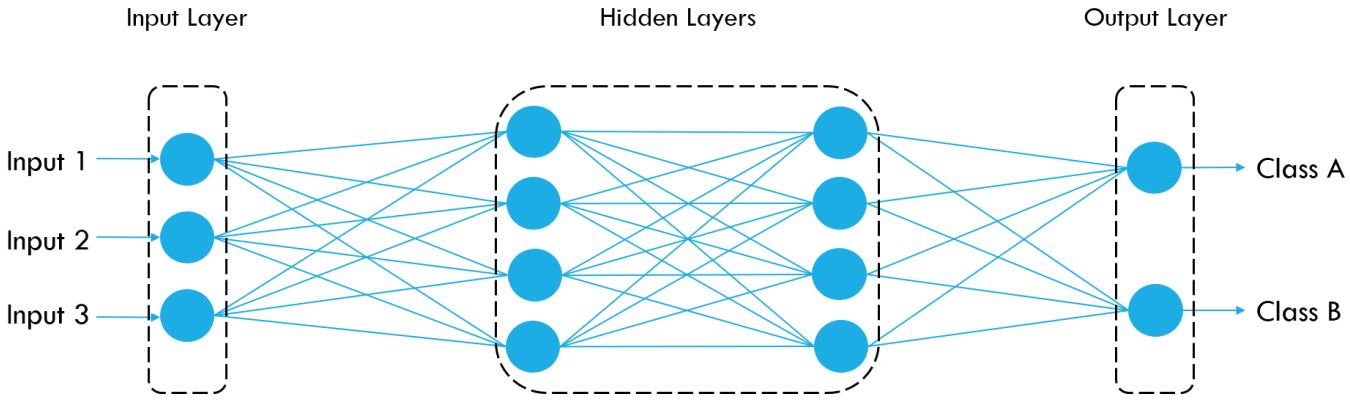

Fig 12: A shallow fully connected artificial neural network

3.2. Learning layers and their types

A neural network can utilize several types of neural layers depending upon the functionality required.

Some of the most commonly used neural layers are:

3.2.1 Fully connected layer

Such layers connect each neuron in a layer with each neuron of the subsequent layer. Consequently, the computations performed are numerous due to many combinations of neurons and thus the data produce is also large and partly redundant. Due to the redundancy, of data dropout is generally used to discard the redundant information.

3.2.2 Convolutional layer

In an artificial neural network, a unique weight parameter is multiplied with each input to a neuron. In a convolutional layer, or a convolutional neural network, instead of associating a weight with each input, a singe filter, or kernel is used that is applied across the input vector, over a series of steps left to right and top to bottom. The step size is defined as the stride hyperparameter.

The convolution of a filter on a similar sized section of the image or a “patch” of the image will define the existence, or lack thereof, of a feature of interest in the image for which that filter has been trained to recognise.

A small sized filter is used that allows the same weights to be used to for all the values in the 2D input matrix. The alternative would be to use a filter of the size of the input image that would produce an enormous computational load. The usage of a filter much smaller than the input also aids in the search for features of interest throughout the input image matrix and not just in a locality of the image.

The resultant of the convolution of a filter with an input vector is a feature map. Since a neural layer will apply multiple filters over an input, as there are multiple neurons in a neural layer, the output of the layer is n number of feature maps produced by n number of filters applied on the image. Subsequent layers will take this feature map as input.

A feature map defines the existence of a feature at a specific location. The initial layers are trained for the fundamental, or small features in case of image recognition, and every subsequent layer defines the existence of a higher, or larger features. The outputs of the neurons of the last layer of a convolutional neural network are flattened, or converted to a one-dimensional array and are then joined end-to-end. The resultant one-dimensional array is fed to a fully connected neural layer, i.e., each value in the one-dimensional array is fed to each neuron in the fully connected layer. Each neuron recognizes the features in the image at a specific location or orientation so that the features in the image are recognized however they may be augmented relative to the training data.

The convolutional neural layers recognize specific features in the input, and hence perform a feature recognition operation. The fully connected layers that use the flattened array input recognise the coexistence of all these features, in several possible augmented forms, producing a class label and therefore perform a classification task.

3.2.3 Pooling layer

While the use filters in convolutional neural layers does reduce the computational load, and a lack of padding also produces an iteratively smaller output, the computations required are still enormous and can be reduced by removing redundant information from the feature maps. The adjacent cells of a feature map usually contain data that define the same features and therefore represent redundant information. This redundant information, or information of neighbouring cells can be reduced by accumulating the information of these adjacent layers to single value. This is commonly done by selecting the maximum value of the information in the adjacent cells and discarding the values of all other cells. The filters of the pooling layer’s nodes produce new and smaller feature maps from the existing feature maps to reduce computational load. A smaller feature map also signifies smaller number of features aiding in noise filtering, and also preventing overfitting to the dataset, as there are smaller number of features in the data to fit the model to.

3.2.4 SoftMax layer

Th neurons of the SoftMax layer produce the probability of an input belonging to a particular class, and the layer produces a series of probabilities, that all add up to one, and define the probabilities of an input belonging to each class.

$$\boldsymbol{Softmax}(\boldsymbol{x_i}) = \frac{\mathbf{exp}(\boldsymbol{z_i})}{\sum_j \mathbf{exp}(\boldsymbol{z_j})}$$

Here, the exponent is non-linear function used to convert $\boldsymbol{z_i}$, and $\boldsymbol{z_j}$ to a non-linear value.

3.3. Deep learning pathways

Deep learning models, or deep neural networks are composed of several layers made up of nodes that use "weights" and "biases" to function. Traditional learning involves the training of each and every layer of the neural network, i.e., the weights and biases of each neuron in every layer can be adjusted during training through backpropagation. Transfer learning is built on the premise of reusing pretrained models, i.e., the weights and biases of certain layers of the model may not be adjusted during training.

3.4. Model training and hyperparameters

The training of a deep learning model is highly configurable with the use of various hyperparameters.

A model's parameters, i.e., weights and biases of its neurons, define its fundamental computational metrics whereas the hyperparameters determine the training process of the model. Since the hyperparameters steer the changes in the parameter values during training to minimize loss, therefore the term hyper + parameter is used. If the set of parameters is the engine of the car, then the hyperparameters are the breaks, the steering wheel, and the gear box.

Following sections describe some useful hyperparameters used to train a deep learning model.

3.4.1 Cost function

The cost function is used to provide a measure to the contrast between the predicted label by a model for all samples over a single epoch of training and the true labels associated with the samples in order to aid in the optimization of the hypothesis. The loss function measures this contrast between the predicted value and the true value for a single sample. However, the terms cost and loss may be used interchangeably.

The categorical cross-entropy, is the most commonly used loss function in deep learning multinomial classification problems. The probabilistic predictions of the deep learning model and the true labels are used to compute this metric.

The concept of cross-entropy comes from Claude Shannon's research on information theory where he aimed at defining mathematical principles for reliable transmission of information.

let, $\boldsymbol{U}$ = Uncertainty reduction by the transfer of information indicating the occurrence of the event

$\boldsymbol{P}$ = Probability of the occurrence of the event

then,

$$\boldsymbol{U = \frac{1}{P}}$$

let, $\boldsymbol{B}$ = Bits of useful information communicated

then,

$$\boldsymbol{B = \log_2 U = \log_2 \left(\frac{1}{P}\right) = -\log_2 P}$$

Now, the entropy is simply summation of the products of each event's probability in the probability distribution with the bits of useful information used to communicate its occurrence.

let, $\boldsymbol{p}$ = Probability of the occurrence of the event

then,

$$\boldsymbol{Entropy(p) = -\sum_i p_i \log_2(p_i)}$$

Cross-entropy is calculated by computing the uncertainty reduction using the predicted probability instead of the true probability.

let, $\boldsymbol{q}$ = Predicted probability of the occurrence of the event

then,

$$\boldsymbol{Cross\ Entropy(p,q) = -\sum_i p_i \log_2(q_i)}$$

The predicted probability distribution is the output of the SoftMax layer that outputs a probability distribution or a vector of probabilities of the input datapoint belonging to each class, that all add up to one. The true probability distribution is the one-hot encoded vector associated with each input datapoint that holds the value 1 for the true class label and zero for the rest.

3.4.2 Optimization function

An optimization algorithm is aimed at minimizing the cost function by adjusting or optimizing the trainable parameters of the model's hypothesis function to effectively classify the datapoints, i.e., the aim of the optimization function is to perform function approximation, of the hypothesis function, using function optimization. The cost function is differentiable with respect to the trainable parameters. This property is used to iteratively minimize each parameter. This process of iterative minimization is called gradient descent.

In gradient descent, the rate of change of the cost function with respect to each trainable parameter is calculated. This value is then multiplied with the learning rate hyperparameter and the resultant is subtracted from the original parameter to produce the new updated parameter.

Assume, that a model has two trainable parameters, represented by $\boldsymbol{W}$ and $\boldsymbol{B}$.

then,

$$\boldsymbol{Gradient, G} = \begin{cases} \dfrac{\partial \boldsymbol{J}}{\partial \boldsymbol{W}}, & \boldsymbol{for\ parameter\ W} \\ \dfrac{\partial \boldsymbol{J}}{\partial \boldsymbol{B}}, & \boldsymbol{for\ parameter\ B} \end{cases}$$

where, $\boldsymbol{J}$ = cost function

A partial derivative of $\boldsymbol{J}$ with respect each parameter is taken as the other parameters are kept constant during the differentiation.

The optimization of the parameters, or gradient descent is described in the following equations.

$$\boldsymbol{W := W - \alpha G}$$
$$\boldsymbol{B := B - \alpha B}$$

Gradient descent is generally done by using either of the two forms of gradients which are, batch gradient and stochastic gradient. In batch gradient descent the weights are updated after calculating the cost over the entire dataset whereas in stochastic gradient descent the weights are updated after calculating the cost of a single datapoint, or instead a minibatch of the dataset which aids in reducing noise in training that a single datapoint would generate. The size of the minibatch, or batch size, is a hyperparameter that is provide during training. Stochastic gradient descent is naturally faster and is more commonly used in deep learning where datasets can be extremely large.

The problem with gradient descent is that there can be too much oscillation in the parameter values before the cost function is minimized. In order to dampen these oscillations and therefore achieve a more straightforward and therefor faster approach towards the minima, or the minimum cost function value, additional computations can be performed before updating the parameters using the gradient. Algorithms such momentum algorithm, RMSprop, Adam optimizer are examples of algorithms that are built upon the idea of stochastic gradient descent.

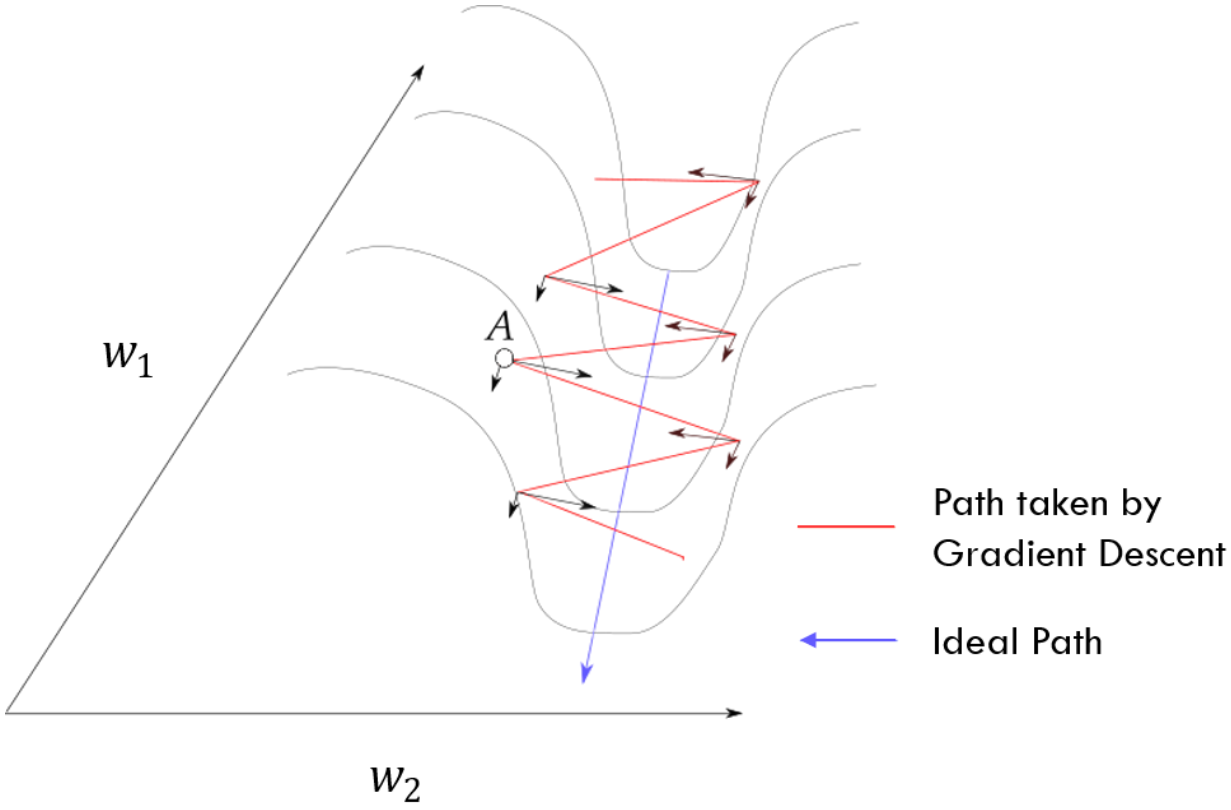

Fig 13: Path of the gradient descent
(Source: Ayoosh Kathuria)

An improved version of stochastic gradient descent called momentum algorithm archives this by accumulating each gradient value at each iteration instead of discarding the gradients and calculating a new gradient at each iteration. The gradients are accumulated using an accumulator, or velocity function. This helps in minimizing the cost function much faster than vanilla SGD, or original, stochastic gradient descent. An exponentially weighted moving average is calculated iteratively by accumulating previous gradients, giving higher weightage to the newer gradients over the older accumulated gradients.

let, $\boldsymbol{V_t}$ = exponentially weighted moving average at time t

$\boldsymbol{V_{t-1}}$ = exponentially weighted moving average at time $t-1$

$\boldsymbol{\beta}$ = hyperparameter, generally taken as 0.9 in deep learning as it produces better results

$$V_t = \beta V_{t-1} + (1 - \beta) G$$

The exponentially weighted average will be calculated for each parameter, just like the gradient. The V or velocity metric is then used to compute new parameters.

$$W := W - \alpha V_W$$
$$B := W - \alpha V_B$$

Adam and RMSprop are the most widely accepted and used optimization algorithms. In these algorithms faster training is achieved using certain additional computations.

The following equations describe the calculation of updated weights using RMSprop.

$$V_{W_t} = \beta V_{W_{t-1}} + (1 - \beta) G^2$$
$$V_{B_t} = \beta V_{B_{t-1}} + (1 - \beta) G^2$$

$$W := W - \frac{\alpha}{\sqrt{V_W + \varepsilon}} G$$
$$B := B - \frac{\alpha}{\sqrt{V_B + \varepsilon}} G$$

where, ε = constant used to ensure that the gradient descent does not overshoot in the case where V approaches 0.0

Adam, or Adaptive moment optimization is formed by the combination of momentum algorithm and RMSprop and combines the advantages of the two algorithms. This algorithm is most commonly used in practice for many applications of deep learning.

The following equations describe the calculation of updated weights using Adam.

$$V_{W_t} = \beta_1 V_{W_{t-1}} + (1 - \beta_1) G$$

$$V_{B_t} = \beta_1 V_{B_{t-1}} + (1-\beta_1) G$$
$$S_{W_t} = \beta_2 S_{W_{t-1}} + (1-\beta_2) G$$
$$S_{B_t} = \beta_2 S_{B_{t-1}} + (1-\beta_2) G$$

$$W := W - \alpha \frac{V_W}{\sqrt{S_W + \varepsilon}} G$$
$$B := B - \alpha \frac{V_B}{\sqrt{S_B + \varepsilon}} G$$

3.4.3 Learning rate, batch size and epochs

As the name suggests, the learning rate determines the rate at which the loss function of the model is minimized. While in several fundamental applications of deep learning a model may be trained on constant learning rate, it is experimentally understood that an adaptive learning rate is generally more useful in minimizing the cost function. A constant learning rate may never converge upon the minima and may instead hover around the minima by continuously and steadily overshooting it. In order to prevent this, the learning rate of every link of a neuron can be adjusted and therefore the rate at which its associated weight is updated can be appropriately controlled.
If the gradient gets reversed by the weight over a few iterations then the learning rate is reduced. Similarly, if the gradient maintains consistency, then the learning rate is increased. This change in the learning rate is determined mathematically using learning rate decay.

let, α_0 = initial learning rate hyperparameter
α = the new learning rate
m = the epoch at the time of learning rate update during training
β = decay rate hyperparameter

$$\alpha = \frac{1}{1+\beta m} \alpha_0$$

Here, the learning rate is inversely proportional to the epoch number the model is training in at the time of learning rate update, therefore the learning rate decreases successively after each epoch.

The learning rate can be updated by several other methods as well, such as decreasing the learning rate when the cost calculated on the validation dataset does not decrease over a number of epochs during the training of a deep learning model.

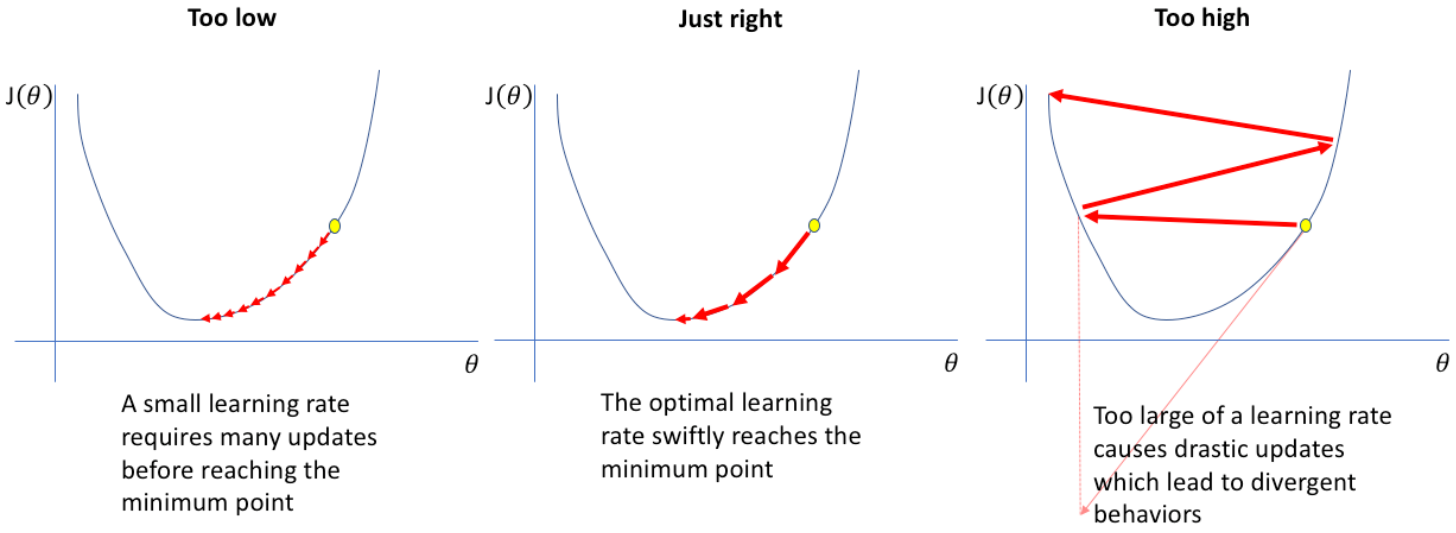

Fig 14: Gradient descent with variations in learning rate
(Source: Jeremy Jordan)

The batch size the defines the number of samples a model iterates through before calculating the error gradient. Batch size may be varied during the training as in AdaBatch.

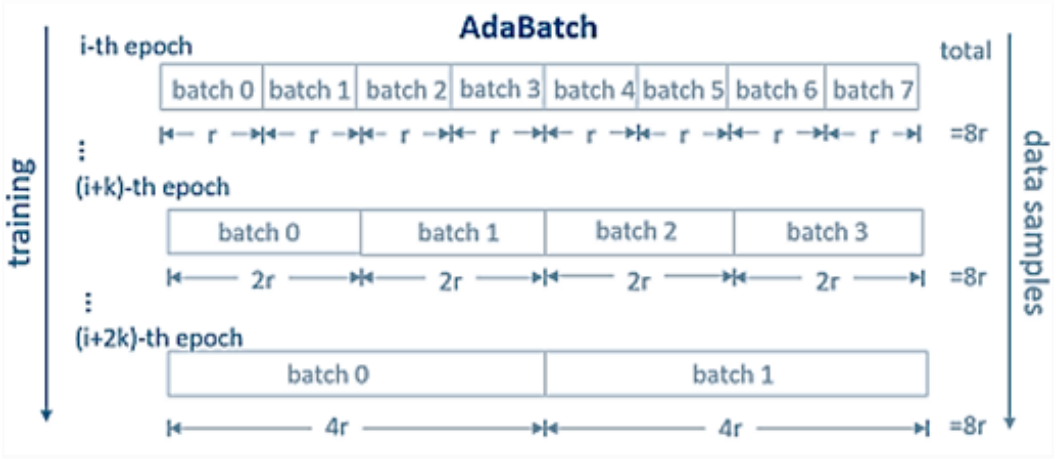

Fig 15: Altering batch size during training over epochs
(Source: Nvidia)

This number defines the times a deep learning model is trained on the entire dataset. Generally, a neural network is trained on multiple epochs since a single epoch is not sufficient to optimize a deep learning model. Although, training a model on too many epochs can result in overfitting of the model to the training set which can be observed from the contrast between accuracy, or loss calculated on the training set and validation set.

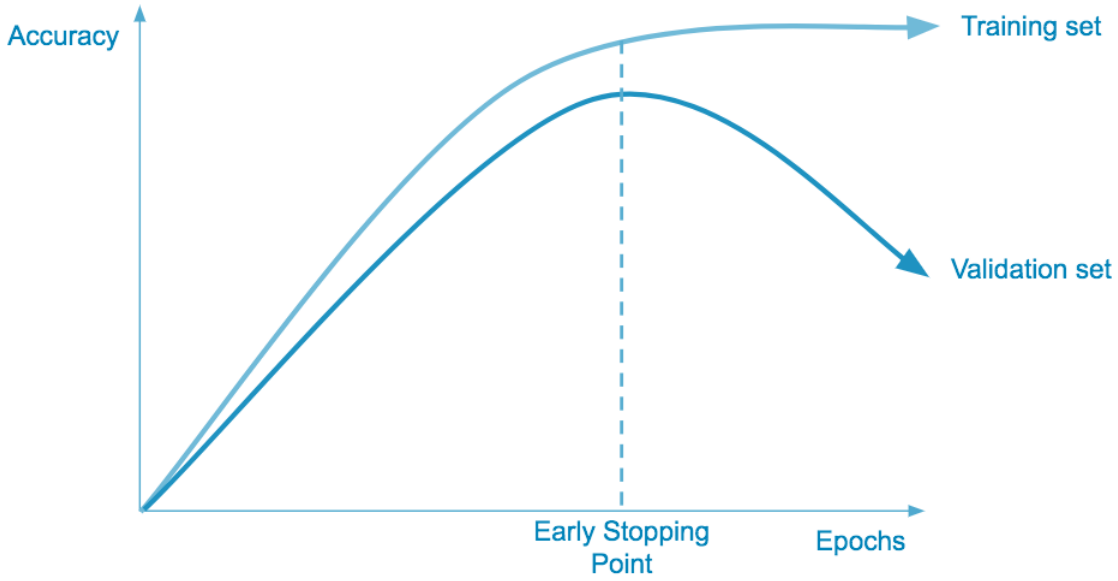

Fig 16: Relation between number of epochs and accuracy

3.4.4 Batch Normalization

Normalization is a strategy used in generic machine learning models such as linear regression where the mean and variance of the dataset are used to scale each datapoint so that the normalized dataset has a zero mean and standard deviation of one. This is done in order to establish a numerical stability in the dataset and thus make the optimization of the hypothesis easier.

In the context of a deep learning model where the inputs of each layer are the activated outputs of the previous layer, the activated outputs, or the original outputs can be normalized to ensure that the adjustment of the parameters of the next layer is more efficient.

let, $\boldsymbol{n}$ = total number of outputs

$\boldsymbol{z_i}$ = i^{th} output of a layer with total n outputs

$\boldsymbol{\mu}$ = mean of the outputs of a layer

$\boldsymbol{\sigma^2}$ = standard deviation of the outputs of a layer

$$\boldsymbol{\mu = \frac{1}{n}\sum_{i=1}^{n} z_i}$$

$$\boldsymbol{\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(z_i - \mu)^2}$$

$$\boldsymbol{z_i^{norm} = \frac{z_i - \mu}{\sqrt{(\sigma^2 + \varepsilon)}}}$$

In the case of deep learning, it can be useful vary the mean and variance of the outputs of the hidden layers instead of setting them to be zero and one respectively.

$$\boldsymbol{\tilde{z}_i = \gamma z_i^{norm} + \beta}$$

where, $\boldsymbol{\gamma}$ = scaling hyperparameter

$\boldsymbol{\beta}$ = shifting hyperparameter

Scaling and shifting hyperparameters are adjusted during training just like the model parameters.

After are training is completed global statistics of mean and standard deviation are used in normalization during the inference of the class label of an unlabelled datapoint.

The strategy of batch normalization was described by Sergey Ioffe, and Christian Szegedy in their research [6] where they describe the problem with un-normalized input using the concept of internal covariate shift wherein updating the parameters of initial layers can radically shift the distribution of the output of the next layers. As the inputs to the later layers keep changing radically due to the parameter adjustment in the initial layers, the adjustment of the parameters in the later layers can become a very difficult task. This can have a significant effect on model training especially in the case of a deep neural network. Batch normalization was proposed by Ioffe and Szegedy.

Chapter 4 – Transfer learning

4.1. Concept

Transfer learning enables researchers to build models with high accuracy using only a small dataset. This is done by reusing layers of base models that are tasked with classifying fundamental image attributes such as edges and shapes and adjusting the parameters of nodes of only specific layers. The base models are trained on relatively larger datasets usually made up of several millions of datapoints.

Fig 17: A representation of transfer learning

Lisa Torrey and Jude Shavlik in their manuscript "Transfer Learning" define the following conditions for the applicability of transfer learning:

1. Higher start: With the use of transfer learning in the new neural network the performance of the new model is higher that it is with traditional learning even before training.
2. Higher slope: The performance of the new model improves quickly.
3. Higher Asymptote: The new model shows better validation scores with transfer learning than it would with traditional learning.

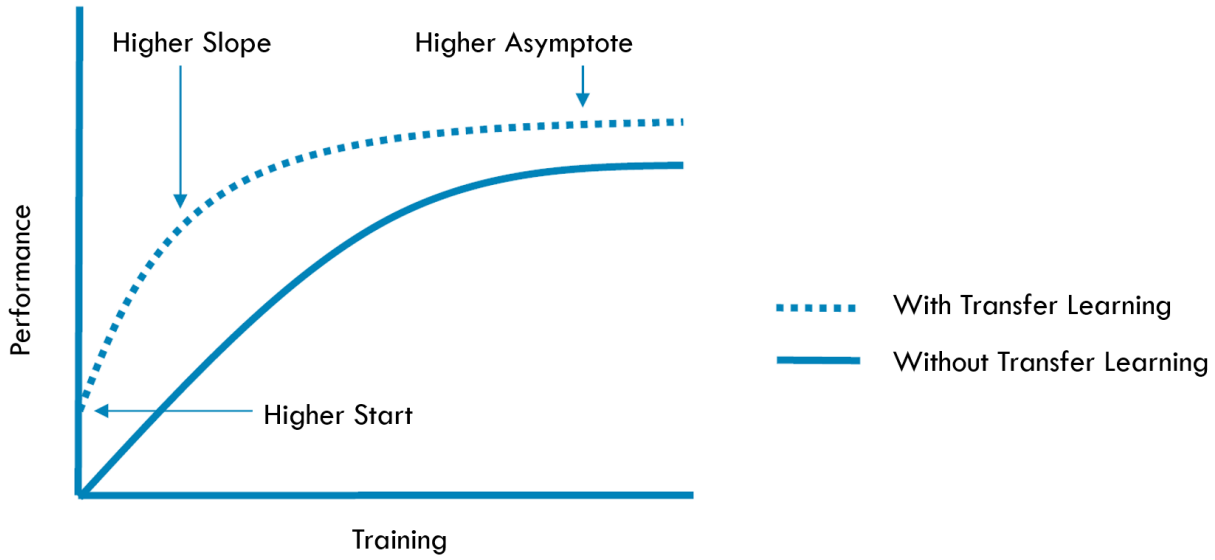

Fig 18: Relation between model training and performance
(Source: Lisa Torrey and Jude Shavlik)

4.2. Methodology

The transfer learning methodology has traditionally the following procedure:

1. Selecting a base model that has been trained to classify data similar to that of the new experimental model.
2. Removing the last SoftMax layer designed to classify the dataset of the base model.
3. Then, the base model architecture can be modified by any of the following possible actions:
    a. Additional layers can be added along with a final SoftMax layer to produce a class label. The layers of the base model are kept "frozen", i.e., there parameters are not adjusted during training.
    a. A number of layers of the base model can be unfrozen, i.e., there parameters are adjusted during training, while the parameters of the remaining layers are not. A final SoftMax layer to produce a class label is also added.
    b. New layers can be added and some layers of the base model can be retrained at the same time. Again, a final SoftMax layer is added to produce a class label.
4. The newly constructed neural network is trained on a relatively smaller dataset according to the set hyperparameters.
5. The new model is fine-tuned until a desirable model is produced.

4.3. Fine tuning

The utility of transfer learning is in reusing the information of a base model, i.e., the parameters of certain layers to classify the fundamental features of the dataset. Yet, it has been observed that after the new model has been trained on the new relatively small dataset while keeping the base model's layers frozen, better accuracy can be achieved by unfreezing some or all of the layers of the base model and retraining the model on the small dataset.

Fine tuning is a crucial step in transfer learning paradigm with the goal of transforming the new model to better classify the inputs from the new and smaller dataset.

4.4. Experiment Design and Results

The performance of some deep learning models in classifying malware samples using transfer learning has been observed in this study till this second phase of this project and more models will be studied in the next and final phase of this project. The MMCC dataset has been used to train the models.

An Anaconda environment on a local machine and Google Collaboratory have been used to perform the experiments in this project.

The configuration of the local machine is described below.

System Component	Configuration
Central Processing Unit (CPU)	11th Gen Intel i5 @ 2.40GHz
Random Access Memory (RAM)	8.00 GB
Graphics Processing Unit (GPU)	Intel Iris Xe Graphics

Table 2: System Configuration

The configuration of the Google Collaboratory is described below.

System Component	Configuration
Central Processing Unit (CPU)	Google cloud-based CPU
Random Access Memory (RAM)	12 GB
Graphics Processing Unit	GPU, TPU

Table 3: Google Collaboratory Configuration

Pre-trained models used in this experiment are described in the following sections.

This model was introduced by Howarda, et al. [6] and its architecture is described in table 4. Here, dw denotes depthwise filter.

Type / Stride	Filter Shape	Input Size
Conv / s2	3 X 3 X 3 X 32	224 X 224 X 3
Conv dw / s1	3 X 3 X 32 dw	112 X 112 X 32
Conv / s1	1 X 1 X 32 X 64	112 X 112 X 32
Conv dw / s2	3 X 3 X 64 dw	112 X 112 X 64
Conv / s1	1 X 1 X 64 X 128	56 X 56 X 64
Conv dw / s1	3 X 3 X 128 dw	56 X 56 X 128
Conv / s1	1 X 1 X 128 X 128	56 X 56 X 128
Conv dw / s2	3 X 3 X 128 dw	56 X 56 X 128
Conv / s1	1 X 1 X 128 X 256	56 X 56 X 128
Conv dw / s1	3 X 3 X 256 dw	28 X 28 X 128
Conv / s1	1 X 1 X 256 X 256	28 X 28 X 256
Conv dw / s2	3 X 3 X 256 dw	28 X 28 X 256
Conv / s1	1 X 1 X 256 X 512	14 X 14 X 256
5 X [Conv dw / s1	3 X 3 X 512 dw	14 X 14 X 512
Conv / s1]	1 X 1 X 512 X 512	14 X 14 X 512

Conv dw / s2	3 X 3 X 512 dw	14 X 14 X 512
Conv / s1	1 X 1 X 512 X 1024	7 X 7 X 512
Conv dw / s2	3 X 3 X 1024 dw	7 X 7 X 1024
Conv / s1	1 X 1 X 1024 X 1024	7 X 7 X 1024
Avg Pool / s1	Pool 7 X 7	7 X 7 X 1024
FC / s1	1024 X 1000	1 X 1 X 1024
SoftMax / s1	Classifier	1 X 1 X 1024

Table 4: MobileNet architecture
(Source: Howarda, et al.)

Other pre-trained models used in this study until this phase are MobileNetV2, and DenseNet169. In all the experiments learning rate is reduced whenever the loss value on the validation set does not improve after certain epochs.

The custom layers added to pre-trained base model after removing the base model's SoftMax layer are described in the table 5.

Layer	Output shape	Parameters (Trainable)
flat1 (Flatten)	(None, 100352)	0
dense (Dense)	(None, 512)	51380736
dropout (Dropout)	(None, 512)	0
batch_normalization_4 (Batch Normalization)	(None, 512)	2048
dense_1 (Dense)	(None, 128)	65664
drop3 (Dropout)	(None, 128)	0
dense_2 (Dense)	(None, 64)	8256
dropout_1 (Dropout)	(None, 64)	0
fc3 (Dense)	(None, 9)	585

Table 5: Trainable layers added to the pre-trained model

All the models trained using the hyperparameters and attributes described in the table 6.

Hyperparameter/Attribute	Configuration
Optimizer	Adam
Loss	Categorical Cross Entropy
Initial learning rate	0.001
Adam optimizer, β_1	0.9
Adam optimizer, β_2	0.999
Metric	Accuracy
Epochs	20-30
Train-Validation-Test size	Train: 70%, Validation: 14%, Test: 16%
Input image size	224 *by* 224

Table 6: Hyperparameters and training attributes

4.4.1 Training Model 1, Base model: MobileNet

Total params	72,318,769
Trainable params	51,456,265
Non-trainable params	20,862,504

Table 7: Model 1 trainable and non-trainable parameters

Epoch	Accuracy	Loss	Learning rate	Validation accuracy	Validation loss
0	0.815959	0.779812	0.001	0.92955	0.232106
1	0.911923	0.355813	0.001	0.951076	0.183201
2	0.924149	0.284498	0.001	0.958252	0.137138
3	0.94413	0.245387	0.00095	0.960209	0.161043
4	0.944656	0.21589	0.00095	0.966732	0.133127
5	0.954778	0.17551	0.000903	0.965427	0.166419
6	0.962666	0.133592	0.000903	0.969341	0.187727
7	0.966084	0.144226	0.000857	0.969341	0.231221

8	0.969502	0.128497	0.000857	0.975212	0.101296
9	0.968056	0.116058	0.000815	0.979778	0.0869
10	0.967004	0.120714	0.000815	0.975212	0.104481
11	0.973314	0.08944	0.000774	0.981083	0.093205
12	0.978572	0.070659	0.000774	0.984344	0.087574
13	0.977389	0.088105	0.000735	0.976517	0.116228
14	0.979624	0.064098	0.000735	0.980431	0.108398
15	0.980807	0.064168	0.000698	0.977169	0.136967
16	0.982648	0.051212	0.000698	0.982387	0.218252
17	0.983173	0.06136	0.000663	0.979126	0.237589
18	0.981464	0.056848	0.000663	0.979126	0.258996
19	0.981859	0.068118	0.000663	0.977821	0.104815

Table 8: Training of model with MobileNet as the base model

4.4.2 Training Model 2, Base model: MobileNetV2

Total params	34,447,689
Trainable params	32,188,681
Non-trainable params	2,259,008

Table 9: Model 2 trainable and non-trainable parameters

Epoch	Accuracy	Loss	Learning rate	Validation accuracy	Validation loss
0	0.786644	0.862669	0.001	0.934768	0.234456
1	0.889707	0.469425	0.001	0.949119	0.181268
2	0.909951	0.344396	0.001	0.947162	0.178628
3	0.921651	0.298174	0.00095	0.953033	0.171935
4	0.934534	0.25114	0.00095	0.958904	0.158687
5	0.941764	0.210776	0.000903	0.964775	0.133804
6	0.953332	0.150714	0.000903	0.964123	0.125716
7	0.957539	0.138426	0.000857	0.960861	0.156413

8	0.962403	0.12688	0.000857	0.969993	0.112201
9	0.96214	0.127747	0.000815	0.969993	0.115311
10	0.965032	0.119901	0.000815	0.966732	0.133711
11	0.962666	0.147236	0.000774	0.971298	0.110964
12	0.960563	0.151383	0.000774	0.972603	0.107977
13	0.967004	0.113691	0.000735	0.971298	0.114174
14	0.970291	0.102435	0.000735	0.972603	0.116963
15	0.975154	0.08885	0.000698	0.969993	0.128018
16	0.972525	0.101858	0.000698	0.966732	0.119688
17	0.970816	0.095911	0.000663	0.977169	0.100027
18	0.975154	0.091426	0.000663	0.977821	0.114197
19	0.974892	0.075426	0.00063	0.978474	0.099199

Table 10: Training of model with MobileNetV2 as the base model

4.4.3 Training Model 3, Base model: DenseNet169

Total params	54,466,377
Trainable params	41,822,473
Non-trainable params	12,643,904

Table 11: Model 3 trainable and non-trainable parameters

Epoch	Accuracy	Loss	Learning rate	Validation accuracy	Validation loss
0	0.754568	0.965269	0.001	0.92107	0.254301
1	0.88366	0.491527	0.001	0.944553	0.187238
2	0.911134	0.382557	0.001	0.943249	0.19239
3	0.906665	0.368756	0.00095	0.951076	0.171712
4	0.921125	0.302178	0.00095	0.948467	0.168505
5	0.925726	0.279684	0.000903	0.953686	0.150245
6	0.931905	0.26511	0.000903	0.962818	0.129668
7	0.94413	0.212745	0.000857	0.958904	0.130889

8	0.94045	0.211903	0.000857	0.969993	0.117183
9	0.949389	0.171024	0.000815	0.96347	0.11618
10	0.954121	0.149138	0.000815	0.970646	0.110141
11	0.95399	0.152365	0.000774	0.96608	0.133428
12	0.958722	0.149538	0.000774	0.96608	0.131985
13	0.962272	0.120351	0.000735	0.968689	0.13313

Table 12: Training of model with DenseNet169 as the base model

Retraining the model from the weights of epoch 11 of the previous training. The learning rate is again initialized at 0.001.

Epoch	Accuracy	Loss	Learning rate	Validation accuracy	Validation loss
0	0.950177	0.190318	0.001	0.949119	0.158339
1	0.945708	0.185338	0.001	0.967384	0.120441
2	0.948206	0.185112	0.001	0.956295	0.155672
3	0.950572	0.180605	0.00095	0.961513	0.15861
4	0.954384	0.159349	0.00095	0.962818	0.149609
5	0.955962	0.178174	0.00095	0.972603	0.132847
6	0.964769	0.107684	0.000903	0.97195	0.109496
7	0.961483	0.130725	0.000903	0.968689	0.125478
8	0.957802	0.133415	0.000857	0.968037	0.152142
9	0.961088	0.127863	0.000857	0.968689	0.11122
10	0.966741	0.111242	0.000815	0.964775	0.170063
11	0.967793	0.116159	0.000815	0.970646	0.162567
12	0.969633	0.0892	0.000815	0.975864	0.129939
13	0.967924	0.108987	0.000774	0.97195	0.133497
14	0.973971	0.082567	0.000774	0.977821	0.130433
15	0.973446	0.095965	0.000735	0.975212	0.137714
16	0.978835	0.073061	0.000735	0.973255	0.148748

Table 13: Retraining of model with DenseNet169 as the base model

4.4.4 Performance of the models

Model	Base Model	Accuracy	Loss	Learning rate	Validation accuracy	Validation loss
1	MobileNet	0.967004	0.120714	0.000815	0.975212	0.104481
2	**MobileNetV2**	**0.974892**	**0.075426**	**0.00063**	**0.978474**	**0.099199**
3	DenseNet169	0.961483	0.130725	0.000903	0.968689	0.125478

Table 14: Comparison of the performance of the models

From this experiment we can conclude that MobileNetV2 shows best performance compared to the other models studied in this experiment.

Chapter 5 – Tools and Technologies

Hardware Requirement:

1. CPU : Intel core i5 8th generation or better
2. GPU : Preferred
3. RAM : 12 GB or better

Software Requirements:

1. Anaconda computer program including all essential machine learning tools.

Tools Used:

1. Anaconda: A python and R distribution which is primarily used for data science and machine learning applications. The primary objective of this tool is to make package management simple. It includes data-science packages adapted to Windows, Linux and macOS.

2. Keras: An open-source python library works as interface for the TensorFlow python library. Keras provides high level APIs for machine learning.

3. JupyterLab: An interactive environment which helps in integrating code, data and notebook in a single interface.

4. Google Collaboratory: Online cloud-based environment to execute python code, especially suited for machine learning applications.

Chapter 6 – Summary & Project Pathway

The project on malware detection using machine learning is divided into three phases as described in fig 21.

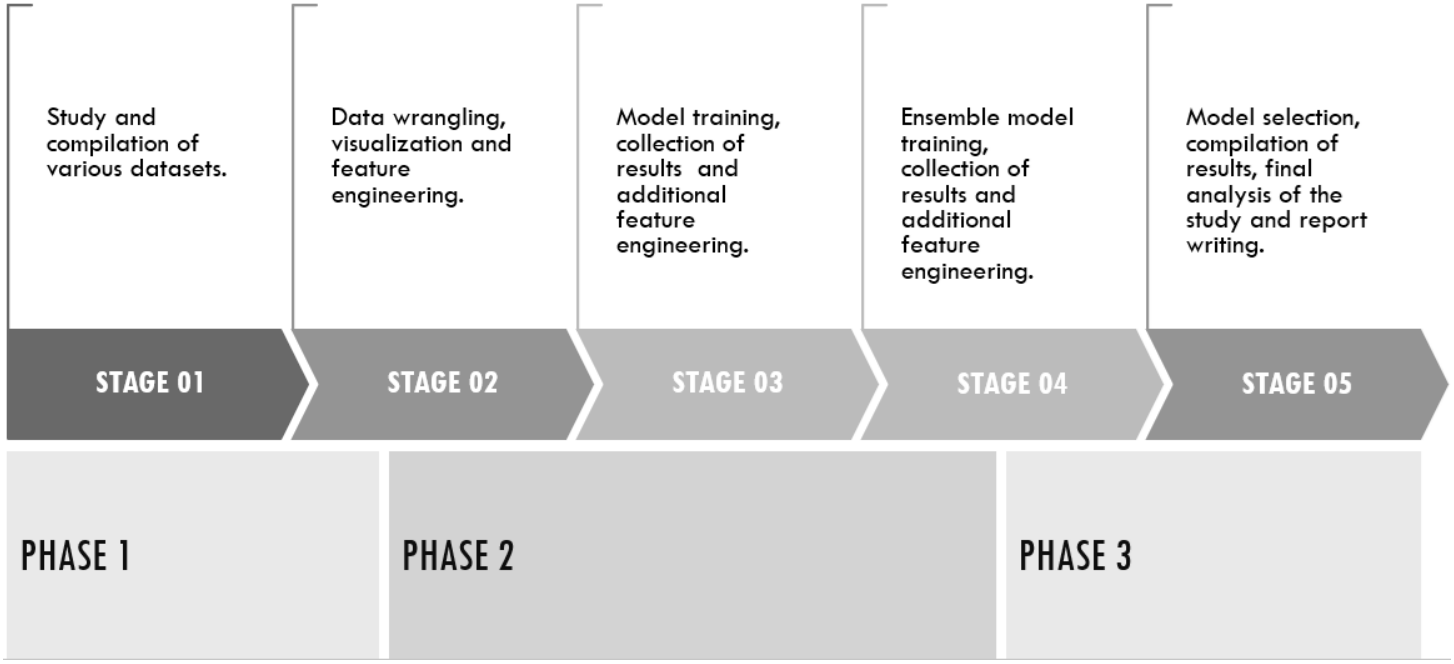

Fig 21: Project pathway

Bibliography

[1] Microsoft, "Microsoft Malware Classification Challenge (BIG 2015)," Microsoft, 2015. [Online]. Available: http://arxiv.org/abs/1802.10135.

[2] Sophos, "Sophos-ReversingLabs (SOREL) 20 Million sample malware dataset," Sophos-ReversingLabs, 14 December 2020. [Online]. Available: https://ai.sophos.com/2020/12/14/sophos-reversinglabs-sorel-20-million-sample-malware-dataset/.

[3] L. &. K. S. &. J. G. &. M. B. Nataraj, *Malware Images: Visualization and Automatic Classification,* 2011.

[4] H. Mallet, "Malware Classification using Convolutional Neural Networks — Step by Step Tutorial," 27 May 2020. [Online]. Available: https://towardsdatascience.com/malware-classification-using-convolutional-neural-networks-step-by-step-tutorial-a3e8d97122f.

[5] N. Bagga, "Measuring the Effectiveness of Generic Malware Models," 2017.

[6] S. I. Sergey Ioffe, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," 2015.

[7] m. fhowarda, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," *Computer Vision and Pattern Recognition,* p. 9, 2017.

[8] D. Braue, "Global Ransomware Damage Costs Predicted To Exceed $265 Billion By 2031," 3 June 2021. [Online]. Available: https://cybersecurityventures.com/global-ransomware-damage-costs-predicted-to-reach-250-billion-usd-by-2031/.

[9] Wikipedia, "Portable Executable," [Online]. Available: https://en.wikipedia.org/wiki/Portable_Executable.

[10] businesswire, "Global Malware Analysis Market Expected to Grow with a CAGR of 31% During the Forecast Period, 2019-2024 - ResearchAndMarkets.com," businesswire, 13 December 2019. [Online]. Available: https://www.businesswire.com/news/home/20191213005123/en/Global-Malware-Analysis-Market-Expected-to-Grow-with-a-CAGR-of-31-During-the-Forecast-Period-2019-2024---ResearchAndMarkets.com.

[11] kaspersky, "Machine Learning for Malware Detection," [Online]. Available: https://media.kaspersky.com/en/enterprise-security/Kaspersky-Lab-Whitepaper-Machine-Learning.pdf.

List of tables