TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Dismas Ndubuisi Ezechukwu 184603IVEM

Capsule Neural Network and its Implementation for Object Recognition in Resource-limited Devices

Master's thesis

Supervisor:	Yannick Le Moullec PhD
Co-Supervisors:	Muhammad Mahtab Alam PhD
	Abdul Mujeeb MSc

Tallinn 2020

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Dismas Ndubuisi Ezechukwu 184603IVEM

CAPSULE-TÜÜPI NÄRVIVÕRK NING SELLE RAKENDAMINE OBJEKTIDE TU-VASTAMISEKS PIIRATUD RESSURSSIDEGA SEADMETES

Magistritöö

Juhendaja:	Yannick Le Moullec PhD
Kaasjuhendajad	Muhammad Mahtab Alam PhD
	Abdul Mujeeb MSc

Tallinn 2020

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Dismas Ndubuisi Ezechukwu

04.01.2021

[illegible]

Abstract

Image recognition is an important part of computer vision and it can be achieved with deep learning techniques such as Convolutional Neural Network (CNN), which often requires a lot of datasets in order to train the network to achieve good performance. So, in cases where only small amounts of datasets are available or when the cost of acquiring large amounts of datasets is high, alternatives should be considered. This thesis explores the use of the novel Capsule Neural Network (CapsNet) on small datasets for image recognition and the possibilities of implementing it on an embedded or edge device.

A literature review on machine learning and deep learning concepts is presented as a preamble to understanding the CapsNet concepts presented afterwards. Image processing and model compression required for deep learning deployment at the edge are also presented. Then, a CapsNet model is designed and trained, exploring the effects of various hyperparameters on its performance. A baseline CNN model is also prepared for comparison purposes.

With a two-class dataset of 30 images per class, our CapsNet model is able to give both training and validation accuracies of 70% after training on 35 epochs. But the model begins to overfit if trained beyond this number of epochs. On the other hand, an equivalent CNN model already overfits before 35 epochs, having a training and validation accuracies of 99% and 66.70%, respectively. This means that with small amounts of datasets, a CNN will overfit rapidly regardless of the number of epochs; on the other hand, we can design a CapsNet model that performs considerably better on small numbers of datasets, if properly tuned.

We also showed in this thesis that a CapsNet model can be made to run on an embedded or edge device after undergoing the necessary compressions aimed at reducing its size and complexity. All in all, this work highlights the value of CapsNets on small amounts of datasets and is deployment at the edge.

This thesis is written in English language and it is 103 pages long, including 6 chapters, 45 figures and 13 tables.

Annotatsioon

Capsule neural network ning selle rakandamine süsteemi, objekti tuvastamise jaoks.

Kujutise tuvastamine on oluline osa masinnägemisest ja see saavutatakse selliste sügavate õppimistehnikate abil nagu Konvolutsiooniline närvivõrk (CNN), mis Sageli nõuab palju andmekogumeid võrgu hea jõudluse saavutamiseks. Niisiis, kui on saadaval ainult väike kogus andmekogumeid või kui suure hulga andmekogumi hankimise kulud on suured, ei soovitata CNN-i antud juhul kasutada. Selles lõputöös uuritakse uudse kapselnärvivõrgu (CapsNet) kasutamist väikestes andmekogumites piltide tuvastamiseks ja selle rakendamise võimalusi manustatud või servaseadmel.

Masinõppe ja sügava õppe kontseptsioonide kirjanduse ülevaade esitati Sissejuhatusena mõistmaks hiljem esitatud CapsNeti kontseptsioon. Samuti esitleti pildi töötlemist ja mudeli tihendamist, mis on vajalik sügava õppimise juurutamiseks servas. Seejärel töötati välja ja Treeniti CapsNet ja CNN-i baasmudel, uurides erinevate hüperparameetrite mõju selle toimivusele.

Kahe klassi andmekogumiga, mis sisaldab 30 pilti klassi kohta, suutis meie CapsNeti mudel anda 35 ~~koolitusjärgse~~ treeningu epohhi järel nii koolituse kui ka valideerimise täpsuse 70%. Kuid see hakkab üle sobima, kui seda epohhe ületada. Teiselt poolt oli samaväärne CNN juba 35 epohhil üle mahtunud, kelle koolituse ja valideerimise täpsus on vastavalt 99% ja 66,70%. See tähendab, et väikese koguse andmekogumi korral läheb CNN üle, olenemata epohhide hulgast, kuid saame CapsNeti, mis töötab korralikult häälestatuna tunduvalt paremini.

Samuti näitasime selles uuringus, et CapsNeti mudeli saab panna tööle sisseehitatud või servaseadmega pärast vajalike kompressioonide läbimist, mille eesmärk on vähendada selle suurust ja keerukust.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 103 leheküljel,6 peatükki, 45 joonist, 13 tabelit.

List of abbreviations and terms

ANN	Artificial Neural Networks
CapsNet	Capsule Neural Networks
CNN	Convolutional Neural Networks
COCO	Common Object in Context
DT	Decision Tree
FC	Fully Connected
GPU	Graphic Processing Unit
HCS	Hyperparameter Configurations Structure
HOG	Histogram of Oriented Gradients
KNN	K-Nearest Neighbour
MAE	Mean Absolute Error
MBE	Mean Bias Error
ML	Machine Learning
MLP	Multilayer Perceptron
MNIST	Mixed National Institute for Standards and Technology
MSE	Mean Squared Error
MTBF	Mean Time Before Failure
OpenCV	Open Computer Vision
PASCAL	Pattern Analysis Statistical and Computational Learning
PC	Personal Computer
POC	Point of Care
POV	Visual Object Classes
R-CNN	Region-based Convolutional Neural Network
ReLU	Rectified Linear Unit
RPN	Regional Proposal Network
SGD	Stochastic Gradient Descent
SIFT	Scale-Invariant Feature Transform
SMC	Systems Man and Cybernetics
SPP-Net	Spatial Pyramid Pooling-Neural Network
SVM	Support Vector Machine
TPU	Tensor Processing Unit
USB	Universal Serial Bus

Table of contents

List of figures

List of tables

1 Introduction

Deep learning techniques, a subclass of machine learning, have now become one of the main methods for performing various object recognition tasks. Machine learning techniques have also proven to be useful in other fields such as forensics, machine vision, robotics, drug discovery, medicine, and geographic information extraction to say but a few [1]. Various tasks such as object detection and recognition, speech recognition, image classification, object tracking, trend uncovering, and predictions for equipment failure, weather conditions, financial trading are a few examples of machine learning applications. Many applications of deep learning use artificial neural networks (ANN) with multiple layers to achieve good performance in various tasks [2].

In supervised machine (deep) learning, data, together with its label(output), is usually fed to the network so that it can “learn” these data and associate them with a particular label. When this is done with a large amount of data for a particular label, the model would have learned how to identify this kind of data and associate it with the corresponding label. The above procedure is called training. Inference, on the other hand, is when the model is only being fed with a completely new dataset, and it will have to associate a label to it based on the experience it has gained during training.

Due to the large amount of computational power required for the training of deep learning models, it is common that they are being run on a high-performance computer (mostly a remote server). It is also common for deep learning inference to be run on these high-performance computers which are mostly in a remote location (a server) to the data source, but in recent times, many deep learning and machine learning inference are being moved to the network edge as increasingly more applications are requiring in situ processing. Although devices running deep learning algorithms at the edge may not be as powerful as the processors in the servers, many factors have contributed to the migration of many machines learning algorithms processing at the edge for certain application categories or specific uses-cases. Factors such as latency, security issues, offline processing capability, energy conservation are some of the reasons driving the trend of edge/near sensor computing. Edge computing has been partially adopted in many areas such as in smart homes, point of care (POC) devices, environmental monitoring systems, smart grids and video surveillance systems [3].

Object detection and recognition-based systems have been mostly implemented through convolutional neural networks (CNN) and excellent performances have been achieved in recent years [4]. To do so, a plethora of data is usually needed which may not be available or very difficult to come by for some applications. For instance, a common problem in the Scandinavian and some countries in Northern Europe such as Estonia is the death of some wild animals such as deers and reindeers due to collisions with vehicles on the highways in the outskirt of major cities. To build a system that recognizes animals quickly in such scenarios with a CNN-based system would require a lot of image/video data for training; unfortunately, such data is scarce and difficult to acquire. Hence, it is important to explore the use of alternative algorithms that can give a good performance using a relatively small amount of data; capsule neural network (CapsNet) is such an alternative, which has shown promising performance in e.g. in identifying handwritten characters[5]. Indeed, some problems with CNNs such as information loss in its pooling layers and its translational invariant are some of the reasons for the adoption of capsule neural network [5].

1.1 Problem Statement

One of the reasons why CNN requires a large amount of data is because the pooling operation used as a routing method in CNN is fraught with problems, and as such makes CNN translational and rotational invariant, making it non-responsive to different variations of the same data sample.

On the other hand, Using CapsNet to solve this problem of enormous data requirement has many challenges as well since it requires a lot more computations and at the end produces a model with a large size which will be difficult to fit on an embedded device. The challenges associated with fitting a model on embedded devices arise from their limited memory and computational capacity.

It is worth noting that, as per the author's best knowledge, CapsNets, which could offer some solutions to the problems with CNNs, have not been tested out or deployed on an embedded device (i.e. targeting edge devices) yet.
Given the above challenges, this thesis seeks to answer the following questions:

- How to efficiently implement machine learning models on the edge instead on the cloud while making the model smaller and at the same time not compromising too much on its performance?
- How to implement an object recognition model that requires a relatively small number of data for training and also achieves good performance while applying model minimization techniques to the model?
- How to implement such a model on an embedded device such as the TPU-based google-coral-range board?

1.2 Aims and Objectives

To answer the above questions, the overall aim of this work is to build a CapsNet-based object recognition model and implement it on an embedded device. This can be achieved by carrying out the following tasks:

- Study and understand the general concepts of machine learning and the existing work in this area, particularly deep learning algorithm.
- Obtain suitable datasets of objects to be recognized and perform the necessary pre-processing needed.
- Building and training of a CapsNet model
- Build and train a baseline CNN model of similar capacity with the built CapsNet for the purpose of comparison.
- Explore and implement suitable model optimization techniques and their suitability of different embedded devices to support them.
- Implement the model on the suitable edge device and conduct proper testing of the system performance.

1.3 Methodology

Building a capsule neural network on an edge device is quite a complex task and the implementation of such is explored in this thesis work. The first step in this undertaking is to thoroughly understand the architecture and the various building blocks of a CapsNet and its mathematical implementation. Next, data is obtained, pre-processed, and then used to train a pre-built CapsNet; the functionality of the model is then assessed and deployed onto the chosen embedded device.

With the image dataset pre-processed, the CapsNet model is trained and its performance is evaluated. After this, model minimization techniques such as quantization and pruning are applied to reduce the size of the model since it will be run on an embedded device that usually have memory and processing capability limitations as compared to server-based processing machines. Since the model will be implemented on an embedded device, it is needed to be converted to a TensorFlow-Lite model and then to C++ code.

The Tensor Processing Unit (TPU)-based Google Coral range board is chosen as the embedded device of choice due to its high processing capability and low power consumption. Image datasets are obtained using a digital camera, some other images were also obtained from Kaggle (an online database) and by using a python script to convert some video files into frames. TensorFlow and Keras are the main deep learning frameworks used and many libraries such as OpenCV (Open Computer Vision) are also used in the image pre-processing section.

After the model is trained and optimized, it is deployed on the target embedded device and then the entire system is tested on new images to examine its performance after deployment and adjustments are made where necessary to optimize its performance.

1.4 Thesis organization

This thesis contains introductory knowledge and background on deep learning and particularly convolutional neural network. Capsule neural network and its architectural analysis, the implementation and deployment details on an embedded device are explored afterwards. This chapter (Chapter 1) provided an introduction to the object detection algorithms, research statement and the intended aim of the thesis.

In Chapter 2, we delve further into the state-of-the-art related to image recognition and explore existing works in this area. A general background of the concept of machine learning, particularly CNN is explored; some techniques used for model minimization are also examined in this chapter.

The drawback of CNN is briefly discussed in Chapter 3, while also introducing CapsNet as an alternative to CNN. The general architectural components of CapsNet and the routing algorithm are also discussed in this chapter.

In Chapter 4, we detail the methodology and implementation of CapsNet, showing its overview, the image processing performed, the model conversion methods, the training procedure, the software libraries and framework used, the key hyperparameters of the model and the deployment strategy on an embedded device. The results of these processes and their analyses are detailed in Chapter 5.

Finally, the last chapter gives a conclusive discussion about the work and suggest ideas for future work that could be carried out based on the author's recommendations.

2 Background

In this chapter, a background on the machine learning and CNN is presented, starting with a state-of-the-art in object recognition and followed by subsections on machine learning, CNN, edge computing, and model minimization.

2.1 State- of -the-Art in Image recognition

The use of a computer for the recognition of an image (or an object in an image) has been a focus of research for a long time, and still is. The reason for this can be attributed to its use in many facets of life [6]. It has found modern application in manufacturing, agriculture, automotive vehicles and military surveillance to name a few. Traditional image recognition methods that are based on manual underlying and high-level features which can properly characterize an image have been used in the past. Such features are SIFT (scale-invariant feature transform), HOG (Histogram of Oriented Gradients) etc.; however, it is somewhat difficult to identify an image in a complex scene because these traditional methods require designing of different features for every image recognition problem [7]. So, in recent years, the advancement in deep learning has shifted the focus to the use of machine learning techniques. Advancement, particularly in CNN, is a major attributing factor to this success in recent years and it is important to realise that although deep learning (or machine learning in general) have been around for a long time, it has never been popularized and used the way it is been applied since the beginning of this decade. There are two major reasons for this proliferation. The first reason is the availability of vast amounts of data because of the increase in global internet penetration. The second reason is the advancement in computer technologies in terms of hardware speed and massive processing capability combined with development of tools that made them easier [8]. Of a particular interest is the use of deep learning for object recognition is CNN because it has been the deep learning workhorse for image recognition as it has performed better than all other algorithms for this particular task [2]. In the simplest sense, image recognition with deep learning is done by following steps: (i) data (or image) acquisition, (ii) preparation or pre-processing of the dataset, (iii) definition of the training algorithm, (iv) training of the model, and finally (v) testing and deployment [9]. Of particular importance and probably the most important step in this is the acquisition of the dataset itself as the algorithm cannot perform better than the dataset it was trained with. The work of LeCun [8] in 1989 on processing grid-like topological data brought CNN into limelight. CNN, which consists of artificial neurons arranged in layers, learns through an algorithm known as backpropagation; here, an image is passed through a CNN network (known as the forward pass), the low level features of the image is extracted with the use of convolutional layers, which is propagated forward for further higher level feature extraction. At the output, the loss function (a metric used to indicate the deviation of an estimated output from the actual output) is calculated to know the accuracy of the network and after that comes the backpropagation. During the backward pass (backpropagation), the weights in the network get updated or adjusted so as to increase the accuracy the next time the image is passed through the network. This whole process known as training will continues back and forth until desired level of accuracy is achieved.

There has been several CNN architectures designed that have achieved different levels of accuracy. For example, R-CNN (Region-based Convolutional Neural Network) was applied to a candidate box to extract feature vectors and it was trained on the ImageNet international computer Vision Challenge (ILSVRC) and Pattern Analysis, Statistical modelling and Computational learning Visual Object Classes (PASCAL VOC) dataset [6]. Later came SPP-Net (Spatial Pyramid Pooling-Neural Network) which is an improvement on the R-CNN; it uses spatial pyramid pooling to eliminate the constant-size network constraint specifically, and the SPP layer is applied on top of the last convolutional layer [10]. Trained on the PASCAL VOC dataset, SPP-Net is 30-170 times faster than R-CNN. The fast R-CNN took care of the shortcoming of R-CNN and SPP-Net [11]. by improving on detection quality and having the simultaneous loss function for multiple tasks to achieve single-level training process[6]. During training, fast R-CNN is 9 times and 3 times faster than R-CNN and SPP-Net, respectively. And during testing, that number is 213 and 10 than for R-CNN and SPP-Net, respectively. The Fast R-CNN and SPP-Net were made even faster (approximately 12 times faster) by the introduction of Faster R-CNN [12]. It has two components which are a fully convolutional Region Proposal Network (RPN) for proposing candidate regions, followed by Fast R-CNN [13].

As pointed out before, a major reason for the ubiquity of deep learning for image recognition is because of the vast amount of data collected used for training. Some for the widely used dataset is the ImageNet dataset [14]; it consists of more than 14 million images of more than 200,000 categories and has been used in many image classification research works and a popular challenge knows as ILSVRC is based on it [15]. PASCAL VOC which has images in 20 classes is another popular dataset used in image recognition research [16]. Another common dataset is known as COCO (Common Object in Context) dataset which is used in image segmentation, captioning and recognition [17]. Sponsored by Microsoft, the COCO dataset has about 300,000 images encompassing 80 object classes.

One common denominator of all image classification schemes based on CNN is their proclivity for training using very large number of data; this is not necessarily a drawback, but it renders them useless in situations where there are not just many data. A reason for this is because CNN loses the spatial relationship between different parts in an image during classification partly due to the use of the pooling operation and has hence having viewpoint and scale variation. For this reason, CNN will have to be trained with a mammoth of data to achieve an accurate prediction. To solve this problem, CapsNet has been developed by Geofrey Hinton and have been applied to the National Institute of Standards and Technology (MNIST) dataset and it has shown state-of-the-art performance and considerable better than CNN on recognizing overlapping digits [5]. The MNIST dataset (grayscale, 28x28 pixel resolution handwritten characters) consist of 10 classes and 70,000 images [18]. However, CapsNet will have an accuracy lower than that of CNN when it comes to using larger datasets and there is still much room for improvement and some of this will be explored in this work. In the sections that follows, some basic principles of machine learning and deep learning that are necessary for the understanding of this work are discussed.

2.2 Machine Learning

Commonly, to make a system perform a task, it is given a set of explicit instructions on how to perform them. Instead, in machine learning, the system learns how to perform a task from experience. Here, learning to perform a task by the system is done by observing a series of examples (training), the system then performs the task on a data that it has not worked on before (testing and inference). Although machine learning is not new, the increase in computing power and a vast amount of data being generated has made machine learning more relevant now than ever in solving many complex problems in recent times [19]. To fully put this thesis work into perspective, it is pertinent to explore some traditional machine learning (ML) component and algorithms as this will give some basis to the terms that will be used throughout this work. Hence, in the sub-sections to follow, the machine learning fundamentals necessary to fully grasp this work are presented.

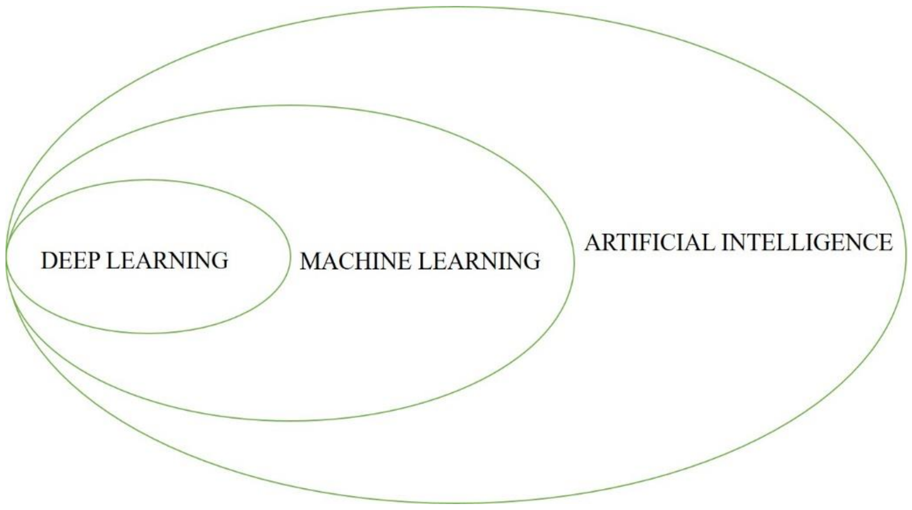


Figure 1 . Hierarchy between Artificial Intelligence, Machine Learning and Deep Learning

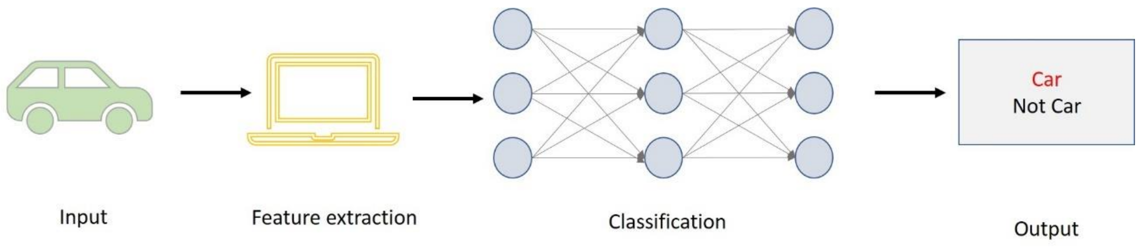


Figure 2 Showing the various component in machine learning for classification task

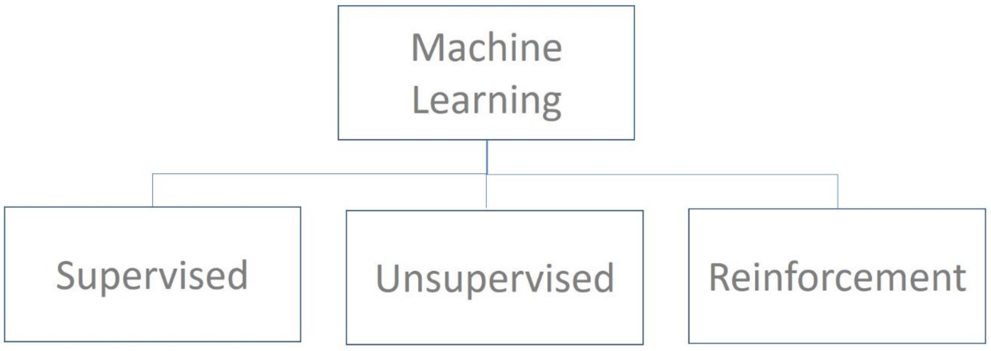


Figure 3 Machine learning comprising of supervised, unsupervised and reinforcement learning

Artificial intelligence is used to define systems that rely on machine learning and deep learning and large amount of data to perceive its environment after being trained and take actions or make decisions in response. It is usually achieved with the use of machine learning as the diagram in figure 1 shows.

As a subset of machine learning, deep learning uses ANNs with multiple layers to important learn features from an input data [20]. Deep learning does not require feature extraction to be done separately by human as in the case of classical machine learning, it rather learns the feature using layers that are deeper (closer to the input) in the network, as shown in Figure 4. How "deep" a network is related to the number of hidden layers it has.

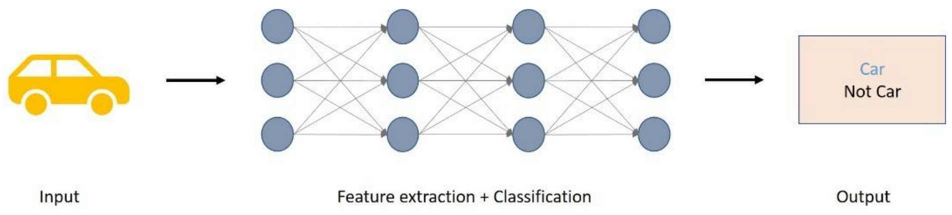


Figure 4 Deep Learning used for image feature extraction and classification

2.2.1.1 Algorithm

Algorithm is the set of instructions or recipe that will define how the model is to be trained to perform a specific task.

2.2.1.2 Model

In the context of machine learning, a model is the output after running the algorithm numerous times (training). It is basically a program that have been trained and now to be deployed for usage (inference).

2.2.2 Supervised Learning

Supervised learning is a machine learning paradigm in which the training set includes the data and the expected outcome of the task with the data. An analogy of this is like a teacher giving a student series of tests questions to solve and also giving them the answers to those problems. Later on, the teacher ask the students to find out how to solve other problems that they will come across in the future [19].

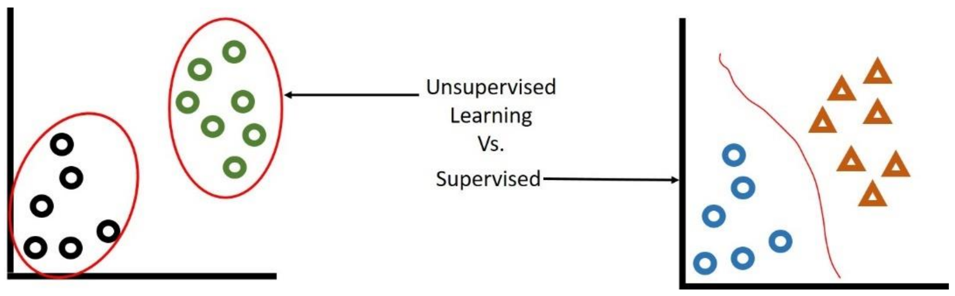


Figure 5 Unsupervised learning (left) showing data clustering and Supervised Learning showing classification

2.2.3 Unsupervised Learning

Unlike in supervised learning where one trains a model by feeding it with labelled input, in unsupervised learning one only feeds the model with raw data without label. The model then figures out patterns that exist in the data, thereby generating analytic insight with minimum human supervision [20]. This makes it very useful for clustering and segmentation of data.

2.2.4 Reinforcement Learning

Reinforcement learning as the third subset of machine learning is a system whereby software agents (or model) take actions in an environment so as to increase the reward it gets. Now, the reward on a particular action it took can be either positive or negative in the case of a positive reward, it “reinforces” that action that brought about the positive reward [22]. In contrast, it also suppresses actions that brought about a negative reward (punishment). As shown in Figure 6, the system keeps on doing this reinforcement/suppression until the model (agent) is good enough to operate well in an environment.

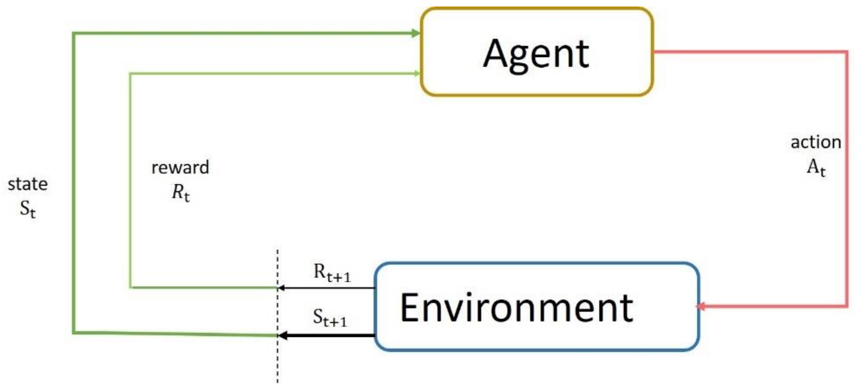


Figure 6 Showing how an agent operate in an environment as is the case with reinforcement Learning

2.2.5 Neural Networks

In simple terms, an (artificial) neural network can be defined as a series of algorithms that consist of mathematical nodes (artificial neurons) arranged in layers which aims to discover underlying relationship in data by trying to mimic the neural networks of a human brain. It does this by a process called training. Usually, it has an input layer, an output layer, and one or more hidden layers, as depicted in Figure 7.

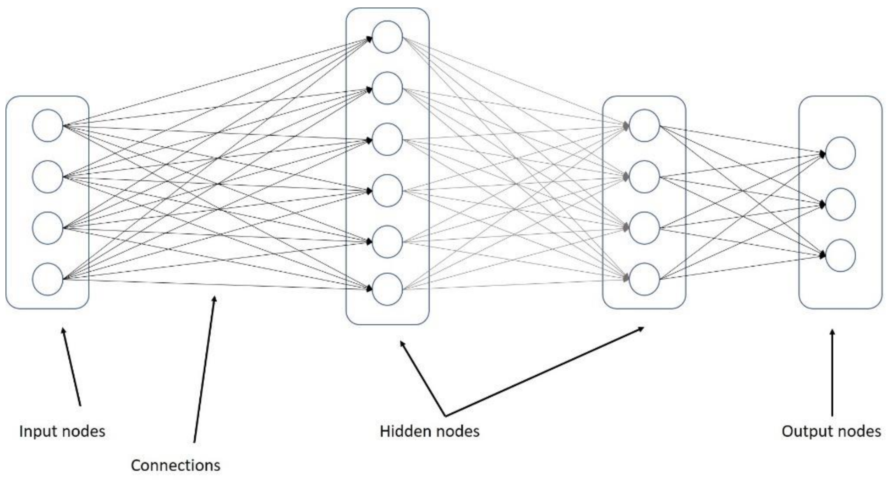


Figure 7 Neural Networks showing the connections of the from the input to output layer

2.2.6 Training an ML Model

Training a machine learning model is the term used to refer to making the model learn. It typically involves passing labelled data to the network in a process known as the forward pass and taking data backwards in the network known as backward pass.

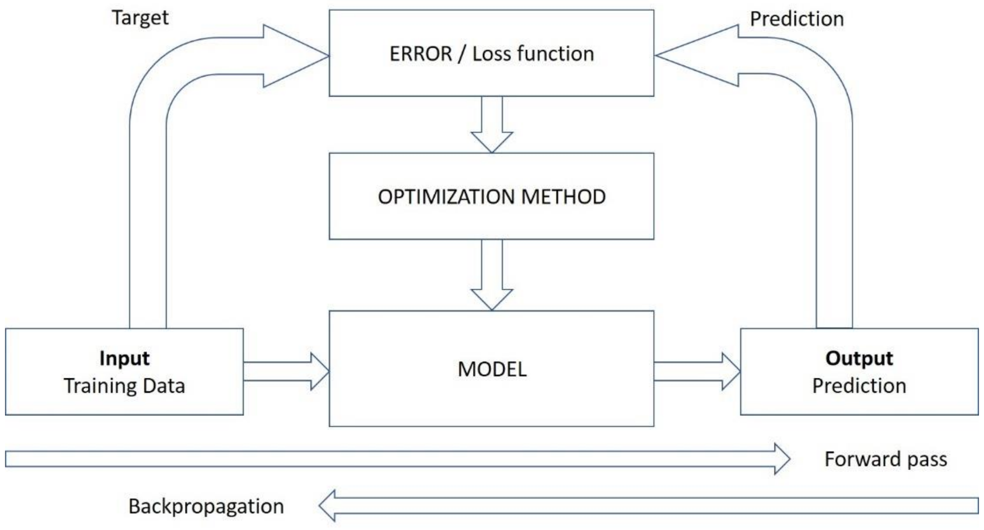


Figure 8 Showing how the training of Machine Learning Model occurs

As shown in Figure 8, during the forward pass, training data are fed to the model and the output is determined; this is usually a classification probability. In the backward propagation (backpropagation), an error or loss function such as cross entropy loss or Mean Squared Error (MSE) is calculated based on the predicted output from the model;

then an optimization method like the Stochastic Gradient Descent (SGD) is used to update the weights and biases of the connections in the network. These forward and backward passes are performed on all the training dataset for a number of times, which is known as the "epoch" until the model has reached the desired performance.

2.2.6.1 Loss Function

There are many types of loss functions used in training; the decision to use a particular loss function depends on the task at hand and the computational capability that is available [21]. Since it is the gradient of the loss function that will be used in the update of weights during training, using a loss function that have an easier to calculate derivate such as the Mean Squared Error (MSE) (also known as L2 or quadratic loss) can be enticing. See Equation 2.1,

$$MSE = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{n} \tag{2.1}$$

where

y_i is expected output and $\hat{y}_i$ is the predicted output and n is the number of classes (number of neurons in the output layer).

Other loss functions include Mean Absolute Error (MAE) (also known as L1 loss), Mean Bias Error (MBE) and cross entropy loss (see Equation 2.2) to name but a few.

$$CrossEntropyLoss = -(y_i \log(\hat{y}_i) + (1 - y_i)\log(1 - \hat{y}_i)) \tag{2.2}$$

2.2.6.2 Stochastic Gradient Descent

The stochastic gradient descent is a technique used to quickly update a solution (in our case, the weights) in machine learning. This approach repeats the update of a solution, f(x) using its gradient ∇f(x) of only a single partial objective function. And the update expression at the n-th iteration is as in Equation 2.3:

$$x_{n+1} = x_n - \alpha \nabla f(x_n) \tag{2.3}$$

where n is known as timestep, and x_n is the value of X at time n. The parameter α is known as the learning rate and it decides the step size in time for this update. Hence, effort must be taken to set its value properly [22]. In doing this, experimentation is usually done with different values for the learning rate in order to determine the best value that is peculiar for our problem. The equation means that the new weight is gotten by subtracting the product of the learning rate and the gradient from the old weight.

2.2.7 Overfitting and Underfitting

In machine learning, overfitting can occur if the model performs excellently on the dataset used during training but poorly on the dataset used for validation testing. This happens when the model memorizes the training dataset instead of learning it features and hence, could not generalize on other datasets. One way to overcome overfitting is by training with more datasets [23]. Underfitting on the other hand is a situation where the model performs poorly on both training and other datasets. It is simply a case where the model has not learnt enough. Further training and data augmentation will improve the performance of the model in this case.

2.3 Convolutional Neural Network

As previously outlined in Section 2.1, convolutional neural network has been the workhorse of deep learning and thus has been used in field such as pattern, voice and image recognition, image processing and most importantly feature extraction from an image. Problems that are solved by CNN have an important assumption, i.e. features should not be spatially dependent [24]. Hence, CNN can simply tell whether an object is present in an image or not, it cannot say where it is located in the image. This can lead to some problems in some particular cases as what it means is that it can detect a human face in an image even if the position of the mouth and nose were to be interchanged. This problem will be solved with CapsNet.

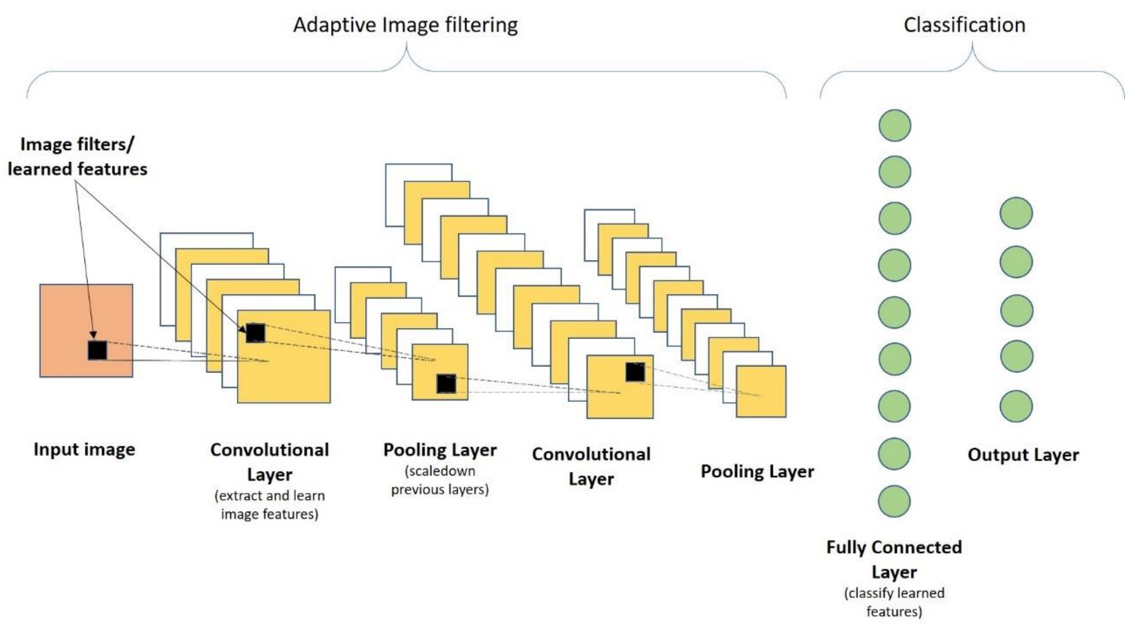


Figure 9 Convolutional Neural Networks structure and its components

In CNN, abstract features are detected by deeper layers (closer to the input) and then propagate forward, and then the higher layer detects higher level features. For example, in detecting a human face, layer one could be detecting low level features such as edges, layer two detects simple mouth, eyes and nose, and then layer three detects the human face. As the features are being propagated, there are usually pooling layers (and non-linearities introduced) in between the convolutional layers, as shown in Figure 9. At the final layer, the network will now be flattened, and all points connected with a fully connected (FC) layer (just a simple neural network) which will be responsible for the classification work. To fully understand the CNN, we shall explore the components of the CNN in the subsequent subsections.

2.3.1 Convolutional Layers

Convolutional layers usually form the first layer in a CNN and they are usually followed a non-linearity function such as ReLU (Rectified Linear Unit) and subsequently the pooling layer [24]. The convolution operation in itself is simply an element-wise multiplication of the so-called filter and the input feature map (input data or image). To further understand the CNN, some terms needs to be explained in the next sets of subsections.

2.3.1.1 Receptive Fields

The receptive field of a particular neuron is the number of neurons that serve as an input to it. For example, Figure 10 shows a simple multilayer perceptron (MLP), as can be seen in the figure, each neuron in the middle layer have a receptive field size of 2 while the single neuron in the output layer has a receptive field size of 5.

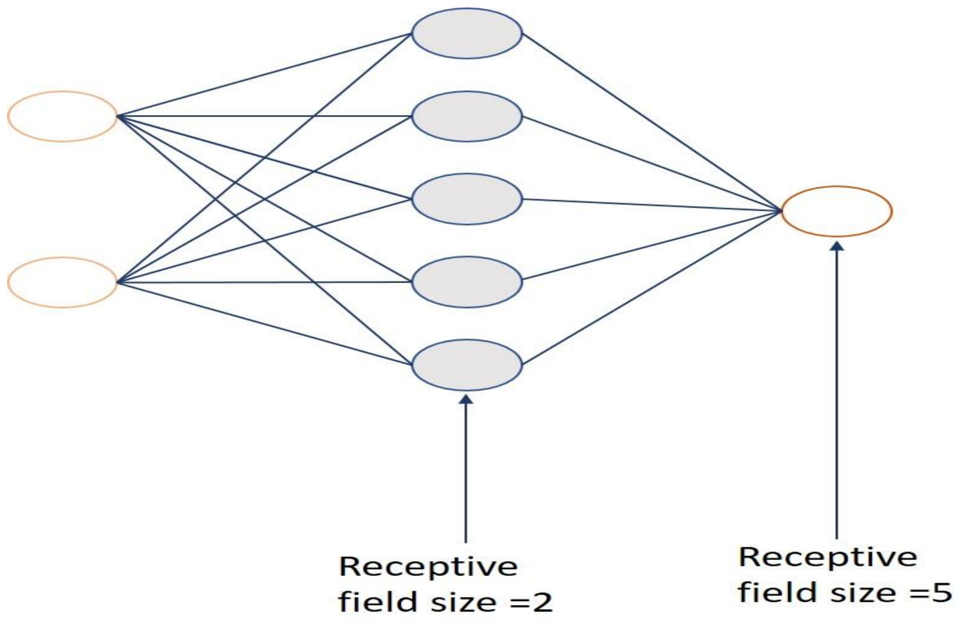
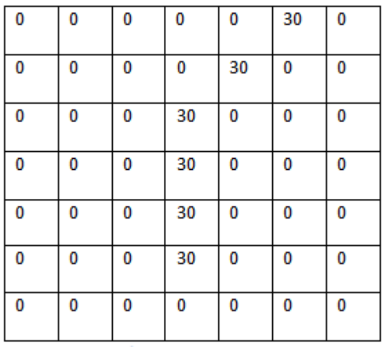


Figure 10 Illustration of the Receptive Fields of a neural network

2.3.1.2 Convolutional Kernels (Filters)

The convolutional kernel or filter is simply a mask or matrix of weights that are dedicated to extracting a particular feature, as shown in Figure 11. Since the convolution is an element-wise multiplication, the part of the filter that has zero will simply multiply-out to zero and the result of the multiplication looks like the shape or edge it is designed to detect. This is illustrated in Figure 11 where the feature (edge) to detect is a curved part and then its pixel values are multiplied with the kernel to get a result which will be stored in another matrix, known as the output feature map (or activation map). During training, the values in the filter are updated to better extract the feature it is meant to extract.



0	0	0	0	0	30	0
0	0	0	0	30	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	30	0	0	0
0	0	0	0	0	0	0

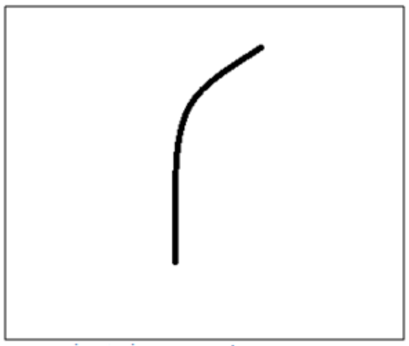


Figure 11 An illustration of a 7x7 filter used to detect an edge

the convolution can be said to just be a multiplication of the filter with its receptive filed and then the result is stored in the output feature map. For the next value in the output feature map, the filter is moved by the stride value and the element-wise multiplication is performed again. This is repeated until the convolution has been performed on the entire area of the image (input feature map), as shown in Figure 12 An analogy to explain this is the use of a flashlight to scan through all the areas of the input feature map.

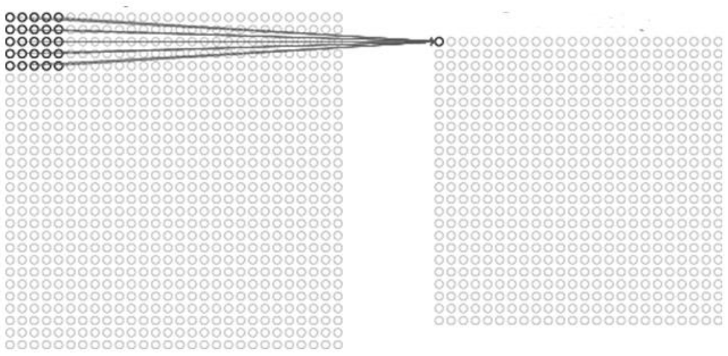


Figure 12 Input feature map and output feature map in a convolution operation [25]

2.3.1.3 Stride

The stride in a CNN refers to the number of rows/columns that is moved in the input feature map before convolving again to get the value of another neuron in the activation map (output feature map). For example, in Figure 13, the filter moves one by one column before performing another convolution, this means the stride used in the convolution is 1.

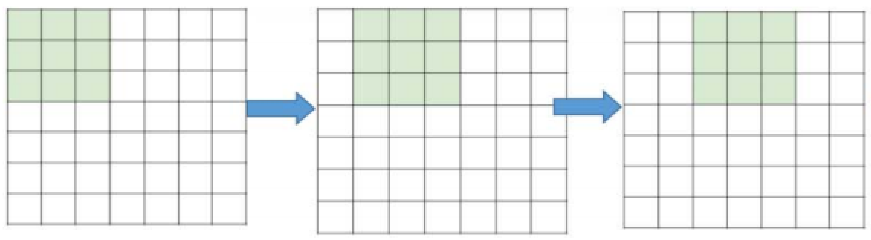


Figure 13 Convolution with a stride of 1. Filter in light-blue [24]

2.3.1.4 Padding

As can be seen in Figure 12, after convolution, the output feature map reduces in size as compared to the input feature map. If reduction continue as we come across more convolutional layers, we might end up losing important features in the image before getting to the classification stage. Hence, we try to keep the size of the output feature map from reducing by adding zeros at its edges. This process of adding zeros at the edges is known as padding. It is not compulsorily used in CNN.

2.3.2 Non-Linearity (Activation Functions)

Activation functions are basically used to introduce non-linearity into our model. After calculating the weighted sum of input and biases to a neuron, we then pass the result through the activation function which will decide whether the neuron will fire ('activate') or not [26]. This way, we try to mimic the biological neurons which also fire when there is enough spike at the input to cause it to fire. There are numerous types of activation functions and the choice to use one or the other often depends on experimentation and the type of problem at hand; however, ReLU has become prominent as the most common for many deep learning problems [27]. Other types of activation function include Sigmoid, Tanh, leaky ReLU and SignReLU, just to name a few [28].

ReLU simply echoes the input when it is more than or equal to zero while it suppresses it otherwise (i.e. sets it to zero when negative), see Equation 2.4. The graph in Figure 15 illustrates the ReLU operation.

$$ReLU, R(z) = \max(0, z) \tag{2.4}$$

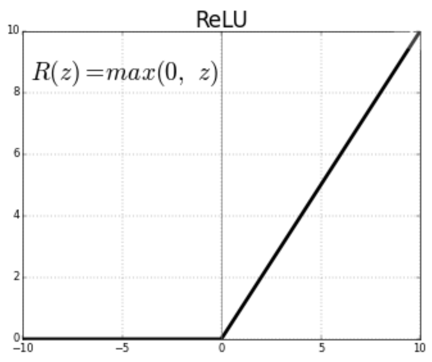


Figure 14 Rectified Linear Unit (ReLU) graph shows that the output is zero when input is less negative

2.3.3 Pooling Layer

Pooling in CNN simply means down-sampling, which is used to further reduce or aggregate the features extracted in the output feature map of a CNN after the convolution and nonlinear operation has been performed. Pooling is mainly done to reduce the model spatial dimension [29] [30]. There are mainly two types of pooling: max pooling and average pooling.

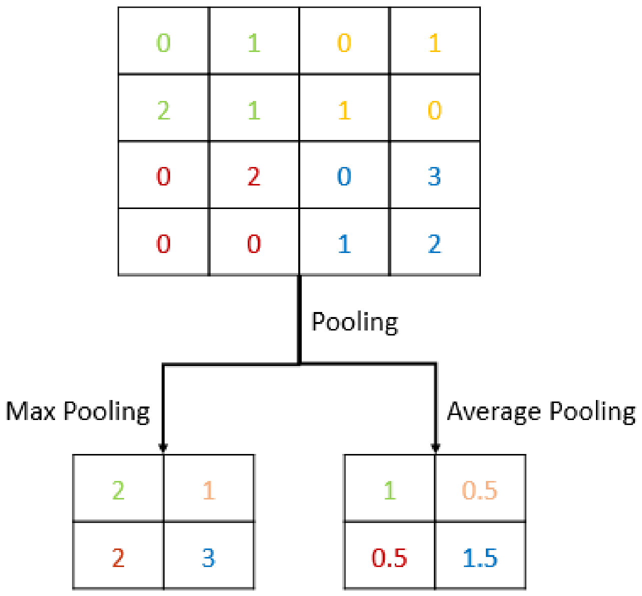


Figure 15 Examples of pooling operations by using a 2×2 filters applied with a stride of 2

In max pooling (as shown on the left-hand side in Figure 15), the maximum value in a section of the activation map is used to represent the entire value of the region and thus reducing the effect of the entire region to the effect of the maximum value. This is also the case with average pooling (right-hand side in Figure 15), the only difference is that instead of taking the maximum, we simply calculate the average of all the values in that window as its representation. It is worthy of note that max pooling is the most commonly used due to its simplicity as it does not require any calculation. These operations may lead to loss some features in the input image, but it is negligible as CNN still performs very well with it. But on the other hand, max pooling makes the model to be invariance to slight rotation in the image.

2.3.4 Fully Connected Layer

The fully connected (FC) layer is essentially an MLP that takes in the final output from the convolutional (and ReLU) layers and use them for classification. It flattens the content of the output feature map from the final convolutional layer and connect them to the individual neurons as may be required. The FC layer can be said to be where the recognition itself happens, while the convolutional parts were only used for feature extraction. The output layer of the FC layer is the output layer for the entire network. This is where the classification probabilities (result) is given. Assuming that a complete CNN is used to determine an image is that of a dog, goat or cat, the cumulative probability that either of the three animals were recognized will be unity. Softmax (normalized exponential function) function is used for this probability distribution [31]. Hence, if dog was recognized with an accuracy of 90%, the output of the softmax layer might be [0.9, 0.04, 0.06] showing that goat and cat were recognized with a probability of 4% and 6 %, respectively. This is expressed as per Equation 2.5.

$$c_{ij} = \frac{exp(b_{ij})}{\sum_k exp(b_{ik})} \tag{2.5}$$

where

c_{ij} : softmax

b_{ij}: input vector

$exp(b_{ij})$: standard exponential function for input vector

k : number of classes in the multi-class classifier

$exp(b_{ik})$: standard exponential function for output vector

2.3.5 Popular CNN Architectures

Due to the popularities and increasing application of CNN in computer vision tasks, the deep learning community has come up with many CNN architectures and varieties of CNN with different levels of optimizations. Some of these architectures have become very popular and useful. Examples of such CNNs are LeNet-5 [32], AlexNet [33], MobileNet [34] and many more. A brief summary of the concept of these network architectures is presented in the next paragraph.

LeNet-5 is a CNN with 2 convolutional layers (each with a non-linear activation and pooling layer) and 3 FC layers. It is not a complicated CNN architecture and the number of parameters (for training) in the network is 60000. AlexNet on the other hand has 8 layers (5 Convolutional layers and 3 FC layers) and has 60 million parameters; ReLU was also first introduced here [35]. MobileNet was developed by Google and it is optimized for mobile application such as smart phones that have lower processing capacity.

2.4 Edge Computing

Data is going to be most important for making intelligent and critical decisions. Based on the type of Internet of Things (IoT) connected device, speed and accuracy are going to be very important – here comes “edge computing.” Edge computing is the processing and analysis of data along a network edge, closest to the point of its collection, so that data becomes actionable in real time without any latency (or with much smaller latency) instead of processing in the cloud. The figure 16 shows the hierarchical nature of edge fog and cloud computing.

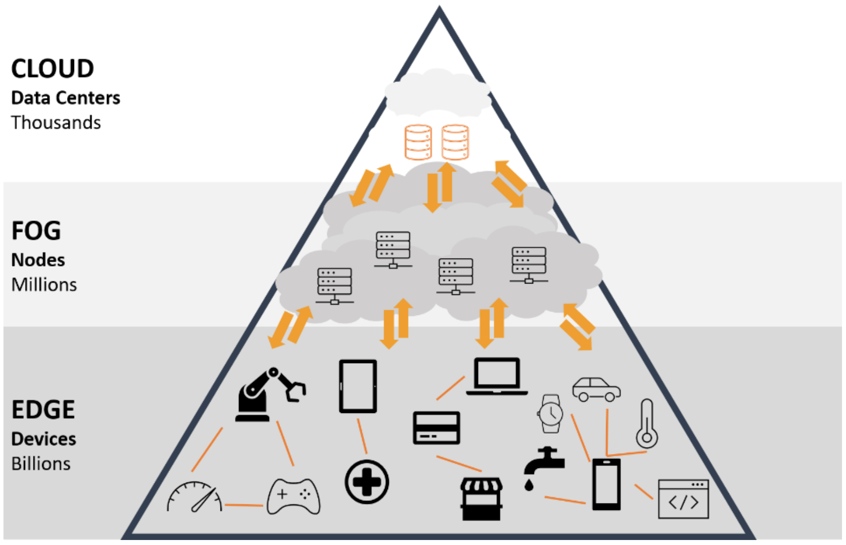


Figure 16 Hierarchy of the cloud, Fog and Edge nodes

2.4.1 Reason for moving Processing to the Edge

The issues mentioned below led to the development of edge computing, the idea of performing processing activities onboard of edge devices (devices at the "edge" of the network). These devices are highly resource-constrained in terms of memory, computation, and power, leading to the development of more efficient algorithms, data structures, and computational methods. The traditional idea of IoT was that data would be sent from a local device to the cloud for processing. Some individuals raised certain concerns with this concept: privacy, latency, storage, and energy efficiency to name a few.

2.4.1.1 Latency

For standard IoT devices, such as Amazon Alexa, these devices transmit data to the cloud for processing and then return a response based on the algorithm's output. In this sense, the device is just a convenient gateway to a cloud model, like a carrier pigeon between the device and Amazon's servers. The device is pretty simple and fully dependent on the speed of the internet to produce a result. If one has a slow internet connection, Amazon Alexa will also become slow. For an intelligent IoT device with onboard automatic speech recognition, the latency is reduced because there is reduced (if not no) dependence on external communications.

2.4.1.1 Privacy and Security

Transmitting data opens the potential for privacy violations. Such data could be intercepted by a malicious actor and becomes inherently less secure when warehoused in a singular location (such as the cloud). By keeping data primarily on the device and minimizing communications, this improves security and privacy.

2.4.1.1 Power Consumption

Transmitting data (via wires or wirelessly) is very energy-intensive, around an order of magnitude more energy-intensive than onboard computations. Developing IoT systems that can perform their own data processing is the most energy-efficient method. AI pioneers have discussed this idea of “data-centric” computing (as opposed to the cloud model’s “compute-centric”) for some time and we are now beginning to see it play out.

2.4.1.1 Communication Bandwidth

For many IoT devices, the data they are obtaining is of no merit as such. Imagine a security camera recording the entrance of a building for 24 hours a day. For a large portion of the day, the camera footage is of no utility, because nothing is happening. By having a more intelligent system that only activates when necessary, lower storage capacity is necessary, and the amount of data necessary to transmit to the cloud is reduced.

2.5 Model Minimization Techniques

Many deep learning models are usually run on very powerful computers usually in remote locations (servers) with large memory capacity to store all the weights and activations. However, as was discussed in Section 2.4, there have been many reasons for computations (particularly inference) to be moved to the edge in some instances but most edge devices (or embedded systems) are usually of lower capacity in terms of memory and compute power available on them, making it difficult to have them run neural networks [38]. Hence, there is need to make the model smaller (and potentially retaining the same efficiency) for them to be efficiently deployed at the edge. Although many model minimization techniques have been devised and applied to neural networks, pruning and quantization has been particularly useful and shall be briefly discussed in the subsequent subsections. Figure 17 shows a process diagram of the post-training minimization process.

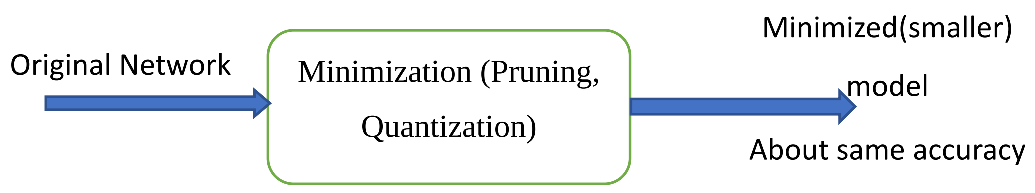


Figure 17 Process diagram of a network minimization scheme

It is also important to discuss regularization, which is a technique that helps reduce overfitting in the network by penalizing for complexity. It does this by adding a penalty term to the cost (or loss) function. Regularization makes the model to better generalize well and perform better on unseen dataset. There are two common types of regularizations, namely L1 and L2 regularization. Both regularizations are essentially the same in operation, only that they have different effects on the model performance. In L1 regularization, the penalty term added to the cost function is the absolute weight, while the square of the weight is added in the L2 regularization [36, p. 2].

2.5.1 Pruning

In simple term, pruning can be said to be the removal of redundant synapses (connections) and or neurons from the network, reducing the size of memory required to store the weights in the network. This also leads to faster computations as there are less parameters present in the network [37] [38]. Pruning varies in types depending on what is being pruned (weights or neurons and layers) and when it is being pruned (during or after training). Figure 18 shows a 3-step iterative pruning process where after the initial training, the system iteratively trains and prunes simultaneously.

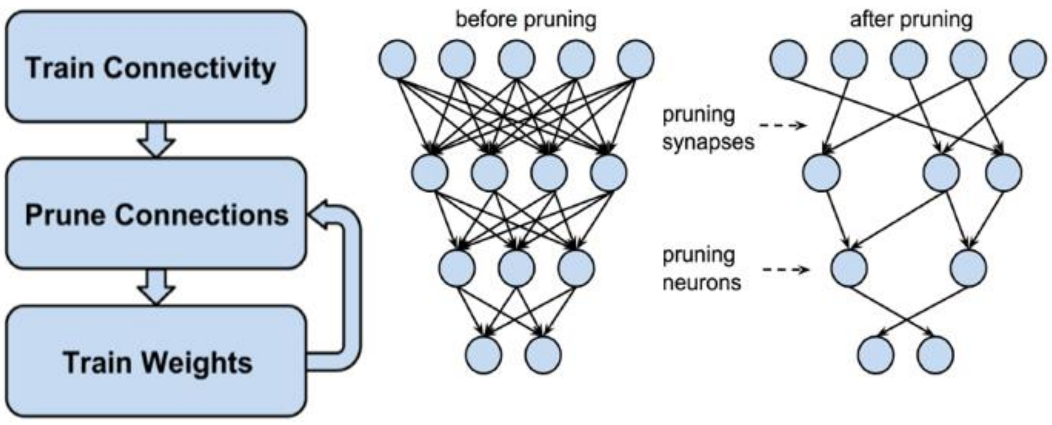


Figure 18 Three-Step Training Pipeline showing the synapses and neurons before and after pruning [39]

Although pruning may lead to a slight loss in accuracy, research has shown that this accuracy can be regained if the model is retrained with some regularization applied [37]. In Figure 19, a network that has had about 90% of the model pruned away was shown to regained all lost accuracy after it was retrained with L2 regularization (colour red curve).

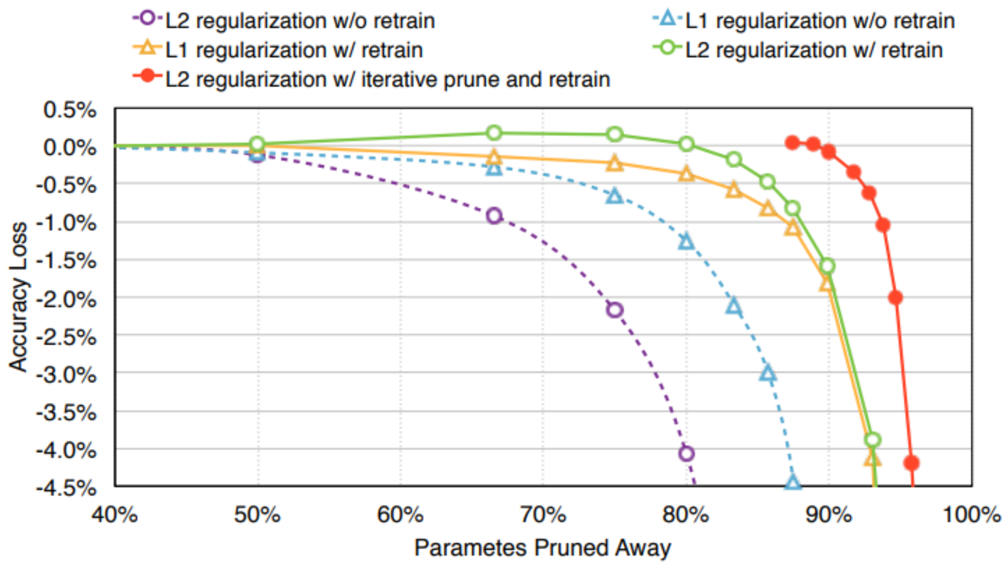


Figure 19 Trade-off curve for parameter reduction and loss in top-5 accuracy [39]

To prune a network after the architecture has been chosen and the model has been trained, we have to first set a threshold for pruning (i.e. any weight less than that threshold will be pruned away). Then the pruning is performed on weight magnitudes that are less than that threshold; after this, the network is retrained until a reasonable accuracy is achieved. If need be, pruning is performed again and the retraining follows again. Applying the iterative pruning on AlexNet (Figure 20) reduces the number of parameters from 60 million to less than 8 million.

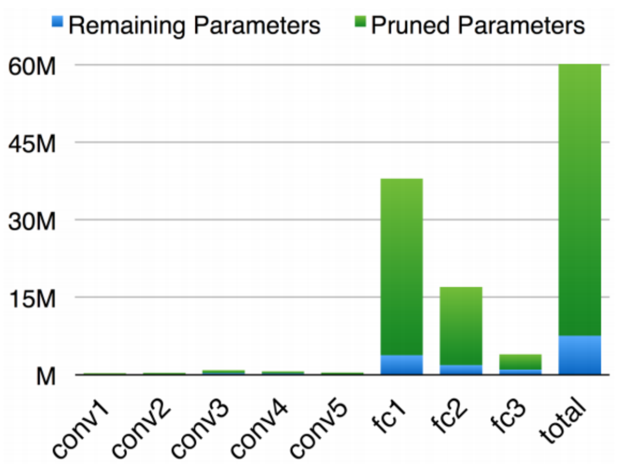


Figure 20 Applying the iterative training on AlexNet, trained with ImageNet dataset [39]

2.5.2 Quantization

Quantization is another common model minimization/optimization technique. Its aim is also to reduce the size and complexity of a network without having an adverse effect on the model accuracy. Quantization can be performed by rounding off many weights that are close in magnitude and using a single value to represent them (scalar quantization). With this, a single value can be used to represent the four different weights, thus reducing the amount of memory required to store the weights. Quantization can also be in the form of representing floating-point weights values with a fixed point representation. With this, a 32-bit floating-point number can be represented with an 8- bit (or even less) fixed-point number [38].

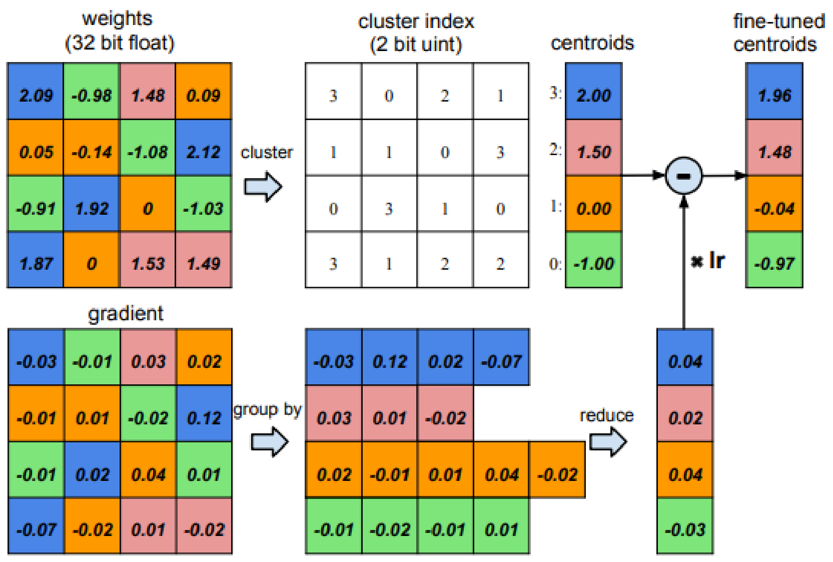


Figure 21 Weight sharing by scalar quantization (top) and centroids fine-tuning (bottom) [37]

An illustration of quantization (by clustering) is shown in Figure 22. Here, weights that are close (given same colour in the figure) are clustered together and represented with a single value (their centroid). For example, four different weights (2.09, 2.12, 1.92 and 1.87) are represented with the value 2.00 and hence the number of bits required to represent them reduces from 128 (32 bits times 4) to 34 (32 bits times one, plus 2 bits). Additionally, a fine tuning is applied on the centroid so as to enable maintenance of the clustering made. Fine tuning is performed by summing up the gradient for all the weights that have the same centroid, and then subtracting this sum from each initial centroid to get the new centroid.

More often than not, pruning and quantization are usually combined to obtain a very good minimization/optimization of a neural network model. Figure 22 shows how this combination (red curve) performs better than applying only one of them as it shows a zero-accuracy loss even after the model size have been reduced to about 3% of its original size [37].

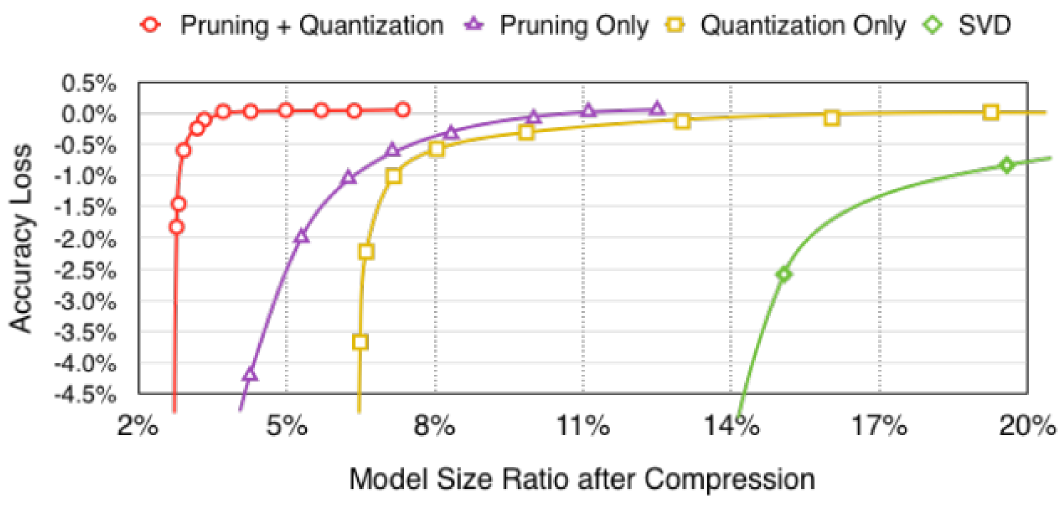


Figure 22 Accuracy v.s. compression rate under different compression methods [37]

Since most embedded devices are not always as powerful as the high-end computers or servers to run complex machine learning models (such as CapsNet or CNN), compressing the model (model minimization) using the techniques described here (and more) is now an important part of the pipeline edge computing paradigm.

3 Capsule Neural Networks

Capsule Neural Network (CapsNets in short) as the name implies is a form of neural network that is composed of ‘capsules’ instead of the traditional artificial neurons. This chapter presents how CapsNets work; first by explaining what a capsule is, and then presenting the CapsNet’s architecture, and other essential characteristics.

3.1 What is a Capsule?

In vanilla neural networks, artificial neurons are used as the computation units and their outputs are usually scalar in nature. Capsules, on the order hand, have vector outputs and they can be defined as a group of neurons that store the instantiation parameters of an object or object part with their activity vectors [5]. Here, the probability of existence of an object is denoted by the length of the activity vector (of the capsule) while its instantiation parameters are represented by the orientation of the vector. In the section that follows, an architecture of a CapsNet used to recognize handwritten digits will be explored and this will give the full workings of the capsule neural network.

3.2 Architecture of a CapsNet

The architecture of a Capsule neural network can be best described when explained in two parts: encoder and decoder.

3.2.1 Encoder Part

Figure 23 shows a simple structure of the encoder part of a neural network used to detect hand-written digits. The encoder takes as input an image of say 28 x 28 x 1 pixel resolution, passing it through the convolution layer and capsule layers, then it get encoded as a 16 - dimensional vector. Note that it does not have to be 16 in dimension, it can be any amount of dimension as desired. But this number denotes how many properties of the image will be encoded. The capsules are only present in the encoder part of the network and the encoder structure also represent the structure used during inference. As can be seen in the figure, the encoding part consists of the input image itself, a convolutional layer, a primary capsule layer and an output capsule layer (called digit capsules in this case). The prediction is then shown as the length of the output vector.

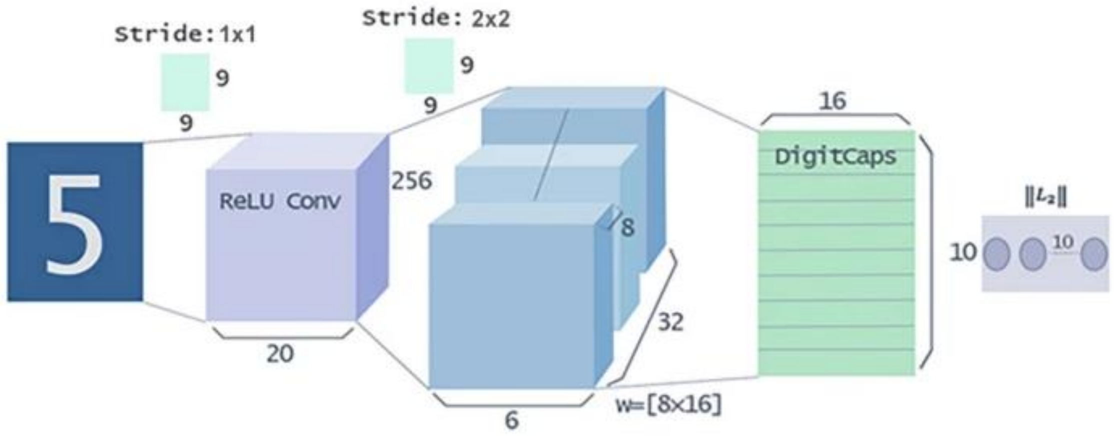


Figure 23 Architectural structure of encoder part of CapsNet

The operation of the encoder is such that we feed the image into the ReLU-activated standard convolution layer, which then applies 256 different filters (9x9 kernels) on the image and we have the output of the convolutional layer to have a 256 number of feature maps (or channels). This output, which is now of 20x20 dimension per feature map, is then fed into the primary capsule. In the primary capsule, the 256 feature maps are grouped into 8 groups each and then we have 32 of such groups (this is represented in Figure 24. Now, kernels of size 9x9x8 are then applied to each group to give us an 6x6 feature map with each element being 8-dimension, or (6x6=36) capsules of 8 dimension each. Applying this across the 32 groups gives us 32 number of 8-D capsules across a 6x6 space thereby making the number of capsules to be 6x6x8x32.

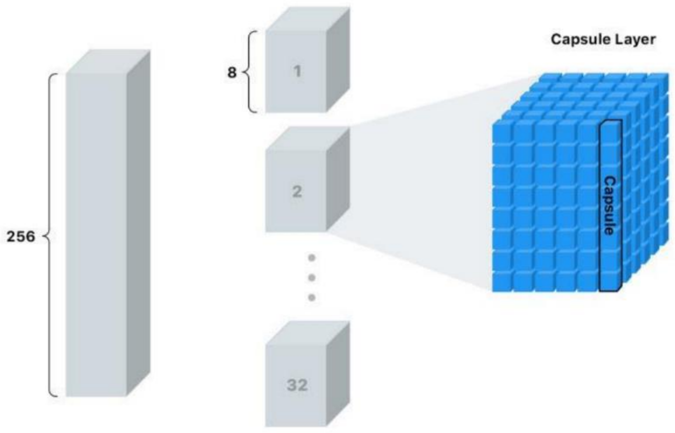


Figure 24 Primary Capsule layer showing the vectoral property of the capsule [25]

In the digit capsule layer, each of the capsules in this layer takes input from all the capsules in the primary capsule layer, i.e. 32x6x6 = 1152 of them in total. Each of these input capsules (8-D vectors) is multiplied by transformation matrix with size 16x8 to convert the 8-dimensional capsules to 16 dimensions for each of the 10 output capsules. A special algorithm is used for routing the vectors from the primary capsules' layers to the digit capsule layers. This algorithm makes it possible for primary capsules to only route their output to the capsules in the output layers that most likely agrees with it output. This algorithm is known as the Dynamic Routing by Agreement Algorithm, as popularized by Hinton [5] (see Section 3.4 for more information).

3.2.2 The Loss Function

As already discussed in Chapter 2, learning in a neural network is basically an optimization process of minimizing a loss function. In the encoder part of the network, the loss function is given by Equation 3.1. Capsules use a distinct margin loss L_c for each category c digit shown in the image. The entire loss of the network during training is the sum of the margin loss and weighted reconstruction loss.

$$L_c = T_c \max(0, m^+ - \|u_c\|)^2 + \lambda(1 - T_c) max(0, \|u_c\| - m^-)^2 \tag{3.1}$$

where T_c=1 if there is an object of class *c*. Furthermore, m^+=0.9 and m^-=0.1. The weighting-factor λ, which is usually set at 0.5, stops learning so as to reduce the activity vectors in all category. And hence, the total loss is the sum of the losses of all classes. The first term of the loss function equation is only executed if the input digit was properly classified while the second part is executed in the event of misclassification. In a situation where the correct output capsule is able to predict correct label with a probability that is more than 0.9, we will have a zero loss, but if the probability is not up to 0.9, then we will have a non-zero loss.

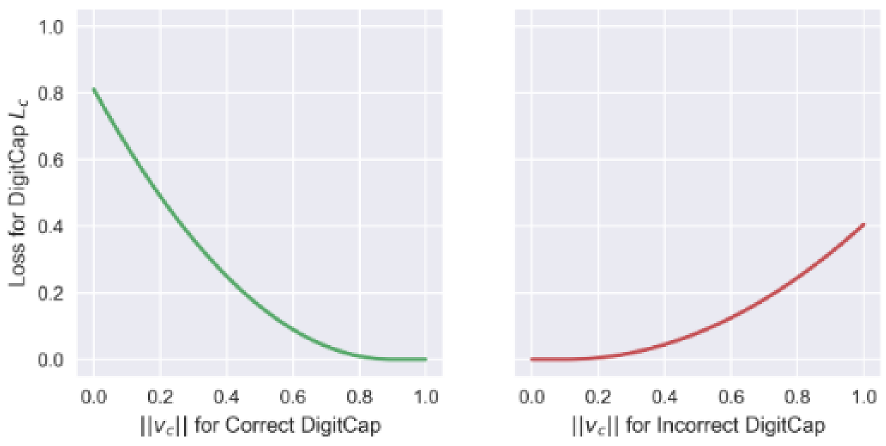


Figure 25 Loss function for correct and incorrect output capsule [40].

For the output capsules that are not in match with the correct labels, T_c will be equal to zero and the only the second term of the equation will be executed (which corresponds to the $(1—T_c)$ part). And in this scenario, we will still have a loss of zero provided that the miss-prediction by the mismatching output capsule was made with a probability that is less than 0.1, otherwise the error will have some values. Figure 25 illustrates these scenarios properly.

3.2.3 Decoder Part

The decoder part of the CapsNet (as illustrated in Figure 26) is mainly fully connected layers of conventional neural networks. It takes in as input the output of the correct output capsule (or DigitCaps) and then decodes it into an image of the correctly predicted digit. It learns all of this during training, and it does not try to decode it when the prediction is not correct, it simply masks them off in that case. The Euclidean distance between the reconstructed image and the original image is used as the loss function for the decoder and the training is dine with the help of backpropagation. This decoding part forces the CapsNet to learn features that are necessary in the reconstruction of the image. And the overall loss function is given as the weighted some of the margin loss (from the output capsule layer) and the reconstruction loss (obtained from the decoder network).

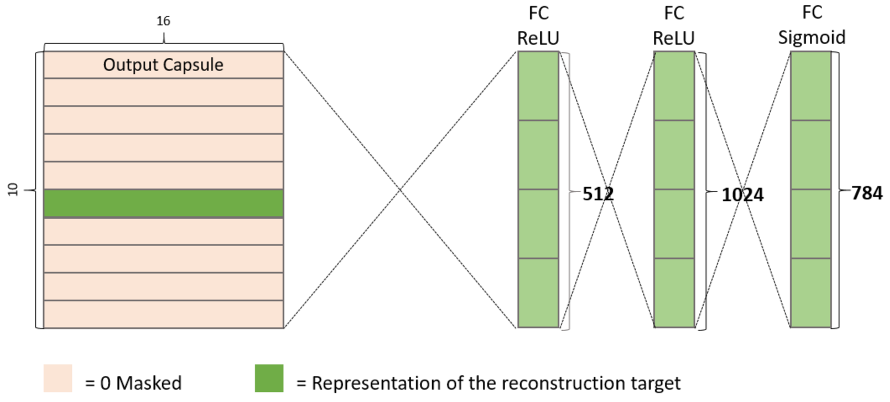


Figure 26 Architecture of encoder part of CapsNet [40]

A summary of the complete CapsNet structure for the recognition of handwritten digits example is shown in Table 1

Table 1 Summary of the CapsNet structure for the recognition of handwritten digits example

Name of Layer	**Function**	**Shape of the output**
Input image	Array of raw image data	28x28x1
ReLu-Activated Convolution	Convolution layer, 9x9 kernels, output is a 256 channel feature map, stride 1, no padding with ReLU activation.	20x20x256
Primary Capsules	Convolution capsule layer with 9x9 kernel. Output = 32x6x6 8-dimensional capsule, stride 2 and there is no padding.	6x6x32x8
Output Capsule (Digit Capsule)	Capsule output computed from a W_{ij} (16x8 matrix) between u_i and v_j (i from 1 to 32x6x6 and j from 1 to 10).	10x16
Fully connected layer 1	Fully connected (ReLU-Activated)	512
Fully connected layer 1	Fully connected (ReLU-Activated)	1024
Reconstructed output image	Fully connected (Sigmoid-Activated)	784(1x28x28)

3.3 CapsNet Forward-Pass Operations

In this section, we look more closely to the operation that occur in a CapsNet during a forward pass. The process describes various operations that take place when vectors are being routed from a deeper (lower) capsule to a higher capsule (output capsule in this case). Figure 27 helps illustrating these operations.

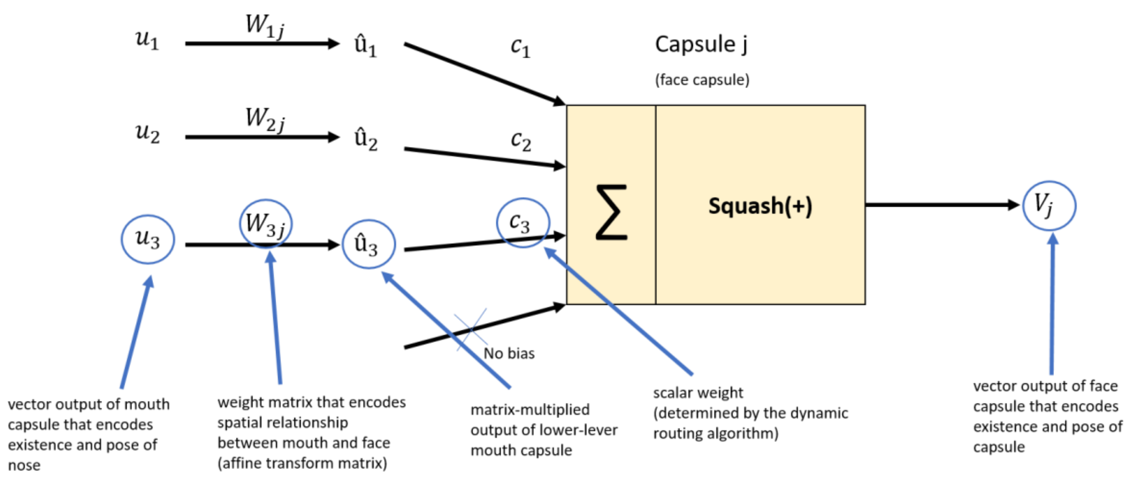

u_1
W_{1j}
$\hat{u}_1$
c_1
Capsule j
(face capsule)
u_2
W_{2j}
$\hat{u}_2$
c_2
u_3
W_{3j}
$\hat{u}_3$
c_3
$\sum$
Squash(+)
V_j
No bias
vector output of mouth capsule that encodes existence and pose of nose
weight matrix that encodes spatial relationship between mouth and face (affine transform matrix)
matrix-multiplied output of lower-lever mouth capsule
scalar weight (determined by the dynamic routing algorithm)
vector output of face capsule that encodes existence and pose of capsule

Figure 27 Diagram showing the data flow operations in a Capsule

For a lower capsule output, U_i and an upper capsule with output V_i, the output of the lower capsule serves as the input of the upper capsule and both outputs are vectors in nature. Four basic operations are performed on the input to a capsule before we have the output of such capsule. These operations are detailed in the subsections that follow.

3.3.1 Matrix Multiplication of Input Vectors

As shown in Figure 27, our capsule gets input vectors ($U_{1,} U_2$ *and* $U_{3,}$) from 3 lower-level capsules. As explained before about capsules, the length of these lower-level capsules represents the probability that the corresponding objects were detected by the low-level capsules and the orientation of these capsules represent internal features or pose of these objects. We now multiply these vectors with a weight matrix W_{ij}, which encodes the spatial information between these lower-level capsules (e.g. Nose, mouth, ear etc. in images of humans) and the upper lever capsule (e.g. face). For example, the spatial relationship between the mouth and nose can be encoded by W_{2j}. These encoded relationship between the mouth and face might be for instance that the width of the face is twice that of the mouth and that the mouth is located at the lower part of the face. The same kind of encoding is done for other capsules in the lower layer. After these multiplications with the weight matrix encoding spatial relationships, we now can now have an output which is the predicted position of the higher-level features. That is, $\hat{u}_2$ might represent the where the face should be according to the detected position of the nose. The same goes for $\hat{u}_3$ and $\hat{u}_1$. And if all the three lower level agree to as to where the position of the face is, then it must be a face. Equation 3.2 shows how these are calculated.

$$\hat{u}_{j|i} = W_{ij} u_i \quad (3.2)$$

3.3.2 Scalar Weighting of Input Vectors

These predicted vectors are then weighted with a scalar value known as the coupling coefficient, C_{ij}. The values of these coefficients help the lower-level capsule decide which higher-lever capsules it should be coupled to. This is done by first sending out its value to every output capsule, and then determine the set of capsules that have high likelihood or agree with it, and then adjust the weighs based on the likelihood calculated; this reduces the weight between it and the higher-level capsules with lower likelihood or agreement, and conversely increases the weight between it and the higher-level capsules with high agreement. The values of the scalar matrix C_{ij} are learned during training, this time not by backpropagation but by the dynamic routing by agreement algorithm which will be explored in Section 3.4.

3.3.3 Sum of Weighted Input Vector

After the weighting has been performed for all lower-level capsule, at this time, all higher-level capsules would have been coupled with all lower-level capsule although with varying weights, now the sum of all the couplings a higher-level capsule has with all lower level capsules. This is shown in Equation 3.3;

$$S_j = \sum_i c_{ij} \hat{u}_{j|i} \qquad (3.3)$$

3.3.4 Squashing

The Squashing function, which takes in vector input and gives a vector output is a novel nonlinearity introduced by G. Hinton [5]. After the input is squashed, its output is capped at 1 (i.e. not more than unity) but its orientation is not changed. Its formula shown in equation 3.4. In conventional neural network, the ReLU function nonlinear activation would be used, but here a squashing function is used instead; one reason being that it can take in vector input and give a vector output and also the fact that it forces the final length to be at most unity, as shown in figure 28, which is good since the length of the vector is a probability.

$$V_j = \frac{\|s_j\|^2}{1+\|s_j\|^2}\frac{s_j}{\|s_j\|^2} \tag{3.4}$$

Where;

V_j = Capsule squashed output vector

S_j = Capsule output vector

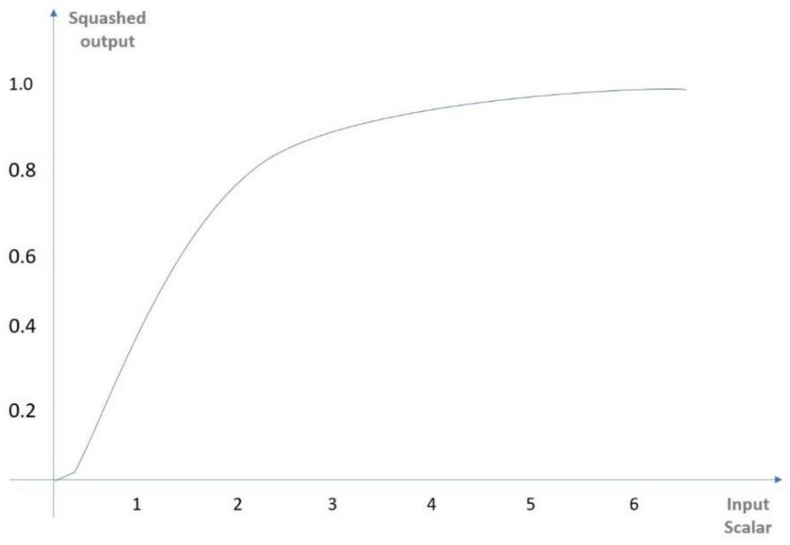


Figure 28 Simulated output of a squashing function

3.4 Dynamic routing by agreement

In the conventional CNNs, the routing of data from a lower level layer, L, to an upper level layer, L+1, is done by the pooling operation. But this is not the case in CapsNet. In CapsNet, this routing is done in an iterative manner using an algorithm known as “routing by agreement”. From the name, it means that there will be an agreement between two capsules before data can be routed between them. So, the intuition behind this is that capsules that are deeper in the network (lower level capsule) will only send their outputs (to serve as inputs) to a higher level capsule that “agrees” with it [5] [41]. This way, capsules from a lower level layer L will not have to send (route) outputs to all capsules in the L+1 layer, and thereby reducing calculations and increasing the efficiency of the network. Before we delve into how this algorithm works, a pseudocode form of the algorithm is shown in Table 2.

Table 2 Dynamic Routing by Agreement [5]

```
Procedure 1: Routing Algorithm

1. Routing function ($\hat{u}_{j|i}, r, l$)
2. for all capsule $i$ in layer $l$ and capsule $j$ in layer $(l+1)$: $b_{ij} \leftarrow 0$ .
3.  for $r$ number of iterations: Loop
4.  for all capsule $i$ in layer $l$: $c_i \leftarrow \mathrm{softmax}(b_i)$
5.  for all capsule $j$ in layer $(l+1)$: $\mathrm{s}_j \leftarrow \sum_i c_{ij}\hat{u}_{j|i}$
6.  for all capsule $j$ in layer $(l+1)$: $\mathrm{v}_j \leftarrow \mathrm{squash}(s_j)$
7.  for all capsule $i$ in layer $l$ and capsule $j$ in layer $(l+1)$: $b_{ij} \leftarrow b_{ij} + \hat{u}_{j|i}.\mathrm{v}_j$ return $\mathrm{v}_j$
```

where $\hat{u}_{j|i}$ is the affine-transformed output of the layer L capsule (or input to the layer L+1 capsule), and v_j is the output of the layer L+1 capsule.

In line 2, b_{ij} is initialized to zero in the beginning. Then some operations are run in a loop r number of times (the typical value of r is 3 [5]). Now, within the loop, each step is explained below;

- **$c_i = \mathrm{softmax}(b_i)$:** The value of the vector c_i is calculated and all routing weight of the low-level layer capsules into probability (between 0 and 1) and also make sure that their sum equals unity. This weight is what determines the "agreement-level" of capsules, but at the first iteration, lower-level capsule have equal agreement-level with all capsules in the higher level and this is the state of maximum confusion.

- $\mathrm{s}_j \leftarrow \sum_i c_{ij}\hat{u}_{j|i}$: In the higher, level layer (L+1), the sum of the linear product of the input vectors, and the weighting factor, c_i is calculated for all capsules. After this is done, the output s_j is squashed ($\mathrm{v}_j \leftarrow \mathrm{squash}(s_j)$) to give the output vector of each capsules in layer L+1. This squashing allows for the direction of the vector be preserved while bounding its length between to 0 and 1 (of probability).

- $b_{ij} \leftarrow b_{ij} + \hat{u}_{j|i}.\mathrm{v}_j$: Here, the initial weight is updated here. In the beginning of the algorithm, the lower-level capsules are in a state of maximum confusion and do not know which higher level capsules will agree with them, but after this update here, the value of the weights will be adjusted and hence, lean towards some a capsule in the upper level. This can be said to be the main part of this algorithm as the output of the higher-level capsule v_j and that of the lower level capsule $\hat{u}_{j|i}$ are checked for similarity (the dot product $\hat{u}_{j|i}.\mathrm{v}_j$ is essentially the similarity check). The new weight will be bigger between two capsules in agreement but decreases between capsules that are not.

- These steps are repeated for a number of iterations (preferably 3) at which capsules in the lower layers have learned which capsule in the higher layer agrees with it and then subsequently only rout data to it.

3.4.1 Decoder

The decoder part of the flowgraph gets the vector of the predicted class(es) from the prediction block (after masking with or without label) and then reconstruct the image (or object) that was detected. The reason for masking is that we want to distinguish training from testing of the network. Unlike testing period where the predicted class is just reconstructed with calculation of loss, during training, the labels of the predicted class are needed in order to find the reconstruction loss, but this is not needed during testing, so we put a flag (called "Mask with Labels") to distinguish this.

As already depicted in Figure 26, the decoder is simply a neural network (an MLP) that takes in the predicted class vector and reconstructs the image from it. Hence, the number of neurons in the output layer of the decoder must be equal to the total number of pixels in the original image as each pixel will be generated by one neuron (i.e. for a 28x28x1 image, the number of neurons in the output layer of the decoder will be 784).

3.4.2 Losses

Next, we shall look at the losses of the network.

3.4.2.1 Margin Loss

The margin loss is the loss calculated based on the classification output of the output capsules. The margin loss calculated separately for class in the output capsule and it is given by the Equation 3.1 below as previously explained in Section 3.2.2.

3.4.2.2 Reconstruction Loss

On the other hand, the reconstruction loss is the squared difference between the original image and the reconstructed image, as per Equation 3.6.

$$R_{Loss} = (Original\ image)^2 - (Reconstructed\ image)^2 \tag{3.6}$$

with R_{Loss} being the reconstruction loss.

3.4.2.3 Final Loss

The final loss is the sum of the of both margin and reconstruction losses. As we can see already, the reconstruction loss depicts the difference in the predicted and the original image and if it is left to dominate the loss, the network might end up trying to memorize the particular image, making it difficult to generalize (i.e. leading to overfitting). Hence, we scale down the effect of the reconstruction loss by a factor β so as to allow the margin loss dominate training as can be seen in Equation 3.7.

$$F_{Loss} = (Margin\ Loss) + \beta(Reconstruction\ Loss) \tag{3.7}$$

with F_{Loss} being the final loss and the value of the scaling factor, β, much smaller than unity. A typical value will be about 0.0005 [42].

3.5 Drawbacks of CNN (Convolutional Neural Network)

To fully appreciate the advantages that CapsNet brings, it is important to look the drawbacks that CNN has. Although CNN has seen rapid rise in application and are performing excellently in many applications, it has some major challenges inherent in it and some of these is motivation for the development of capsule networks. For instance, CNN can predict the presence of an object in an image but cannot give the instantiation parameters such as pose, texture and deformation of the object [5] [43]. Also, the pooling operation used in CNN makes it lose some information about the object, so to train a CNN model, a large amount of data will be required to achieve a good efficiency. In addition, CNN does not keep the spatial relationships among object in an image, it simply gives a probability value of the presence or absence of an object in an image. These makes CNN invariant instead of being equivariant to translation in the image [43]. Lastly, as pointed out by [44], CNN is prone to adversarial attack like pixel perturbation which can have grave consequence if used it leads to wrong classification.

3.6 Comparison of Neurons in CNN and Capsules in CapsNet

A concise difference between the operations in CNN and CapsNet is presented in Table 3. Figure 29 also depicts this difference in the form of a diagram.

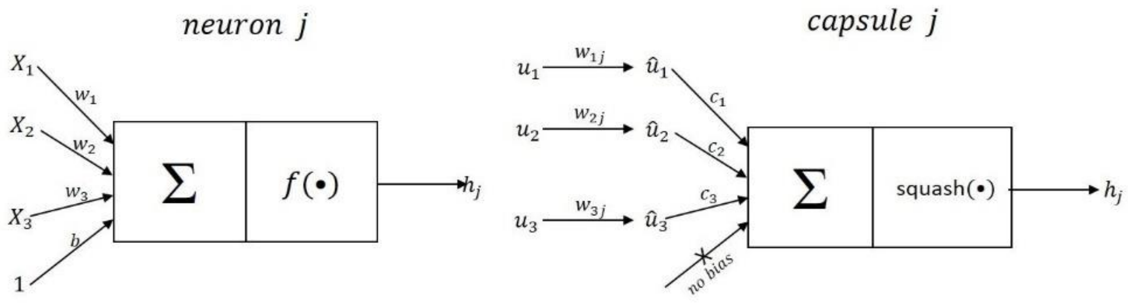


Figure 29 Neurons vs. Capsules showing the difference in the structure

Table 3 Comparison between Neurons and Capsules

		Traditional Neuron vs.	Capsule
Input from low-level capsule/neuron		**scalar$(\boldsymbol{x_i})$**	**vector$(\boldsymbol{u_i})$**
Operation	Affine Transform	-	$\hat{\boldsymbol{u}}_{j\vert i}=\boldsymbol{W}_{ij}\boldsymbol{u}_i$
	Weighting	$a_j = \sum_i a_i \boldsymbol{x}_i + b$	$\mathbf{s}_j = \sum_i c_{ij}\hat{\boldsymbol{u}}_{j\vert i}$
	Nonlinear Activation	$h_j = f(a_j)$	$\boldsymbol{v}_j = \frac{\lVert S_j \rVert^2}{1+\lVert S_j \rVert^2} \cdot \frac{S_j}{\lVert S_j \rVert}$
Output		scalar(h_j)	vector$(\boldsymbol{v}_j)$

3.6.1 Invariance vs Equivariance

Invariance in a model is the detection of the presence of (part of) an object without translational or rotational variations in the original image. The pooling layer makes CNN do this, although this was not the intention of the pooling layer. The way CNN recognizes a high-level object such as a face is just to have neurons which recognize the low level parts such as mouth, nose and eyes to get fired. This means that even if the mouth and the ear positions are interchanged, a CNN would not know this, as long as it has been trained with many translated or rotated examples of that part of the object. On the other hand, CapNet is able to keep the spatial relationship among objects in an image so it can extrapolate possible variants of an object without being trained with those rotated or translated versions [45]. This is known as “Equivariance”. Thus, CNN requires much more datasets of the same object for training as compared to capsule network to achieve the same amount of efficiency.

3.7 Performance Metrics

In the next chapter, terms such as training loss, training accuracy, validation loss and validation accuracy will be used in defining the performance of our model, and it is important we explain them here.

3.7.1 Training and validation

The term “Training” is used to indicate when the model is learning (i.e. any action whose overall effect led to the update of weights in the model). Validation, on the other hand, is simply used to denote the evaluation of the model. Validation is similar to testing, the difference is that testing is done after the model has been completely trained and while validation is done during training and it helps in fine tuning the parameters of the model.

3.7.2 Loss and Accuracy

The accuracy of the network is simply the ratio (percentage) of the correctly classified images to all the images used as given in Equation 3.8.

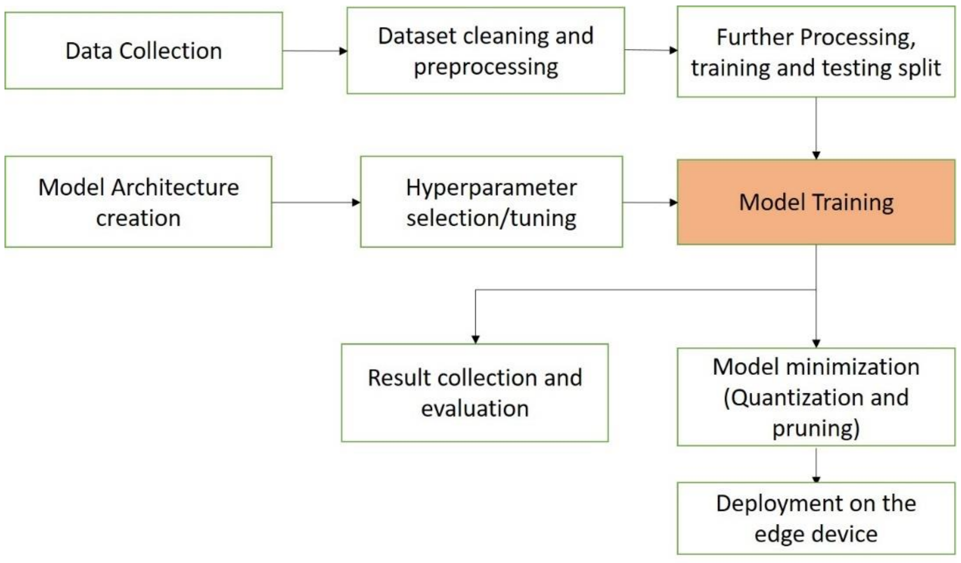
$$Accuracy = \frac{\sum Images\ classified\ correctly}{\sum All\ images} \tag{3.8}$$

Loss is the sum of the errors made for each example in training or validation sets averaged over the entire dataset. And unlike accuracy, it is not a percentage and expressions for it has been previously given in Section 3.5.2.

This chapter has introduced the main elements and properties of the CapsNet. The next chapter presents the implementation part of this study.

4 Implementation Workflow

This section contains the implementations and experimentations carried out in this project; it begins by giving a flowgraph of the workflow from data collection till the deployment of the model on an edge device as depicted in Figure 30. Figure 30 shows the process involved in this project implementation for a CapsNet. Moreover, a CNN model was also built and trained to serve as our baseline for results comparison; it should be noted that this same workflow depicted in Figure 30 was used for the CNN model as well, so there will not be a separate discussion for the CNN model although a summary of its architecture is presented later in this chapter.



Data Collection
Dataset cleaning and preprocessing
Further Processing, training and testing split
Model Architecture creation
Hyperparameter selection/tuning
Model Training
Result collection and evaluation
Model minimization (Quantization and pruning)
Deployment on the edge device

Figure 30 Implementation workflow of CapsNet model training and deployment on the edge device

As shown in figure 30, the workflow begins with data collection and its processing, which are further elaborated, then the CapsNet model is created, and then we tune its hyper parameters to assess what works best, before the training of our actual model is performed. When the training is done, we save the model and go ahead to optimize for inferencing on an edge (i.e. embedded type) device. The training results are also collected and analysed in Chapter 5.

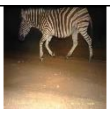
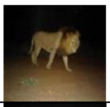
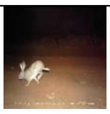
4.1 Datasets, Training and Pre-processing

The datasets of thirteen (13) categories of animals were used for this study and they were obtained from three different sources. Images of the following animals were used; female deer, male deer, hare, polar bear, wolf, reindeer, elk, impala, zebra, lion, cheetah, cats and dogs. That of cats and dogs were used for an initial model that was evaluated with the aim of fine-tuning and obtaining appropriate hyperparameters that are suitable for training CapsNet with animal images, while images of the eleven remaining animal categories were used for the final training. Table 4 shows some images of the 11 animals and the sources were these images were obtained as follows;

- Camera recording of video and then conversion of the video to frames.
- Snapshot Wisconsin dataset from the camera-trapped images used in the work "Identifying Animal Species in Camera Trap Images using Deep Learning and Citizen Science" [7].
- Lastly, web-scraping was done with a python script to get animal images from Google.

These sources were used because these images are similar to real life datasets that deep learning models are expected to see when deployed. After getting the images, there were frames with either empty images of wrong images, these were identified and removed. Then they were resized to resolutions of 28x28, 32x32, 50x50 and 80x80. Also, conversions were made from RGB format to grayscale. Some techniques employed to do the above are shown in the next sections. After the image processing part, the datasets were divided into different classes and then converted to pickle format (as byte files) before saving, to be used later for training.

Table 4 Showing datasets of the animal classes, their image sample and their respective sources

Class	**Image**	**Source**
Polar bear		Web scrapping with python script
Cat		Video to frame conversion
Zebra		Snapshot Wisconsin dataset
Lion		Snapshot Wisconsin dataset
Cheetah		Snapshot Wisconsin dataset
Dog		Video to frame conversion
Elk		Snapshot Wisconsin dataset
Impala		Snapshot Wisconsin dataset
Rabbit Hare		Snapshot Wisconsin dataset
Reindeer		Web scrapping with python script
Female deer		Web scrapping with python script
Male deer		Web scrapping with python script
Wolf		Web scrapping with python script

4.2 Image Processing

4.2.1 Conversion of RGB to Grayscale

Conversion of colour images (3 channels RGB) to grayscale (single channel) can be done in either of the following ways;

1. **Average method**: This is a very trivial method whereby an average of the values of three colors (Red, Green and Blue) are taken, and this gives us the grayscale value. That is;

$$\text{Image(grayscale)} = \frac{\text{image(Red)} + \text{Image(Green)} + \text{Image(Blue)}}{3} \tag{4.1}$$

2. **Weighted method or luminosity method**: There is a problem with the above average method; we only took the average of the colours, making each colour have equal contribution. But in reality, each colour has a different wavelength[46] and thus have different contribution to the image. So, in the Weighted method, the contribution of the red color is decreased since it has a longer wavelength than the green color, the contribution of the green colour is slightly increased and the blue color will occupy the remaining contribution. A suitable factor is indicated in Equation 4.2. The result of applying this method on a picture is shown in Figure 34.

$$\text{Image(grayscale)} = \frac{0.3 \text{ x image(Red)} + 0.59 \text{ x Image(Green)} + 0.11 \text{ x Image(Blue)}}{1} \tag{4.2}$$

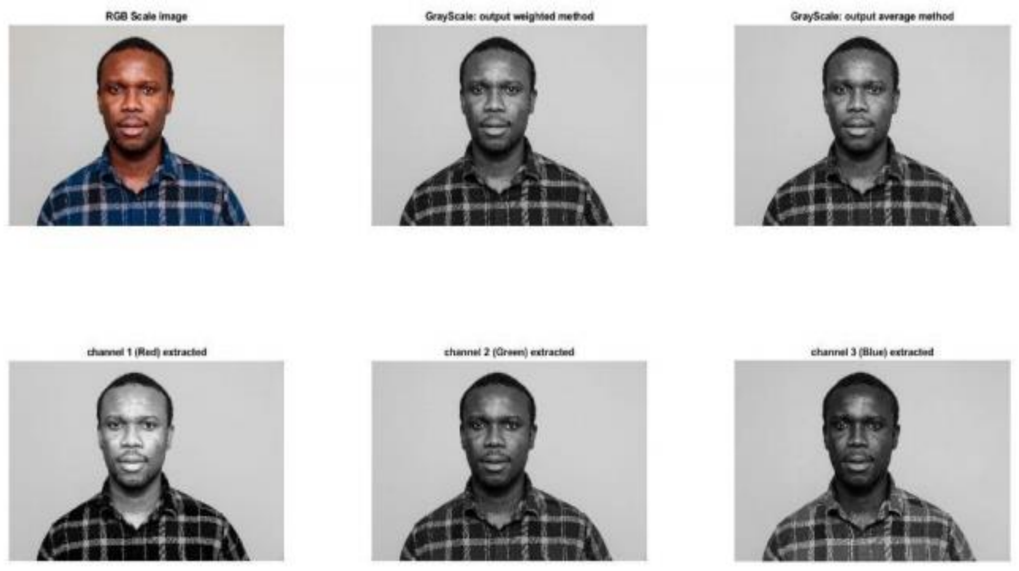

RGB Scale image
GrayScale: output weighted method
GrayScale: output average method
channel 1 (Red) extracted
channel 2 (Green) extracted
channel 3 (Blue) extracted

Figure 31 Results of converting RGB images (top left) to grayscale image using the weighted method or luminosity method (top middle) and the average method (top right). The bottom part show the red, green and blue channels of the RGB image.

4.2.2 Resizing of image (Image resampling)

The process of resizing an image is known as image resampling and there are three main methods that can be used for this [47]. Viz;

1. **Nearest neighbor**: The output pixel (in the resized image) assumes the value of the pixel nearest to it from the original pixel. Here, only one pixel from the original image is used to generate the pixel in the output image.
2. **Bilinear interpolation**: Here, each pixel in the resized image is a weighted sum of the 2-by-2 neighborhood pixels that are nearest to it in the original image.
3. **Bicubic interpolation**: Just like the bilinear interpolation method, the output of each pixel in the resized image is a weighted sum of a 4-by-4 neighborhood pixels in the original image.

In this project, the bicubic interpolation has been employed.

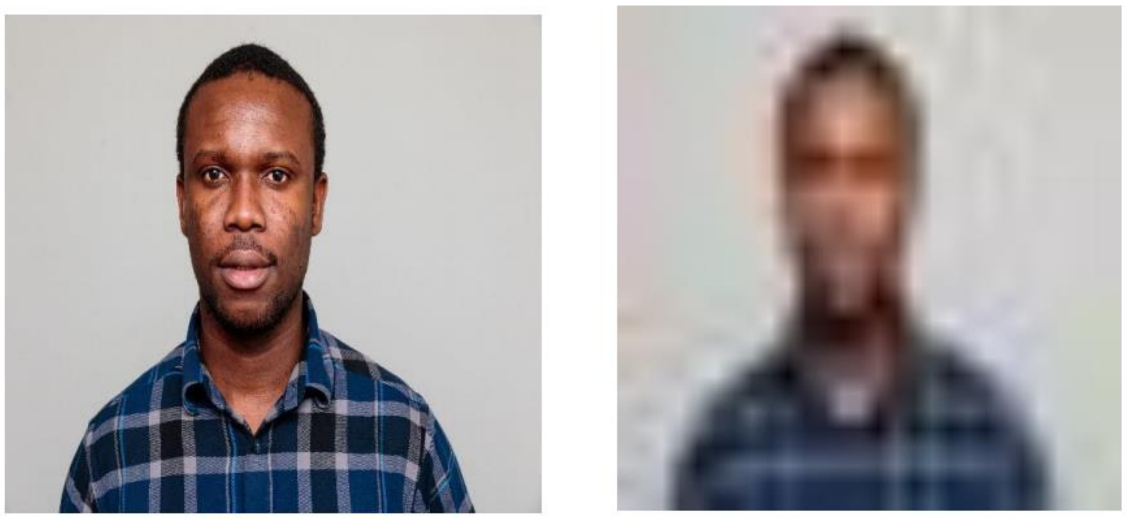


Figure 32 Original and resized images using the nearest neighbour technique

4.3 Software Environment

Image recognition tasks using deep learning usually involves very complex matrix calculations, so a system with good configurations were required for this training of the models. Three systems were used;

- Personal Computer (PC): Intel octa-core Core-i7-2630Q running at 2 GHz clock speed with 8GB memory. This was used for the rather lighter training work. It usually takes about 8 hours to train a 2-class CapsNet model of 12501 28x28x3 images per class for 100 epochs.

- Cloud: Google-Colaboratory platform which provided free access to CPU, GPU and TPU for some limited amount of time. The training often gets interrupted so requires constant monitoring. It usually takes about 5 hours to train a 2-class CapsNet model of 12501 28x28x3 images per class for 100 epochs.

- A Hewlett-Packard (HP) workstation with 2.5 GHz intel Core-i7, memory of 16 GB DDR3, Nvidia Graphic Processing Unit (GPU) GeForce GTX 1080 with GPU clock speed of 1888 MHz and GPU memory of 9028 MB. This was used and for the more complex processing. It usually takes about 50 minutes hours to train a 2-class CapsNet model of 12501 28x28x3 images per class for 100 epochs.

Python 3 was used as the programming language and the implementations were done in the Jupyter - notebook which is available in the Anaconda 3.7 environment. Deep learning frameworks Tensorflow-gpu 1.15 was used in both frontend and the backend while Keras 2.2.4 was used in the frontend. Tensorboard 1.5 was used for recording and visualizing the model training logs and histories. Other packages that were used include Numpy, Matplotlib, Pickle, and OpenCV libraries. And finally, to make the model run on the GPU, several drivers and libraries were installed as specified by Nvidia [8] [1], they include the Nvidia driver, CUDA Toolkit v10.1 and CuDNN v9.1.

4.4 Training CapsNet (Capsule Neural Network)

Training a neural network is always about updating weights and biases, but the architecture is a little bit different in CapsNet and it shall be examined in the subsequent subsections using Figure 31.

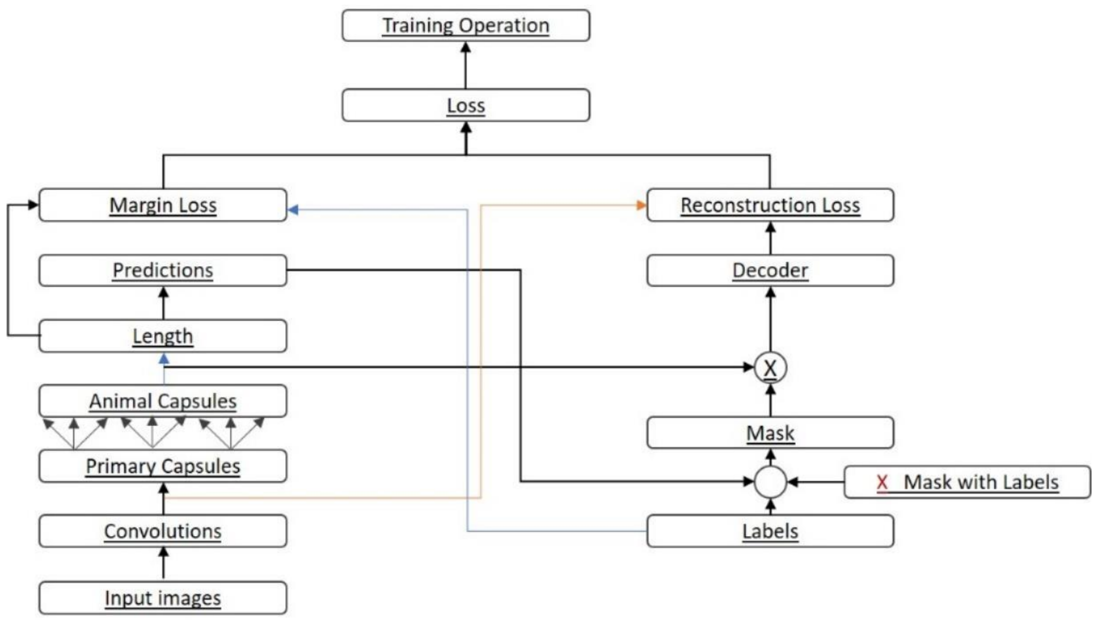


Figure 33 Training Flow graph used to train the “Animal” CapsNet

Figure 31 shows the flow graph for CapsNet training (with little modification for testing) and as can be seen from the figure (bottom left side), it begins with an input image with size (28x28x3) and then through a convolutional layer(256 channels, filter size = 9x9, stride = 1) and then to the primary capsule layer. Now between the primary capsule and output capsule is where the dynamic routing algorithm is takes place. It should be noted that pooling was not used at all, even at the convolution layer. The output of the output layer is an array of vectors for all classes. The length of these vectors are then calculated in order to get the probability of a class (object) detection which now serves as the predicted output. The flow just described can be used for only inference. But for training, much more computations are required and the extra flows will be discussed in section that follows.

4.4.1 Decoder

The decoder part of the flowgraph gets the vector of the predicted animal class from the prediction block (after masking with or without label) and then reconstructed the animal image (or object) that was detected. The reason for masking is that we want to distinguish training from testing of the network. Unlike testing period where the predicted class is just reconstructed with calculation of loss, during training, the labels of the predicted class will be needed in order to find the reconstruction loss but this is not needed during testing, so a flag was put (called “Mask with Labels”) to distinguish this.

As already depicted in figure 26, the decoder is simply a neural network (an MLP) that takes in the predicted animal vector and reconstruct its image from it. Hence, the number of neurons in the output layer of the decoder must be equal to the total number of pixels in the original image as each pixel will be generated by one neuron (i.e for a 28x28x3 image, the number of neurons in the output layer of the decoder will be 2352). In summary, the CapsNet architecture details is shown in table 4.

Table 5 Summary of the CapsNet Architecture used for Animal classification

CapsNet Layer	**Details**
Input Image	Width = 28, height = 28, channels = 3
Convolution Layer	Filters = 256, kernel size = 9x9, stride = 1, Activation = ReLU, with padding.
Primary Capsule	Vector size = 8, channels = 32, kernel size = 9, stride = 2
Animal Capsule	Vector size = 16, capsules = 11, routing = 3
Decoder Fully connected layer 1	512, ReLU
Decoder Fully connected layer 2	1024, ReLU
Decoder output layer	2352, sigmoid

4.5 Model Configuration and key hyperparameters

Hyperparameter tuning is necessary in order to find the best configuration for the main model. First, a CapsNet model of two classes (cats and dogs) was developed and several instances thereof were trained with various hyperparameter changes and the result was monitored. The sizes of images, number of channels, batch size and validation split were varied for different number of images per class. Table 5 shows the summary of the parameters; their performance are analysed later in Chapter 5.

Table 6 Training parameters used for the two-class CapsNet model

Instance	No. of image per class	Image size (number of pixels)	Number of channels (3 channels: RGB; 1 channel: grayscale)	Batch size	Validation split (ratio of images for training/validation)
1	1000	32x32	3	32	0.1
2	1000	50x50	3	32	0.1
3	1000	50x50	3	32	0.2
4	1000	80x80	3	32	0.2
5	1000	80x80	3	32	0.2
6	1000	80x80	3	32	0.2
7	1000	32x32	3	32	0.3
8	1000	50x50	3	32	0.3
9	12501	28x28	3	32	0.2
10	12501	28x28	3	32	0.1
11	12501	28x28	1	32	0.2
12	12501	32x32	3	256	0.2
13	12501	32x32	3	32	0.3
14	12501	32x32	3	32	0.2
15	12501	50x50	3	32	0.2
16	12501	50x50	3	256	0.2
17	12501	50x50	3	32	0.1
18	12501	50x50	3	32	0.3
19	12501	50x50	3	64	0.2
20	12501	50x50	3	32	0.2
21	12501	50x50	3	32	0.2
22	12501	50x50	3	32	0.2
23	12501	80x80	3	32	0.2

After the training for the two classes-model was completed, the 11-class CapsNet model was then trained as well with an image size of 28x28x3 with batch sizes 32, and 256. The validation split (ratio of images for training/validation) was also varied.

4.6 Model Conversion and Minimization

After training of the models, minimization (quantization and pruning) of the model was done to reduce its size and convert it to a Tensorflow-Lite (a Tensorflow version optimized to run on mobile and embedded/edge devices) model before deploying to the edge device. Post training quantization (as opposed to quantization-aware training) was used as this was more straightforward in implementation and both methods leads to similar reduction in size of the model as concluded from experimental observation.

4.7 Embedded Hardware Deployment Setup

After the above optimization of the trained model, it is now deployed on the edge device. The steps used in deploying the model on the board are depicted in the flow graph in Figure 34. The embedded hardware used is the Nvidia Jetson TX2 board, which is a GPU-based power-efficient embedded AI computing edge device. It features and Nvidia Pascal GPU architecture with 256 Compute Unified Device Architecture (CUDA)-cores, together with it are dual-core Nvidia Denver 64-Bit CPU and ARM Cortex-A57 CPU.

Also, the Google coral range board containing an edge Tensor Processing Unit (TPU) was initially experimented on but it was later dropped as it couldn't work well because of some limitations in the TPU. It does not support 4-dimensional tensors (which my model has).

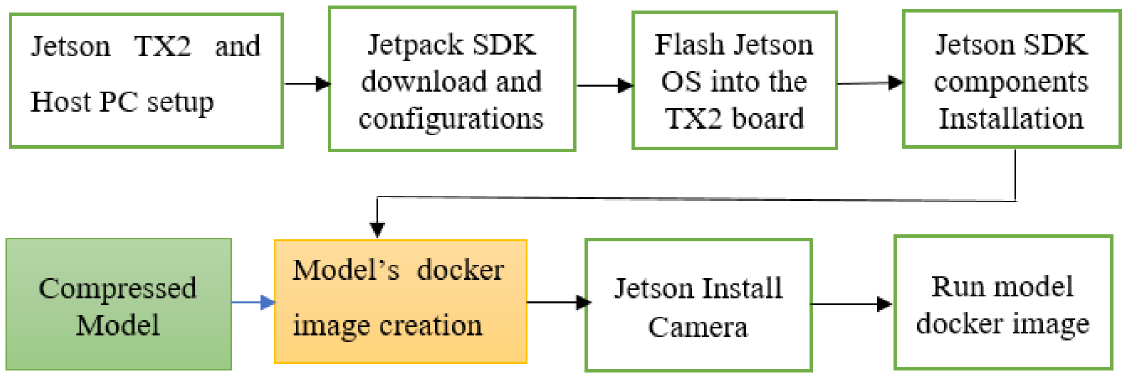

Jetson TX2 and Host PC setup
Jetpack SDK download and configurations
Flash Jetson OS into the TX2 board
Jetson SDK components Installation
Compressed Model
Model's docker image creation
Jetson Install Camera
Run model docker image

Figure 34 Flowgraph showing the steps involved in the deployment of the trained model on the edge device

As can be seen in Figure 32, the Jetson board is first set up with a host PC running Ubuntu 18.04 operating system (OS) and then the configurations and installation of packages follow. The green coloured box showed the compressed model obtained after the model has been trained and compressed, while the orange-coloured box shows the docker image creation step. The docker image of the compressed model is created after the component installations been performed. After this, a Universal Serial Bus (USB) camera is installed and the docker image is executed. The entire process took a lot of hours (about 24) to get them to work together.

4.8 Brief Overview of the Baseline CNN

As mentioned earlier, a CNN model was used as a baseline for comparison purposes; a brief overview thereof is shown in Table 7. The architecture is carefully chosen because it has a similar layout as that of the CapsNet model. It was trained on the GPU based HP workstation.

Table 7 Summary of the baseline CNN Architecture used for "Animal" classification

Baseline CNN Layer	**Details**
Input Image	Width = 28, height = 28, channels = 3
Convolution Layer 1	Filters = 256, kernel size = 5x5, stride = 1, Activation = ReLU, pooling = 2x2
Convolution Layer 2	Filters = 256, kernel size = 5x5, stride = 1, Activation = ReLU, pooling = 2x2
Convolution Layer 3	Filters = 128, kernel size = 5x5, stride = 1, Activation = ReLU, pooling = 2x2
Decoder Fully connected layer 1	328, ReLU
Decoder Fully connected layer 2	192, ReLU
Output layer	11, softmax

5 Results and Analysis

5.1 Model Training Results analysis

Having trained the models with several configurations (hyperparameters), a number of results and corresponding plots were obtained; they are analysed in the following subsections, each subsection focusing on different aspects of decisions made. Then finally, a comparison is made between the CapsNet and CNN models' performances.

5.1.1 Number of Routing Iterations in the Routing by Agreement Algorithm

This subsection focuses on the results obtained which inform the decision on the best number of routing iterations to use in the entire CapsNet model training.

Table 8 Results obtained from training a CapsNet with different algorithms for the MNIST Dataset.

Serial Number	Number of routing iterations	Output capsule loss (Training)	Output capsule loss (Validation)	Output capsule accuracy (Training)	Output capsule accuracy (Validation)
1	1	0.0130	0.0127	0.9914	0.9924
2	2	0.0139	0.0127	0.9920	0.9915
3	3	0.0120	0.0121	0.9924	0.9919
4	4	0.0121	0.0116	0.9920	0.9915
5	5	0.0121	0.0137	0.991	0.9889

Judging from the result presented in Table 8, it is not straightforward to decide which number of routing iterations is best to be used in the dynamic routing by agreement algorithms as their performances are not very different from each other. The decision was made to use 3 routing iterations, which is explained in what follows. An insight is to look at the validation loss; from here we can see that using 1, 2 and 3 routing iterations gives the same result, so why use 3? The reason is because 3 routing iterations give a better training accuracy than 1 and 2. A case can also be made for why 3 and 4 routing iterations were not used, this is because there is no increase in accuracy gained using 4 and 5 routing iterations although they have a lower validation loss.

5.1.2 Result of Two-Class Animal

Now that we have chosen the number of routing iterations, a model with 2 classes was developed to further explore the effect of some hyperparameters that we shall eventually justify. Subsections here explore and show the results of these studies.

Before we go into exploring the results, the structure used in denoting the hyperparameters shall be explained. Since, there are quite a number of graphs with the same structure but different values of hyperparameters, I decided to simplify the presentation by creating a structure to represent the results without having to repeat several lengthy sentences that are similar.

5.1.2.1 Training Configuration Structure

To simplify the results' presentation, a configuration structure that will be used to present the hyperparameters is presented below

Hyperparameter Configurations Structure (HCS) = (A, B, C, D, E, F).

where;

A = Number of images per class

B = Image resolution (Number of Pixels present in one dimension of the image)

C = Number of Channel (1 = Grayscale Image, 3 = Colour image)

D = Batch Size

E = Number of Epochs

F = Validation Split

Also, Tensorboard (a tool used to visualize tensorflow-trained models) does not explicitly give intuitive names to the graphs, so the captions present in the graphs are explained in Table 9. This is necessary in order to understand the graphs present in this chapter.

Table 9 Explanations of the captions used in the graphs generated by Tensorboard

Name (y-axis) as can be seen on graphs	**Meaning**
“Output_capsule_acc”	The capsule accuracy
“Output_capsule_loss”	The capsule loss
“val_output_capsule_acc”	The validation capsule accuracy
“val_output_capsule_loss”	The validation capsule loss
“epoch_acc”	The training accuracy of the CNN model
“epoch_loss”	The training loss of the CNN model
“epoch_val_acc”	The validation accuracy of the CNN model
“epoch_val_loss”	The validation loss of the CNN model

Again, it is important to note that for all graphs presented in this section, the x-axis denotes epoch (i.e. number of times the entire dataset is passed through the network during training). The Tensorboard tool has a limitation in displaying it properly. And lastly, the Tensorboard tool does not necessarily display the entire graph, it only displays the area of interest (sometimes, it may cover the entire graph and sometimes it may not).

5.1.2.2 Image Resolution

Now that the structure of the HCS has been defined, we shall start looking at the effects of resolution (width and height) of the images used for training.

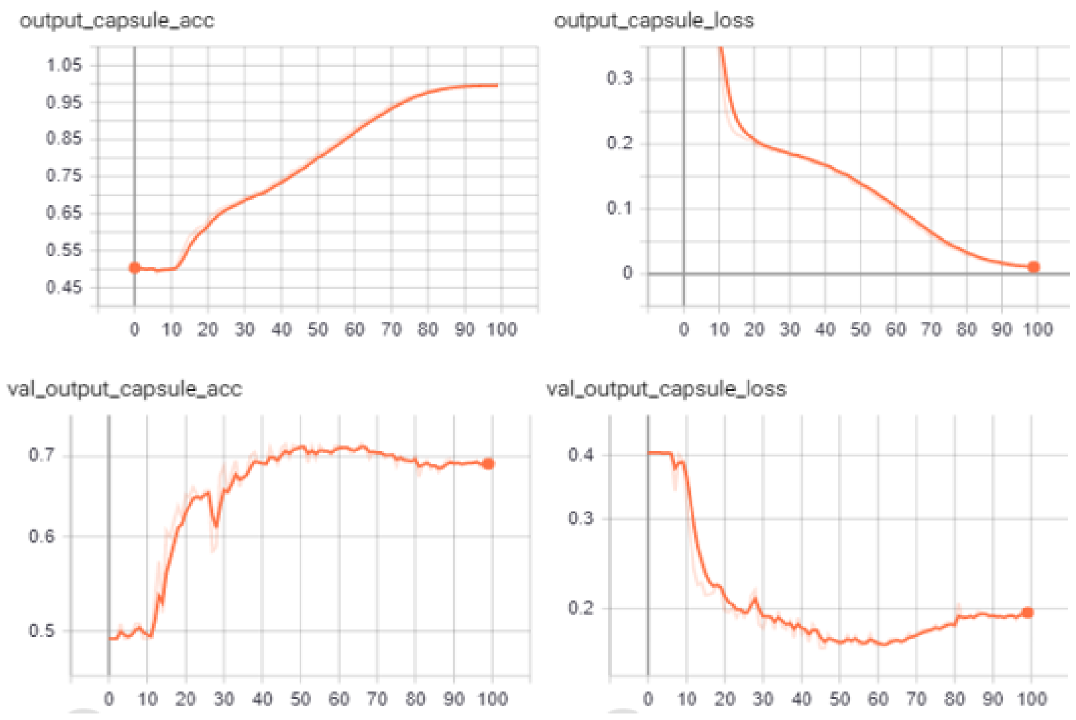


Figure 35 Part 1 results of using large and small image resolution with HCS = (1000, 28, 3, 32, 100, 0.2), with channel resolution = 28

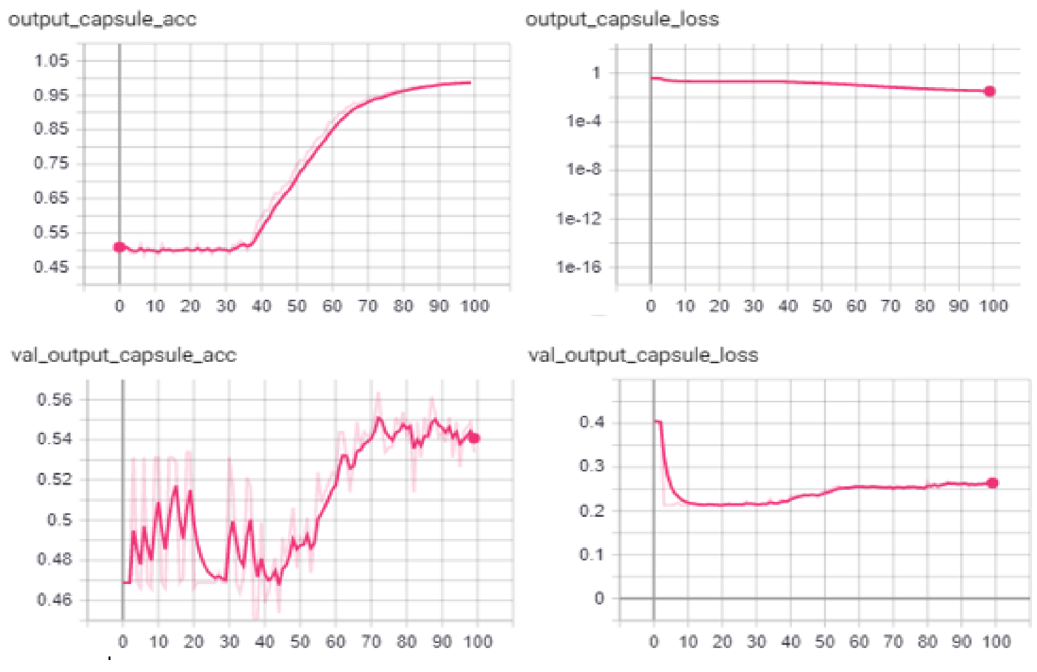


Figure 36 Part 2 results of using large and small image resolution with HCS = (1000, 80, 3, 32, 100, 0.2), with channel resolution = 80

Comparing figures 35 and 36, we can see that the output capsule accuracy in Figure 35 is about 75% at an epoch of 40 and its corresponding validation capsule accuracy is about 70% at 40 epochs. This cannot be said of Figure 36 at that same epoch and even beyond.

In Figure 36, the training accuracy gradually ramps up and settles at about 98%, while the validation accuracy remains at around 55% at 100 epoch. The stark difference between the training and validation accuracy shows that the model is overfitting and hence unable to generalize as the image resolution increases from 28 to 80. The losses (training and validation) in Figure 36 shows the validation loss rising at about 30 epochs, which shows that the model is about to start overfitting. The reason for this is not totally clear but several experiments ran on this model shows this same trend. A possible explanation could be that the higher resolution images has more information than the model can learn considering its vector size.

Further research in CapsNet in the future could explore the effect of the vector sizes and image resolution on CapsNet performance.

5.1.2.3 Red-Green-Blue (RGB) and Grayscale Images

Experimentation to show the effect of using grayscale images vis-à-vis RGB (colour images) for training is shown with tables and graphs. In Table 10, the results show that when the CapsNet model was trained with the CIFAR-10 ("Canadian Institute For Advanced Research" datasets with 10 classes) datasets, better performance were obtained by using RGB. RGB images gave a training accuracy of 73.06% while grayscale images gave 62.46%. The same trend is also shown in the validation accuracy. Looking at the losses, that of the RGB images are lower. Hence this justifies the reason for training the model with RGB images going forward in this study.

Table 10 Results of training the CapsNet with CIFAR-10 dataset in both RGB and grayscale

Image (channel) type	Output Capsule loss (Training)	Output Capsule loss (Validation)	Output Capsule Accuracy (Training)	Output Capsule Accuracy (Validation)
Grayscale	0.2849	0.3112	0.6246	0.5721
RGB	0.2263	0.2824	7306	0.6345

The CIFAR-10 datasets used with 3 routing iterations has 10 classes and 6000 images per class.

And as can also be seen in Figures 37 and 38, when RGB images are used, the performance of the model is better (lower loss, higher accuracy in both training and validation).

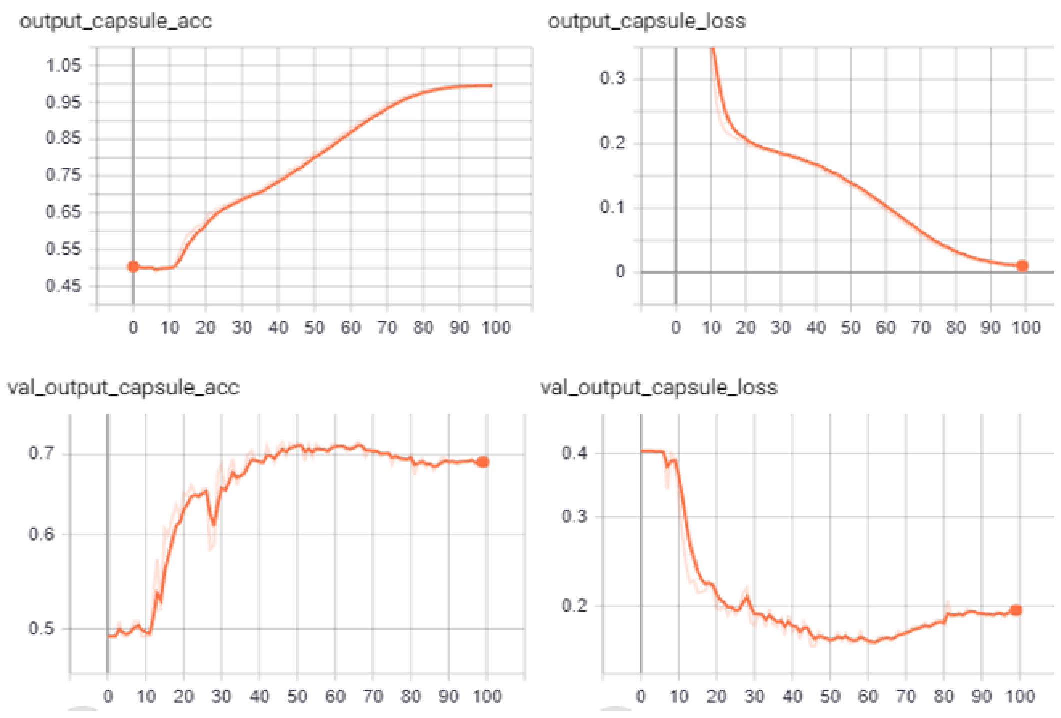


Figure 37 Part 2 results of using RGB and Grayscale images with HCS = (12501, 28, 3, 32, 100, 0.2), RGB images used

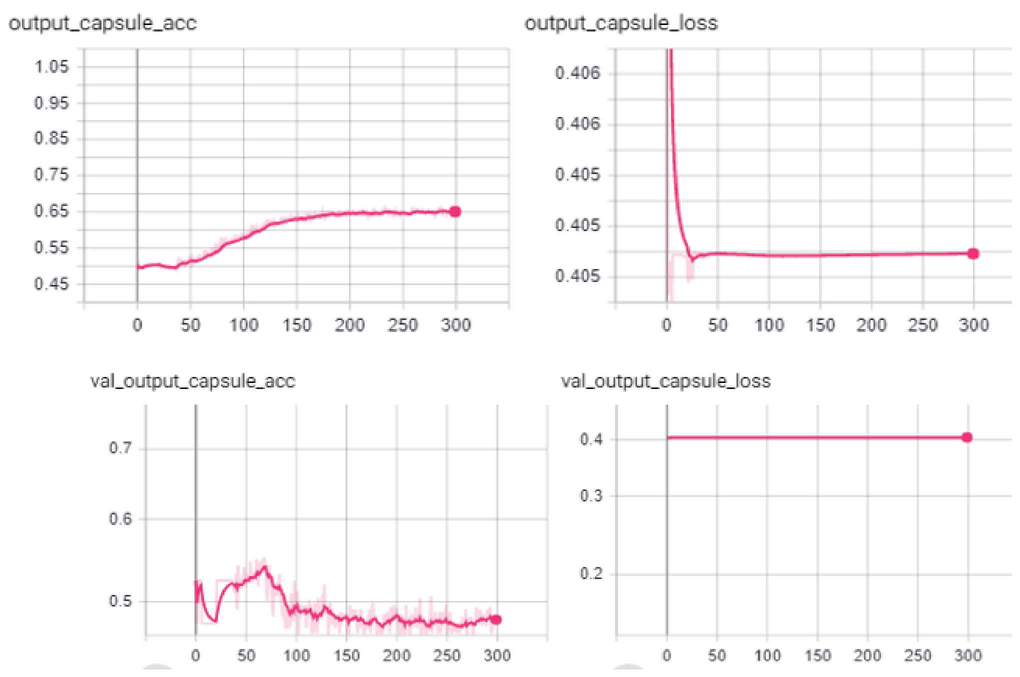


Figure 38 Part 2 results of using RGB and Grayscale images with HCS = (12501, 28, 1, 32, 100, 0.2), Grayscale images used

5.1.2.4 Batch Sizes Effects

Another study that was carried out in this work to explore the effect of having a different batch sizes on the CapsNet model; after several experiments to determine this, the results show that a higher batch size leads to better performance. This is evident in Figures 39 and 40. In Figure 39, 256 batch size was used, resulting in a capsule accuracy of 80% at around 96 epochs and the loss also kept decreasing, which is desired. However, in Figure 40, a batch size of 32 was used and it appears to have a better accuracy of close to 100%, but this is only overfitting, as the value of the loss tells us more about how good the model is.

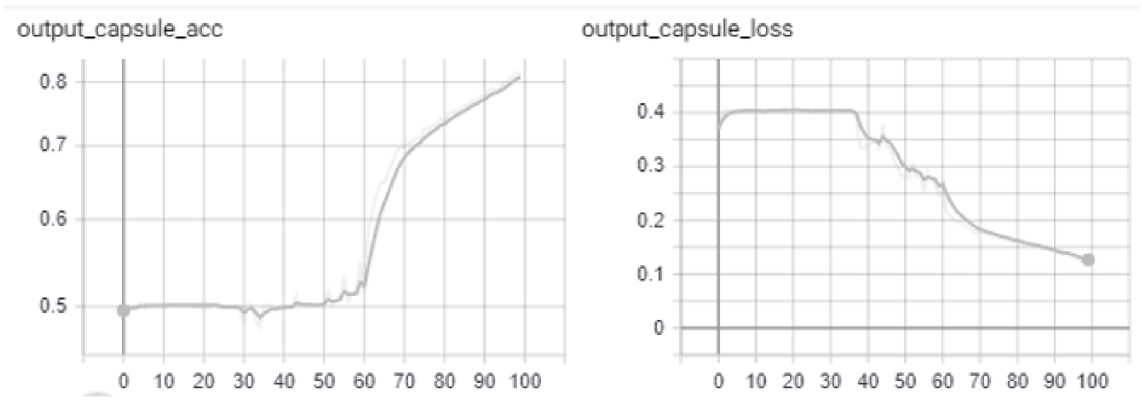


Figure 39 Part 1 results of using large and small batch size with HCS = (12501, 32, 3, 256, 100, 0.2), Batch size used = 256.

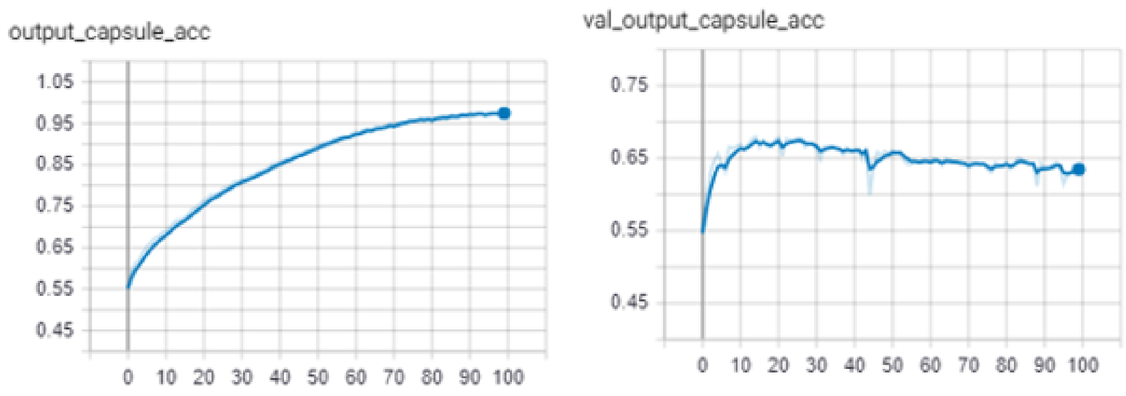


Figure 40 Part 2 results of using large and small batch size with HCS = (12501, 32, 3, 32, 100, 0.2), Batch size used = 32.

5.1.2.5 Validation split

Next, the effect of the validation split on the CapsNet model was also examined. The validation split is the percentage of the entire datasets used for validation. For example, a validation split of 0.4 means that 40% of the entire datasets is used for validation.

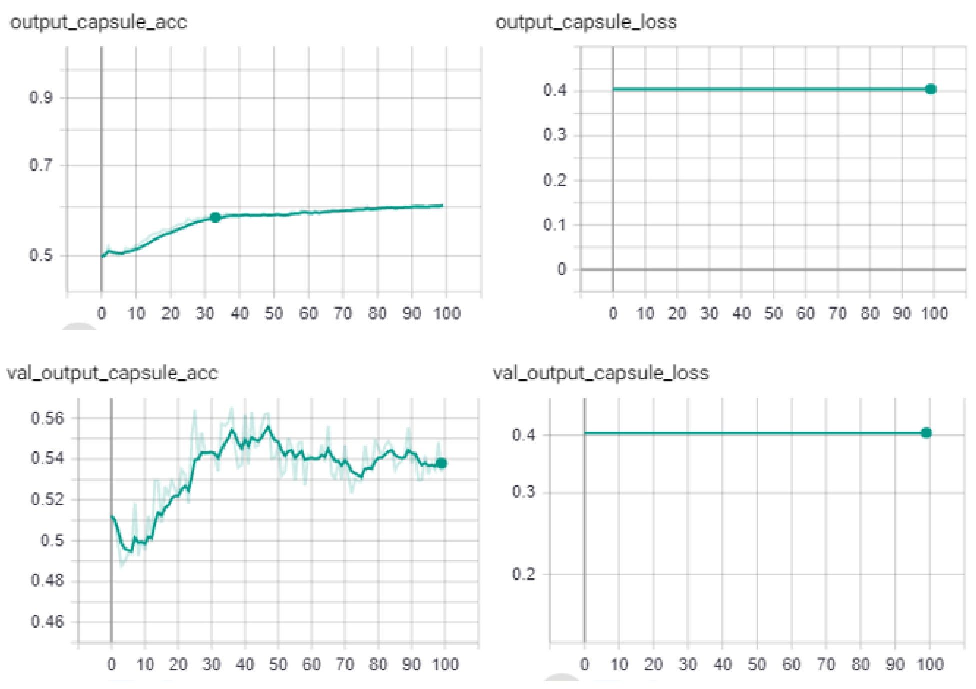


Figure 41 Part 1 results of using small and very small validation split, with HCS = (12501, 50, 3, 32, 100, 0.3), Validation split used = 0.3)

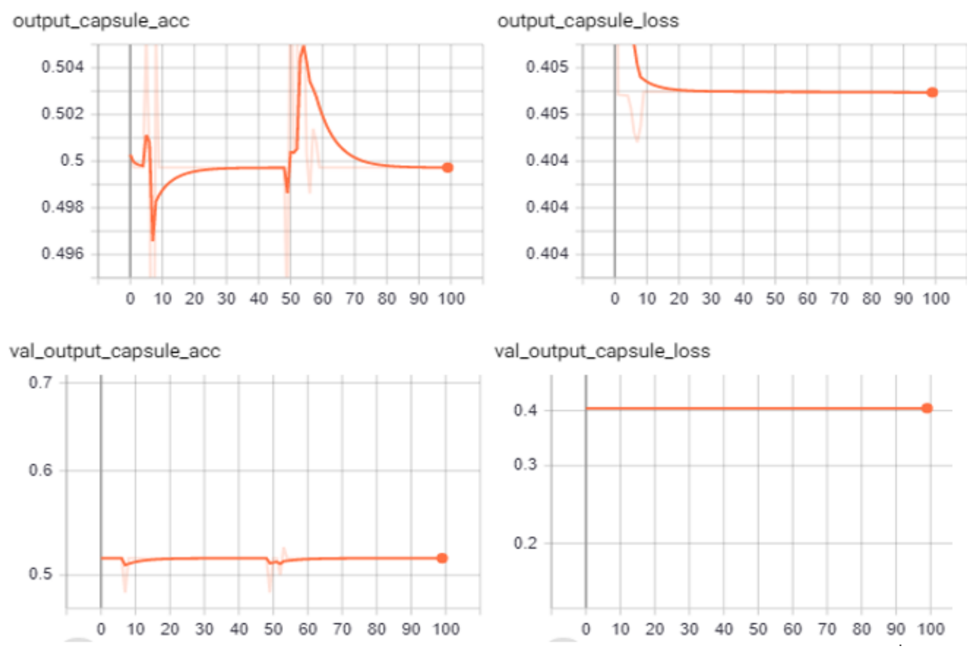


Figure 42 Part 2 results of using small and very small validation split, with HCS = (12501, 50, 3, 32, 100, 0.1), Validation split used = 0.1

In figures 41 and 42, a bigger validation split (0.3 as in Figure 41) shows better performance as opposed to a validation split of 0.1 (shown in Figure 42). The loss remains fairly constant in both Figures 41 and 42.

5.1.3 Result of Eleven-Class Animal model

Now that the results of the studies done in Section 5.1.2 have been analyzed, obtaining better results in subsequent CapsNet models requires that we use the better performing hyperparameters used in that section. Now, an 11-class CapsNet model is trained to be used for comparison with a CNN model, which is one of the major studies in this thesis. Unfortunately, the performance of this model was not recorded as it turns out that the datasets needs significantly more pre-processing than the scope of this thesis permit. So, decision was made to reduce the classes to 2 again, this time not for dog and cat images but for polar bear and deer images. Although the choice of animals is irrelevant.

Further experiments are now conducted on this latest CapsNet animal model for extra studies.

5.1.3.1 The use of two convolutional layers in the CapsNet Architecture

With the new CapsNet animal model, training was made with 4000 images, for each of the two animal classes as depicted in the first argument of the HCS value in Figure 43.

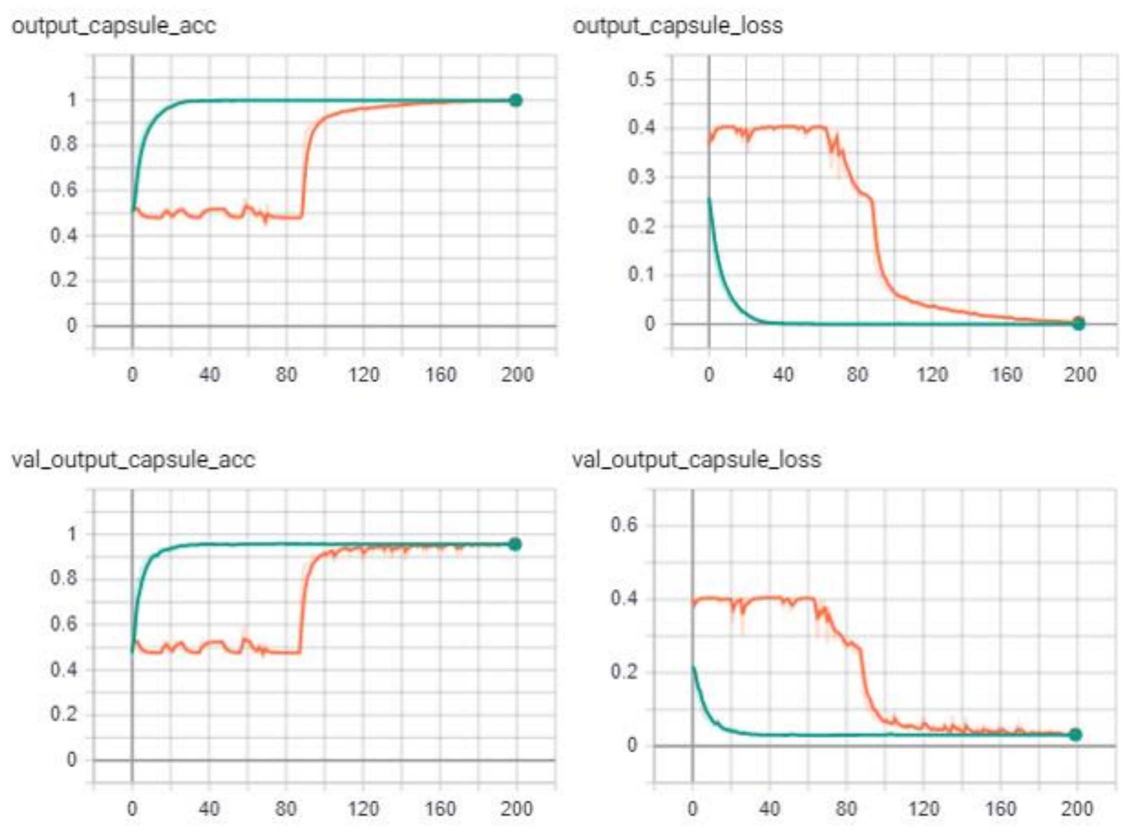


Figure 43 Results of using 2 convolutional layers in CapsNet architecture HCS = (4000, 28, 3, 256, 200, 0.35), Green curve = 2 convolutional layer, and orange curve = 1 convolutional layer

Figure 43 shows the recorded performances (capsule accuracy and loss) of using 2 convolutional layers and one convolutional layer in the CapsNet animal model. In Figure 43, the green curve represents the result of using two convolutional layers while the pink curve represents the result of 1 convolutional layer in the CapsNet. The effect of this study is conspicuous as it shows that the CapsNet model with 2 convolutional layers optimized faster than the model with 1 convolutional layer. It can be seen that the green curve already has a 94% accuracy after 20 epochs, while the orange curve took about 95 epochs to reach that same accuracy level. Although this study was not continued for even higher number of convolutional layer (because more compute power is needed than I had access to in order to do this study), the model is expected to converge faster with higher number of convolutional layers because the convolutional layers inherently extract more higher level features from the image before sending it to the first capsule layer in the network. Of course, there is an expectation that there will be a limit to the number of convolutional layers that can be added, a further study on capsule network can explore this.

5.1.3.2 CapsNet vs. CNN on small number of datasets

Figure 44 and 45 show the performances of a CapsNet and CNN models trained with 30 images obtained from a video-to-frame conversion of animal video clips of two animals. As already explained in Chapters 2 and 3 of this thesis, the pooling operation used in CNN makes it unable to extrapolate new orientation from a single image (it is viewpoint invariant), hence it has to be trained with all orientations of an image it is expected to recognize; this means a lot more data is required to train a CNN model than it is to train a CapsNet model of the same complexity. The CapsNet is able to extrapolate new viewpoints (viewpoint equivariance) because it already stored the instantiation parameters of the object in its vector and can keep the relationship between objects in an image due to the affine transformation matrix, as explained in Chapter 3 of this thesis.

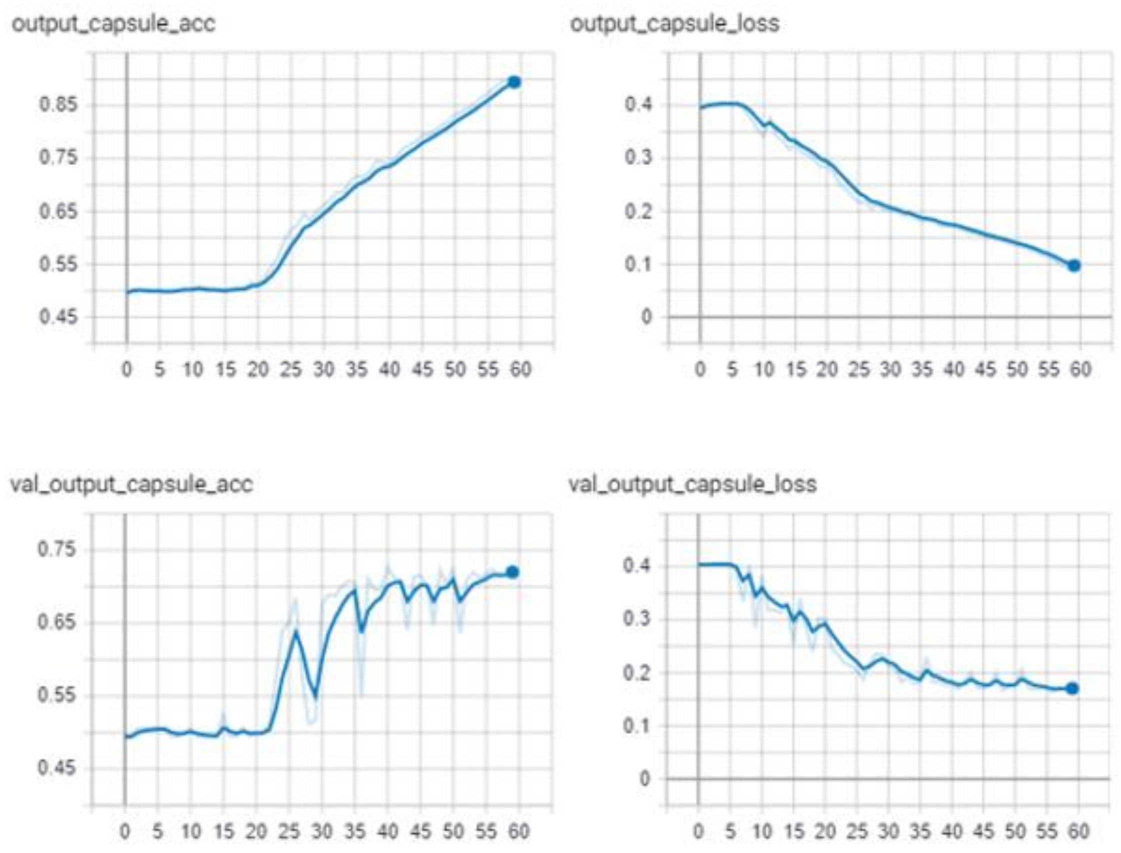


Figure 44 Part 1 of plots showing final model for comparison between CapsNet and CNN based on the amount of data used for training, with HCS = (30, 28, 3, 256, 60, 0.35), CapsNet shown here (see Figure 45 for CNN)

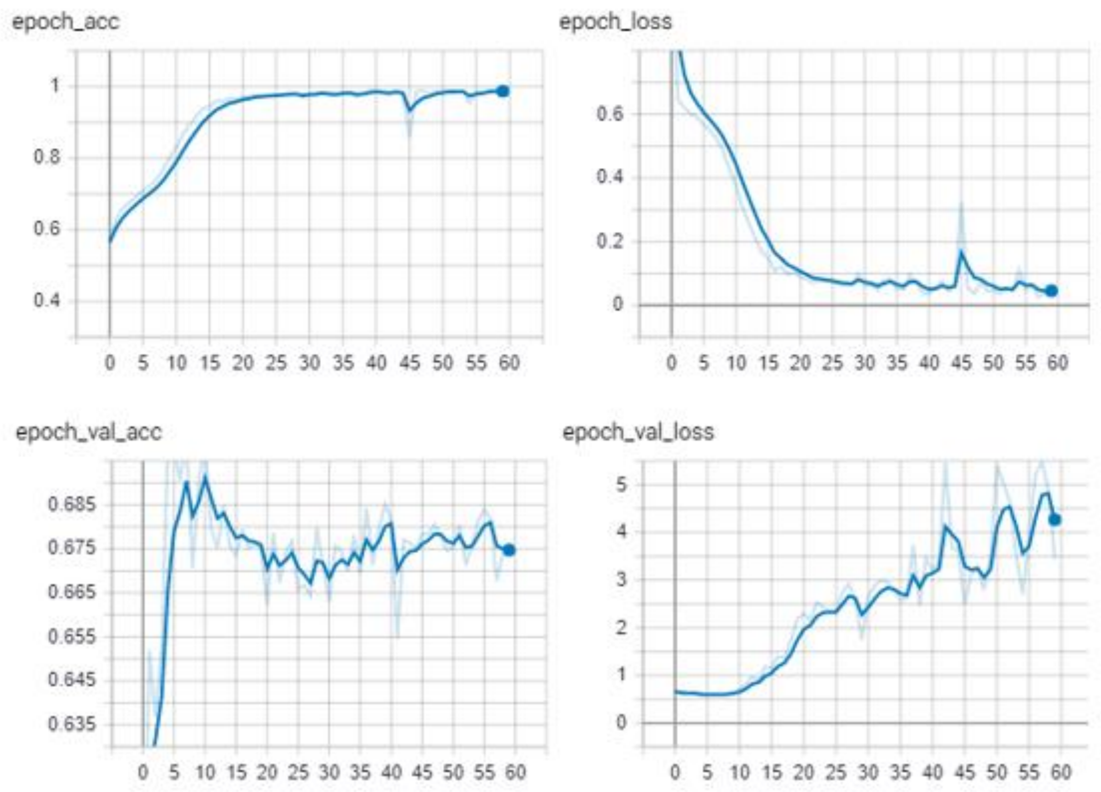


Figure 45 Part 2 of plots showing final model for comparison between CapsNet and CNN based on the amount of data used for training, with HCS = (30, 28, 3, 256, 60, 0.35), CNN shown here (see Figure 44 for CapsNet)

With both Figures 44 and 45 having the same parameters in their HCS, the only difference is the model used. Both models were trained for 60 epochs and they both had good training accuracies at the end of this epoch (99.01% and 99.12% for the CNN and CapsNet models, respectively); the catch is to look at the **validation metrics (especially the loss)**. The validation loss for the CNN model (Figure 45) is rather increasing, which shows that the model is not really learning but memorizing the training datasets (overfitting), but this is not the case in Figure 44 (CapsNet). Hence, this clearly shows that CapsNet performs better on smaller datasets as compared to an ‘equivalent’ CNN, making CapsNet a potential replacement for CNN in some applications where much data is not necessarily available at the time of training, or where the cost of obtaining enough datasets is prohibitive.

5.1.4 Model Minimization Results

After the model is trained, it is now compressed to be able to run on an edge device (with small memory size). Quantization was performed on this model to get it reduced to half its original sizes as shown in Table 11. The 32-bit weights and biases were converted to 16-bit. I initially set out to use the Google edge TPU, which would have made the model even smaller (8-bit), but a limitation on the operations supported by the TPU (it does not support 4-D tensors and some user defined functions in tensorflow) motivated the switch to a GPU-based NVIDIA Jetson TX2 board (16-bit). The size of the model (before and after compression) is shown in Table 11.

Table 11 Effect of applying the post training quantization method on models

S/N	**Model**	**Input Image Dimension (pixels)**	**Size before compression (MB)**	**Size After Compression (MB)**
1	CapsNet	28x28x1	27.9	13.8
2	CapsNet	28x28x3	34.5	17.25
3	CapsNet	32x32x3	97.8	48.9
4	CNN	28x28x1	13.9	6.95

5.2 Model Validation

Table 12 shows various validation accuracies gathered throughout the course of this study. That of MNIST datasets performed the best in all, while the CapsNet animal model performed better than the CNN animal model for reasons already explained in Section 5.1.3.1. It seems that both CapsNet and CNN animal models performed better than when the CIFAR-10 dataset is used. The reason is because the datasets used for the animal models are not as complex and diverse as the one found in CIFAR-10 dataset.

Table 12 Effect of applying the post training quantization method on models

Algorithm	Model	Validation Accuracy
CapsNet	MNIST	99.23%
CapsNet	CIFAR-10	63.45%
CapsNet	2-Class Animal Dataset	68.12%
CNN	2-Class Animal Dataset	67.50%

5.3 Hardware Performance and Stability Analysis

As indicated in Chapter 4 (Section 4.3) of this thesis, three different hardware platforms were used in the training of the models used for this study, and their performance and stability affected the pace of this research. Table 13 shows a brief analysis of their performances and stability.

The mean time before failure (MTBF) concept is used for this analysis, as shown in Equation 5.1.

$$MTBF = \frac{TWT - TBT}{NB} \tag{5.1}$$

where;

TWT = Total working time

TBT = Total breakdown time

NB = Number of Breakdowns

Table 13 Showing the training hardware performance metric

Hardware	**CapsNet Training time (100 epoch)**	**Accessibility**	**TWT**	**TBT**	**Nr. Of break downs**	**MTBF (hours)**
Core i7 PC with 8 GB memory	23 hours	Always	192	94	16	6.125
Cloud based CPU, with 16 GB memory: (Google-Colaboratory)	18 hours	Internet-dependent	192	48	48	3
HP workstation with NVIDIA GeForce GTX 1080 GPU	2 hours	Laboratory status	192	30	5	32.4

The performances in Table 13 is based on the same model configuration, with HCS = (12501, 28, 3, 32, 60, 0.2) monitored over a period of 192 hours. Overall, the HP workstation with GPU is the best in terms of speed and reliability, although with a high image dimension (like 80x80x3), it crashes and runs out of memory. This was a limiting factor in this study as also pointed out earlier. While the problem of intermittent disconnection can be solved by upgrading to the paid version of Google-Colaboratory, this service is only available for users from north America (United States and Canada) for now. The PC also was performing badly as it always becomes very hot and eventually restarted which terminated the simulations.

This chapter presented the experimental results obtained in this thesis and illustrated the value of CapsNet as compared to CNN for small datasets. Furthermore, compression of the model has been performed as a first step towards implementing CapsNet on a resource-constrained edge device. The next chapter summarizes the main points of the thesis and briefly outlines future work

6 Conclusion

6.1 Summary

The aim of this thesis is deemed to be largely fulfilled by implementing a CapsNet model than can be deployed to the edge. A rather detailed overview of machine learning and deep learning concepts have been presented as they are prime tools for object recognition. The state of the art in object recognition has also been presented. The CapsNet was finally presented as its understanding depends on the deep learning concepts. It was also presented in such a way that the promises that it brings as to building an image recognition system with minimal dataset was carefully introduced. This thesis has been implemented with the help of GPU-based workstation available at the university. The author's personal computer and cloud-based deep learning platforms were also used for the study. Various software packages and tools were used, particularly Tensorflow was used to program and perform the implementation.

In particular, the question of how to implement (or move) machine learning model from the cloud to the edge was explored and implemented due to various promises that the edge holds in the future of machine learning. Image recognition system that can work with minimal dataset, compressed and deployed to the edge was also explored and implemented thanks to the special properties of the recent CapsNet. Although we set out to implement the model on a TPU, this was not done because of limited support offered by the Coral range TPU board from Google, and thus CapsNet was implemented on and exploited the flexibility of and Nvidia Jetson TX2 board.

Various image processing techniques and data collection methods were used in the project. A central part of the study involved the tuning of the CapsNet model hyperparameters and exploration of its flexibility was done and presented. We found out how changing various hyperparameters could affect the performance of the CapsNet model, and a CNN of equivalent complexity was used as baseline to compare the results. Lastly, in the process of presenting the results, a novel method to label a model according to its configurations has been developed, i.e. the Hyperparameter Configuration Structure (HCS). This method makes it easy to label the results of a model instance in details without having to use lengthy sentences. Naturally, the HCS method can be expanded as desired to include other parameters that were not considered here.

With an Hyperparameter Configuration Structure (HCS) value of (30,28,3,256,60,0.35), a Capsule Network (CapsNet) was able to achieve a training accuracy of 99.12% at a validation accuracy of 73% and training loss of 0.1 at a validation loss of 0.17. An equivalent CNN achieved 99.01% and a validation accuracy of 67.5% and a training loss of 0.05 at a validation loss of 4.5 (severe overfitting). Both of the results shows the model has overfitted. But at an epoch of 35, the CapsNet model has both its training and validation accuracies at 70% while the CNN model instead has a training accuracy of 98% but a validation accuracy of 66.70%. Meaning the CNN overfits on small dataset regardless of the amount of epoch used but we can get a CapsNet that performs optimally at 70% accuracy.

In conclusion, we have seen that CapsNet model, although new, has its advantages and drawbacks. It can make image recognition with minimal dataset possible as well as permit compression and implementation on the edge. On the other hand, CapsNet has some challenges in its complexity and hence takes more time for training than the equivalent CNN (about 10 times longer).

1.1.1 Considerations with respect to the state of the art

Deep neural networks such as CNN is still considered the state of the art in object recognition [9]. And they certainly outperform CapsNet in object recognition. But CapsNet has shown a capability to be useful where limited amounts of datasets are available. The study in this thesis only compared CapsNet against an architecture of CNN that is comparable to the CapsNet structure. Several other architectures of CNN outperform CapsNet, but considering that CapsNet is relatively new and much research has not been done on it as on CNN, it has the potential of becoming even more powerful and actually give a good competition to CNNs that are very good at image recognition, especially in areas where there are not large amounts of datasets.

6.2 Future work

The study presented in this thesis has its limitations, as for all the training done, none was performed up to 100 epochs. This is because of the limitation in hardware resources accessible for training. Also, the GPU-based workstation that was used for most of the training did not have enough memory to handle higher image resolutions, hence, image resolution beyond 80x80 pixels were not studied.

Further research into CapsNet can be directed towards the study of its behaviour with very large image resolutions (like 1800x2000 pixels) to assess more thoroughly what could be learnt from it, and at the same time, studying the effects of changing the vector dimension. Also, using an object detection algorithm alongside the CapsNet will be interesting to explore. Further studies can also be directed towards creating a platform that can make deep leaning models be easily deployed at the edge. Lastly, as suggested in Chapter 5, studies can be directed towards finding the limit of the number of convolutional layers that can be embedded into the CapsNet and the effects thereof.

References

[1] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015, doi: 10.1038/nature14539.

[2] X. Du, Y. Cai, S. Wang, and L. Zhang, “Overview of deep learning,” in *2016 31st Youth Academic Annual Conference of Chinese Association of Automation (YAC)*, Nov. 2016, pp. 159–164, doi: 10.1109/YAC.2016.7804882.

[3] Hassan, S. Gillani, E. Ahmed, I. Yaqoob, and M. Imran, “The Role of Edge Computing in Internet of Things,” *IEEE Communications Magazine*, vol. 56, no. 11, pp. 110–115, Nov. 2018, doi: 10.1109/MCOM.2018.1700906.

[4] Z.-Q. Zhao, P. Zheng, S. Xu, and X. Wu, “Object Detection with Deep Learning: A Review,” *arXiv:1807.05511 [cs]*, Apr. 2019, Accessed: Dec. 07, 2020. [Online]. Available: http://arxiv.org/abs/1807.05511.

[5] S. Sabour, N. Frosst, and G. E. Hinton, “Dynamic Routing Between Capsules,” *arXiv:1710.09829 [cs]*, Nov. 2017, Accessed: Feb. 14, 2020. [Online]. Available: http://arxiv.org/abs/1710.09829.

[6] X. Zhou, W. Gong, W. Fu, and F. Du, “Application of deep learning in object detection,” in *2017 IEEE/ACIS 16th International Conference on Computer and Information Science (ICIS)*, May 2017, pp. 631–634, doi: 10.1109/ICIS.2017.7960069.

[7] C. Chen, J. Huang, C. Pan, and X. Yuan, “Military Image Scene Recognition Based on CNN and Semantic Information,” in *2018 3rd International Conference on Mechanical, Control and Computer Engineering (ICMCCE)*, Sep. 2018, pp. 573–577, doi: 10.1109/ICMCCE.2018.00126.

[8] A. Khan, A. Sohail, U. Zahoora, and A. S. Qureshi, “A survey of the recent architectures of deep convolutional neural networks,” *Artif Intell Rev*, vol. 53, no. 8, pp. 5455–5516, Dec. 2020, doi: 10.1007/s10462-020-09825-6.

[9] H. Huang, Q. Li, and D. Zhang, “Deep learning based image recognition for crack and leakage defects of metro shield tunnel,” *Tunnelling and Underground Space Technology*, vol. 77, pp. 166–176, Jul. 2018, doi: 10.1016/j.tust.2018.04.002.

[10] K. He, X. Zhang, S. Ren, and J. Sun, “Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 37, no. 9, pp. 1904–1916, Sep. 2015, doi: 10.1109/TPAMI.2015.2389824.

[11] R. Girshick, “Fast R-CNN,” 2015, pp. 1440–1448, Accessed: Dec. 25, 2020. [Online]. Available: https://openaccess.thecvf.com/content_iccv_2015/html/Girshick_Fast_R-CNN_ICCV_2015_paper.html.

[12] S. Ren, K. He, R. Girshick, and J. Sun, “Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 39, no. 6, pp. 1137–1149, Jun. 2017, doi: 10.1109/TPAMI.2016.2577031.

[13] L. Zhang, L. Lin, X. Liang, and K. He, “Is Faster R-CNN Doing Well for Pedestrian Detection?,” in *Computer Vision – ECCV 2016*, Cham, 2016, pp. 443–457, doi: 10.1007/978-3-319-46475-6_28.

[14] J. Deng, W. Dong, R. Socher, L. Li, Kai Li, and Li Fei-Fei, “ImageNet: A large-scale hierarchical image database,” in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, Jun. 2009, pp. 248–255, doi: 10.1109/CVPR.2009.5206848.

[15] O. Russakovsky *et al.*, “ImageNet Large Scale Visual Recognition Challenge,” *International Journal of Computer Vision*, vol. 115, Sep. 2014, doi: 10.1007/s11263-015-0816-y.

[16] M. Everingham, L. Van Gool, C. Williams, J. Winn, and A. Zisserman, “The Pascal Visual Object Classes (VOC) challenge,” *International Journal of Computer Vision*, vol. 88, pp. 303–338, Jun. 2010, doi: 10.1007/s11263-009-0275-4.

[17] T.-Y. Lin *et al.*, “Microsoft COCO: Common Objects in Context,” in *Computer Vision – ECCV 2014*, Cham, 2014, pp. 740–755, doi: 10.1007/978-3-319-10602-1_48.

[18] Z. Dan and C. Xu, “The Recognition of Handwritten Digits Based on BP Neural Network and the Implementation on Android,” in *2013 Third International Conference on Intelligent System Design and Engineering Applications*, Jan. 2013, pp. 1498–1501, doi: 10.1109/ISDEA.2012.359.

[19] P. Louridas and C. Ebert, “Machine Learning,” *IEEE Software*, vol. 33, no. 5, pp. 110–115, Sep. 2016, doi: 10.1109/MS.2016.114.

[20] M. Usama *et al.*, “Unsupervised Machine Learning for Networking: Techniques, Applications and Research Challenges,” *IEEE Access*, vol. 7, pp. 65579–65615, 2019, doi: 10.1109/ACCESS.2019.2916648.

[21] K. Janocha and W. M. Czarnecki, “On Loss Functions for Deep Neural Networks in Classification,” *SI*, vol. 1/2016, 2017, doi: 10.4467/20838476SI.16.004.6185.

[22] T. Watanabe and H. Iima, “Nonlinear Optimization Method Based on Stochastic Gradient Descent for Fast Convergence,” in *2018 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, Oct. 2018, pp. 4198–4203, doi: 10.1109/SMC.2018.00711.

[23] X. Ying, “An Overview of Overfitting and its Solutions,” *Journal of Physics: Conference Series*, vol. 1168, p. 022022, Feb. 2019, doi: 10.1088/1742-6596/1168/2/022022.

[24] S. Albawi, T. A. Mohammed, and S. Al-Zawi, “Understanding of a convolutional neural network,” in *2017 International Conference on Engineering and Technology (ICET)*, Aug. 2017, pp. 1–6, doi: 10.1109/ICEngTechnol.2017.8308186.

[25] A. Deshpande, “A Beginner’s Guide To Understanding Convolutional Neural Networks.” https://adeshpande3.github.io/A-Beginner's-Guide-To-Understanding-Convolutional-Neural-Networks/ (accessed Dec. 27, 2020).

[26] C. Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall, “Activation Functions: Comparison of trends in Practice and Research for Deep Learning,” *arXiv:1811.03378 [cs]*, Nov. 2018, Accessed: Dec. 27, 2020. [Online]. Available: http://arxiv.org/abs/1811.03378.

[27] P. Ramachandran, B. Zoph, and Q. Le, “Searching for Activation Functions,” 2018, Accessed: Dec. 27, 2020. [Online]. Available: https://arxiv.org/pdf/1710.05941.pdf.

[28] G. Lin and W. Shen, “Research on convolutional neural network based on improved Relu piecewise activation function,” *Procedia Computer Science*, vol. 131, pp. 977–984, Jan. 2018, doi: 10.1016/j.procs.2018.04.239.

[29] A. Muñío-Gracia, J. Fernández-Berni, R. Carmona-Galán, and Á. Rodríguez-Vázquez, “Impact of CNNs Pooling Layer Implementation on FPGAs Accelerator Design,” in *Proceedings of the 13th International Conference on Distributed Smart Cameras*, New York, NY, USA, Sep. 2019, pp. 1–2, doi: 10.1145/3349801.3357130.

[30] J. Naranjo-Torres, M. Mora, R. Hernández-García, R. J. Barrientos, C. Fredes, and A. Valenzuela, “A Review of Convolutional Neural Network Applied to Fruit

Image Processing," *Applied Sciences*, vol. 10, no. 10, Art. no. 10, Jan. 2020, doi: 10.3390/app10103443.

[31] J. S. Bridle, "Probabilistic Interpretation of Feedforward Classification Network Outputs, with Relationships to Statistical Pattern Recognition," in *Neurocomputing*, Berlin, Heidelberg, 1990, pp. 227–236, doi: 10.1007/978-3-642-76153-9_28.

[32] R. Karim, "Illustrated: 10 CNN Architectures," *Medium*, Nov. 28, 2020. https://towardsdatascience.com/illustrated-10-cnn-architectures-95d78ace614d (accessed Dec. 27, 2020).

[33] S. Arya and R. Singh, "A Comparative Study of CNN and AlexNet for Detection of Disease in Potato and Mango leaf," in *2019 International Conference on Issues and Challenges in Intelligent Computing Techniques (ICICT)*, Sep. 2019, vol. 1, pp. 1–6, doi: 10.1109/ICICT46931.2019.8977648.

[34] A. G. Howard *et al.*, "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications," *arXiv:1704.04861 [cs]*, Apr. 2017, Accessed: Dec. 27, 2020. [Online]. Available: http://arxiv.org/abs/1704.04861.

[35] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, pp. 84–90, May 2017, doi: 10.1145/3065386.

[36] Z. Liu, Y. Xu, and F. Dong, "L1-L2 Spatial Adaptive Regularization Method for Electrical Tomography," in *2019 Chinese Control Conference (CCC)*, Jul. 2019, pp. 3346–3351, doi: 10.23919/ChiCC.2019.8865488.

[37] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," *arXiv:1510.00149 [cs]*, Feb. 2016, Accessed: Dec. 27, 2020. [Online]. Available: http://arxiv.org/abs/1510.00149.

[38] S. Wang *et al.*, "C-LSTM: Enabling Efficient LSTM using Structured Compression Techniques on FPGAs," in *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, New York, NY, USA, Feb. 2018, pp. 11–20, doi: 10.1145/3174243.3174253.

[39] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both Weights and Connections for Efficient Neural Networks," *arXiv:1506.02626 [cs]*, Oct. 2015, Accessed: Feb. 22, 2020. [Online]. Available: http://arxiv.org/abs/1506.02626.

[40] M. Pechyonkin, “Understanding Hinton’s Capsule Networks. Part 4. CapsNet Architecture.,” *Max Pechyonkin*. https://pechyonkin.me/capsules-4/ (accessed Dec. 25, 2020).

[41] B. Jia and Q. Huang, “DE-CapsNet: A Diverse Enhanced Capsule Network with Disperse Dynamic Routing,” *Applied Sciences*, vol. 10, no. 3, Art. no. 3, Jan. 2020, doi: 10.3390/app10030884.

[42] D. Amara, *Novel Deep Learning Model for Traffic Sign Detection Using Capsule Networks*. 2018.

[43] M. Kwabena Patrick, A. Felix Adekoya, A. Abra Mighty, and B. Y. Edward, “Capsule Networks – A survey,” *Journal of King Saud University - Computer and Information Sciences*, Sep. 2019, doi: 10.1016/j.jksuci.2019.09.014.

[44] J. Su, D. V. Vargas, and K. Sakurai, “Attacking convolutional neural network using differential evolution,” *IPSJ Transactions on Computer Vision and Applications*, vol. 11, no. 1, p. 1, Feb. 2019, doi: 10.1186/s41074-019-0053-3.

[45] J. E. Lenssen, M. Fey, and P. Libuschewski, “Group Equivariant Capsule Networks,” *arXiv:1806.05086 [cs]*, Oct. 2018, Accessed: Dec. 28, 2020. [Online]. Available: http://arxiv.org/abs/1806.05086.

[46] “Grayscale to RGB Conversion - Tutorialspoint.” https://www.tutorialspoint.com/dip/grayscale_to_rgb_conversion.htm (accessed Mar. 01, 2020).

[47] A. Nasonov, K. Chesnakov, and A. Krylov, “Convolutional neural networks based image resampling with noisy training set,” in *2016 IEEE 13th International Conference on Signal Processing (ICSP)*, Nov. 2016, pp. 62–66, doi: 10.1109/ICSP.2016.7877797.

Appendix 1 - Code for image processing

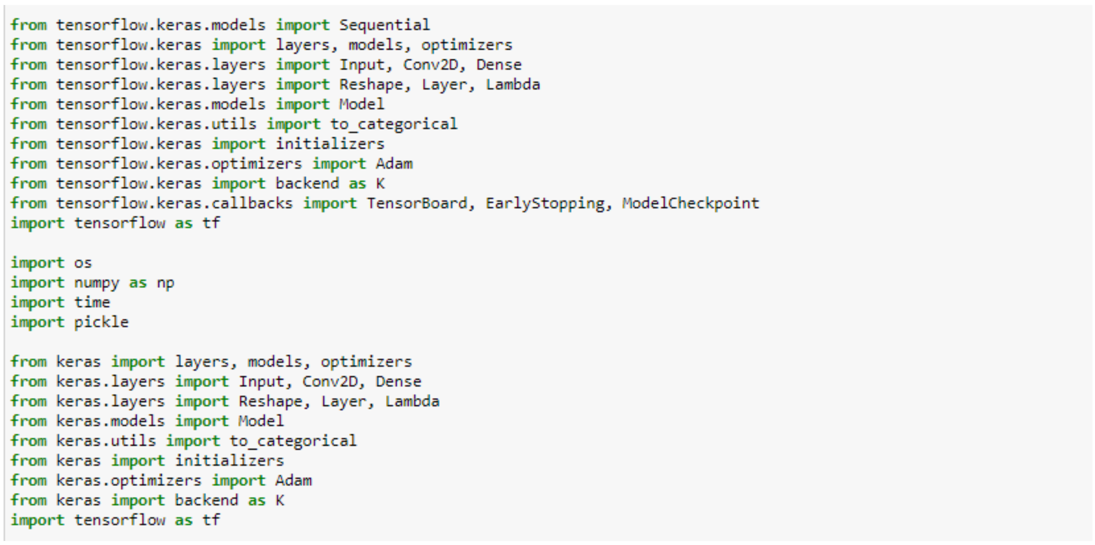
```
import numpy as np                #To do some array operations
import matplotlib.pyplot as plt #To do some plotings
import os                         #TO iterate through the directories
and join paths
import cv2                        # To do some image operations



######configuratiosn##########
n_img_per_class = 12502
IMG_SIZE = 28
n_channel = 3                      #...1 for grayscale, 3 for color
image_type = "RGB"
class_a = 'a_' + str(n_img_per_class)
class_b = 'b_' + str(n_img_per_class)
class_c = 'c_' + str(n_img_per_class)
class_d = 'd_' + str(n_img_per_class)
class_e = 'e_' + str(n_img_per_class)
class_f = 'f_' + str(n_img_per_class)
class_g = 'g_' + str(n_img_per_class)
class_h = 'h_' + str(n_img_per_class)
class_i = 'i_' + str(n_img_per_class)
class_j = 'j_' + str(n_img_per_class)
class_k = 'k_' + str(n_img_per_class)


CATEGORIES = [class_a, class_b, class_c, class_d, class_e, class_f, cl
ass_g, class_h, class_i, class_j, class_k]#, class_l, class_m, class_
n, class_o, class_p, class_q, class_r, class_s, class_t]
DATADIR = "Z:\Dataset\THESIS\Video_convert"


training_data = []

def create_training_data():
    for category in CATEGORIES:
        path = os.path.join(DATADIR, category) # #path to dog or cat
directories
        #convert the class names to a number
        class_num = CATEGORIES.index(category)
        print(class_num)
        for img in os.listdir(path):
            try:
                img_array = cv2.imread(os.path.join(path,img), cv2.IM
READ_COLOR) # for RGB
                #img_array = cv2.imread(os.path.join(path,img), cv2
.IMREAD_GRAYSCALE)       # for grayscale
                new_array = cv2.resize(img_array, (IMG_SIZE, IMG_SIZE
))  # RESIZE THE IMAGE
                training_data.append([new_array, class_num])
            except Exception as e:
                #print(e)
                pass

create_training_data()
```

```
import random

random.shuffle(training_data)

#creating list for storing the pictures and their labels
X = [] #features (image)
Y = [] #labels (cat or dog, represented with 0 and 1)

for image, label in training_data:
    X.append(image)
    Y.append(label)

X = np.array(X).reshape(-1, IMG_SIZE, IMG_SIZE, n_channel) #the -1 ar
gument means anything... that is it can be any value


#So we don't have to be re processing our dataset any time we want
to run the network, we are going to import pickle to save it.
#we can as well use numpy.save()

import pickle
X_dataset_name = 'X_Datasets_' +  str(n_img_per_class+7) + '_images_
per_class_' + str(image_type) + '_' + str(IMG_SIZE) + 'x' + str(IMG_
SIZE) + '_.pickle'
Y_dataset_name = 'Y_Datasets_' +  str(n_img_per_class+7) + '_images_
per_class_' + str(image_type) + '_' + str(IMG_SIZE) + 'x' + str(IMG_
SIZE) + '_.pickle'

X_store =  r"C:\Users\EliteBook\Anaconda3\envs\env1\prepared_datase
t\RGB\{}".format(X_dataset_name)
Y_store =  r"C:\Users\EliteBook\Anaconda3\envs\env1\prepared_datase
t\RGB\{}".format(Y_dataset_name)

pickle_out = open(X_store, "wb")
pickle.dump(X, pickle_out)
pickle_out.close()

pickle_out = open(Y_store, "wb")
pickle.dump(Y, pickle_out)
pickle_out.close()

#END
```

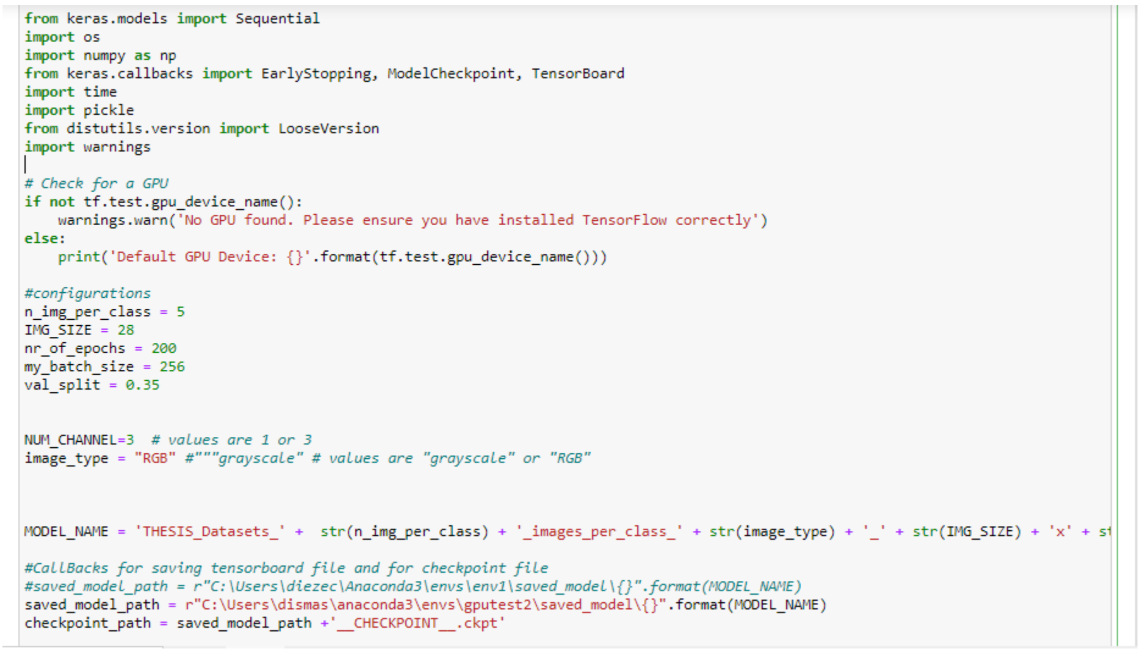
Appendix 2 – Code for web-scrapping

```
import os
import requests
from bs4 import BeautifulSoup

Google_Image = \
    'https://www.google.com/search?site=&tbm=isch&source=hp&biw=187
3&bih=990&'


System = {
    'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleW
ebKit/537.36 (KHTML, like Gecko) Chrome/85.0.4183.83 Safari/537.36'
,
    'Accept': 'text/html,application/xhtml+xml,application/xml;q=0.
9,*/*;q=0.8',
    'Accept-Charset': 'ISO-8859-1,utf-8;q=0.7,*;q=0.3',
    'Accept-Encoding': 'none',
    'Accept-Language': 'en-US,en;q=0.8',
    'Connection': 'keep-alive',
}

Destination = 'DISMAS_THESIS_IMAGES'

def main():
    if not os.path.exists(Destination):
        os.mkdir(Destination)
    Get_Animal_Images()

def Get_Animal_Images():
    data = input('Type the name of the animal you are looking for:
')
    quantity_of_animal_images = int(input('Type the amount of image
s you want : '))

    print('Looking....')

    _url = Google_Image + 'q=' + data


    response = requests.get(_url, headers=System)
    html = response.text


    b_soup = BeautifulSoup(html, 'html.parser')
    outcome = b_soup.findAll('img', {'class': 'rg_i Q4LuWd'})


    add_up = 0
    quantity = 0
    url= []
    for outs in outcome:
        try:
            link = outs['data-src']
            if (add_up >= 40):
                url.append(link)
                quantity = quantity + 1
            else:
                pass
            add_up = add_up + 1
            if (quantity >= quantity_of_animal_images):
                break

        except KeyError:
            continue

    print(f'Found {len(url)} Animal Images')
    print('Downloading...')
    for i, url in enumerate(url):

        response = requests.get(url)

        animalname = Image_Folder + '/' + data + str(i+1) + '.jpg'
        with open(animalname, 'wb') as file:
            file.write(response.content)

    print('Downloaded!')
    print(quantity)
    print(outcome)

if __name__ == '__main__':
    main()
```

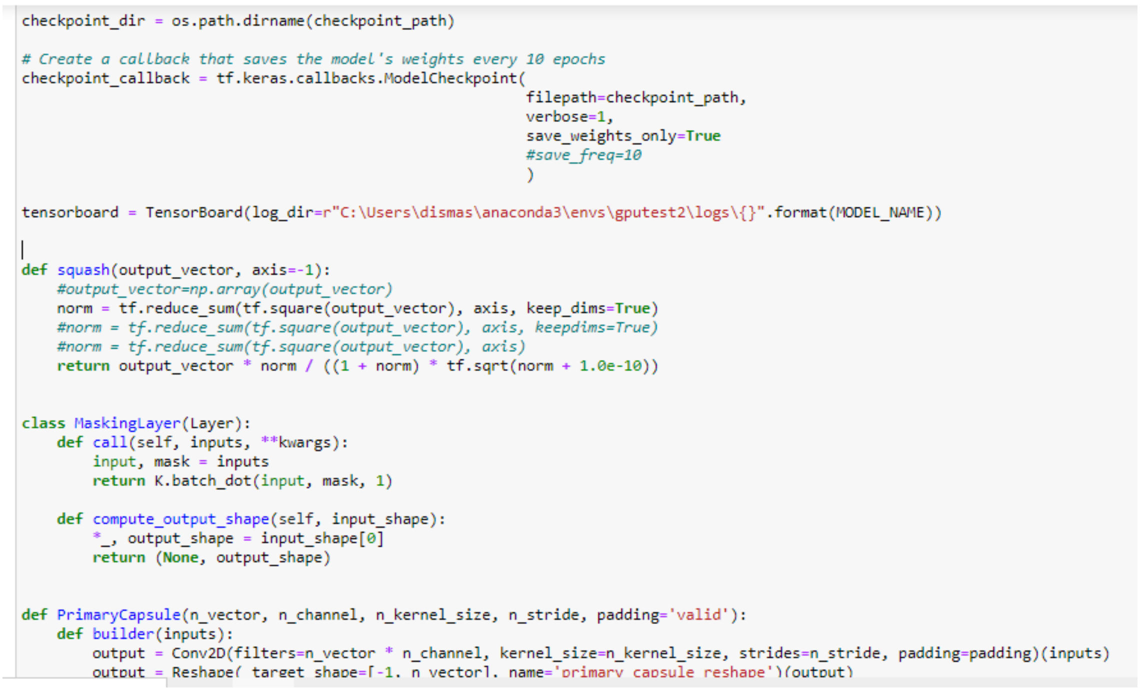
Appendix 3 – Main code



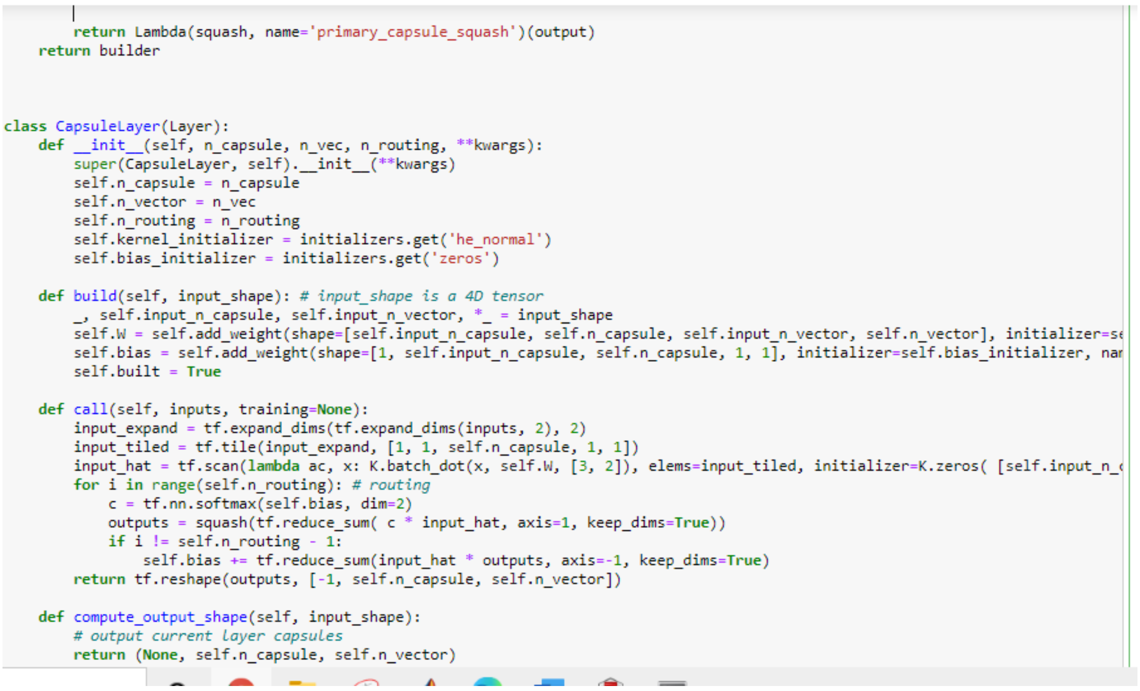
```
from tensorflow.keras.models import Sequential
from tensorflow.keras import layers, models, optimizers
from tensorflow.keras.layers import Input, Conv2D, Dense
from tensorflow.keras.layers import Reshape, Layer, Lambda
from tensorflow.keras.models import Model
from tensorflow.keras.utils import to_categorical
from tensorflow.keras import initializers
from tensorflow.keras.optimizers import Adam
from tensorflow.keras import backend as K
from tensorflow.keras.callbacks import TensorBoard, EarlyStopping, ModelCheckpoint
import tensorflow as tf

import os
import numpy as np
import time
import pickle

from keras import layers, models, optimizers
from keras.layers import Input, Conv2D, Dense
from keras.layers import Reshape, Layer, Lambda
from keras.models import Model
from keras.utils import to_categorical
from keras import initializers
from keras.optimizers import Adam
from keras import backend as K
import tensorflow as tf
```



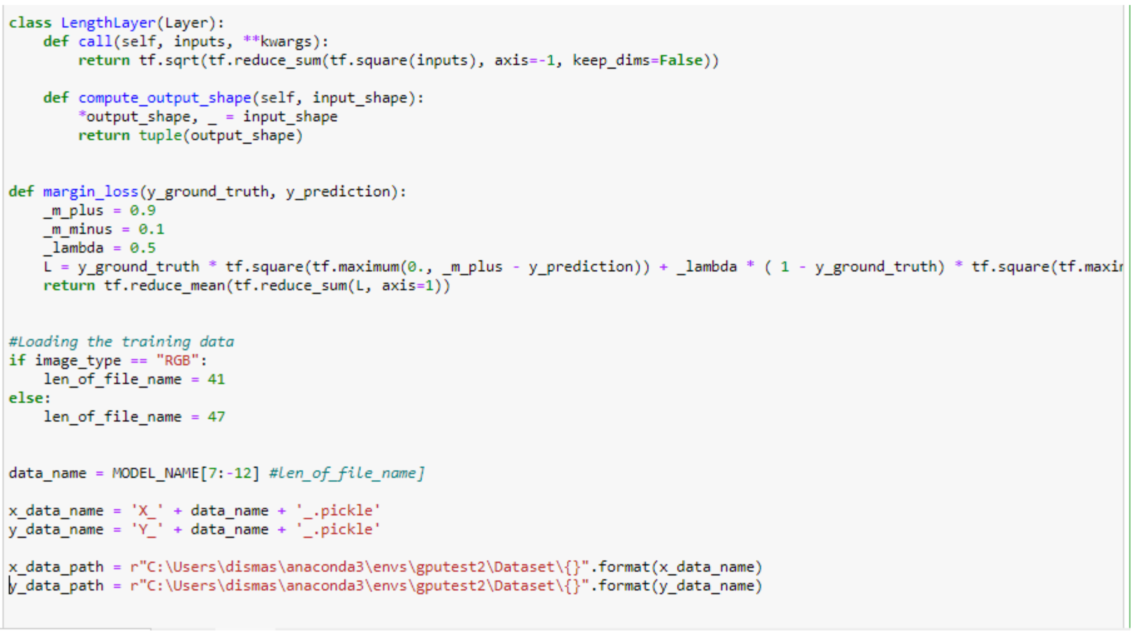
```
from keras.models import Sequential
import os
import numpy as np
from keras.callbacks import EarlyStopping, ModelCheckpoint, TensorBoard
import time
import pickle
from distutils.version import LooseVersion
import warnings

# Check for a GPU
if not tf.test.gpu_device_name():
    warnings.warn('No GPU found. Please ensure you have installed TensorFlow correctly')
else:
    print('Default GPU Device: {}'.format(tf.test.gpu_device_name()))

#configurations
n_img_per_class = 5
IMG_SIZE = 28
nr_of_epochs = 200
my_batch_size = 256
val_split = 0.35


NUM_CHANNEL=3  # values are 1 or 3
image_type = "RGB" #"""grayscale" # values are "grayscale" or "RGB"



MODEL_NAME = 'THESIS_Datasets_' +  str(n_img_per_class) + '_images_per_class_' + str(image_type) + '_' + str(IMG_SIZE) + 'x' + s

#CallBacks for saving tensorboard file and for checkpoint file
#saved_model_path = r"C:\Users\diezec\Anaconda3\envs\env1\saved_model\{}".format(MODEL_NAME)
saved_model_path = r"C:\Users\dismas\anaconda3\envs\gputest2\saved_model\{}".format(MODEL_NAME)
checkpoint_path = saved_model_path +'__CHECKPOINT__.ckpt'
```



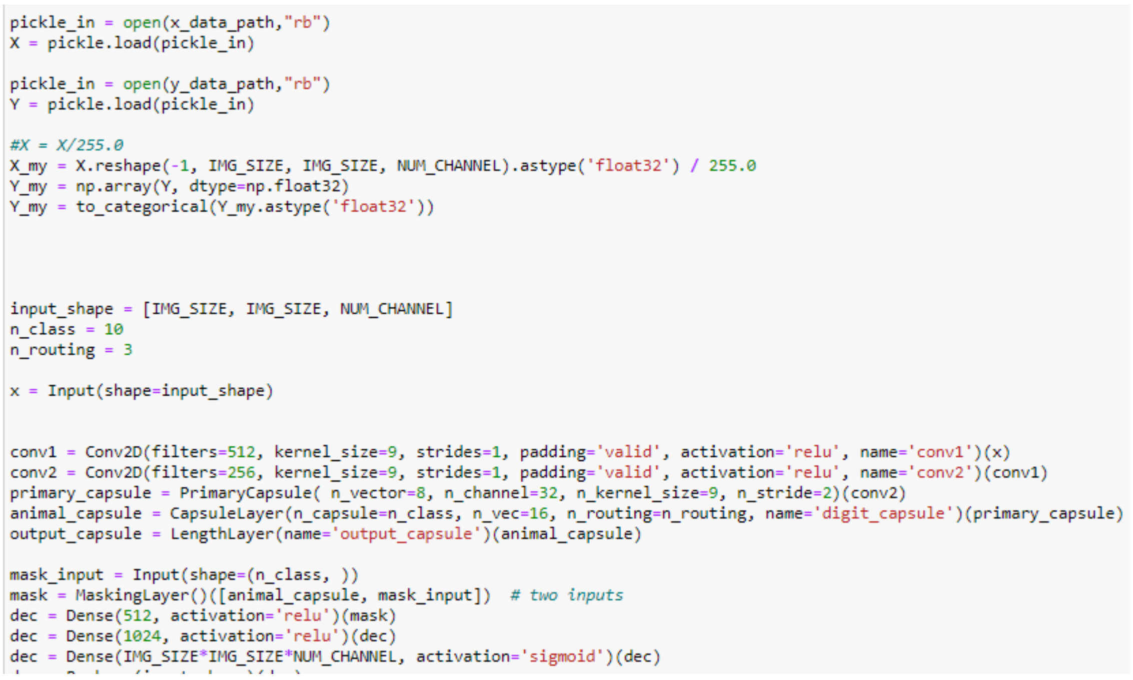
```
checkpoint_dir = os.path.dirname(checkpoint_path)

# Create a callback that saves the model's weights every 10 epochs
checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
                                                         filepath=checkpoint_path,
                                                         verbose=1,
                                                         save_weights_only=True
                                                         #save_freq=10
                                                         )

tensorboard = TensorBoard(log_dir=r"C:\Users\dismas\anaconda3\envs\gputest2\logs\{}".format(MODEL_NAME))

|
def squash(output_vector, axis=-1):
    #output_vector=np.array(output_vector)
    norm = tf.reduce_sum(tf.square(output_vector), axis, keep_dims=True)
    #norm = tf.reduce_sum(tf.square(output_vector), axis, keepdims=True)
    #norm = tf.reduce_sum(tf.square(output_vector), axis)
    return output_vector * norm / ((1 + norm) * tf.sqrt(norm + 1.0e-10))


class MaskingLayer(Layer):
    def call(self, inputs, **kwargs):
        input, mask = inputs
        return K.batch_dot(input, mask, 1)

    def compute_output_shape(self, input_shape):
        *_, output_shape = input_shape[0]
        return (None, output_shape)


def PrimaryCapsule(n_vector, n_channel, n_kernel_size, n_stride, padding='valid'):
    def builder(inputs):
        output = Conv2D(filters=n_vector * n_channel, kernel_size=n_kernel_size, strides=n_stride, padding=padding)(inputs)
        output = Reshape( target_shape=[-1, n_vector], name='primary_capsule_reshape')(output)
```



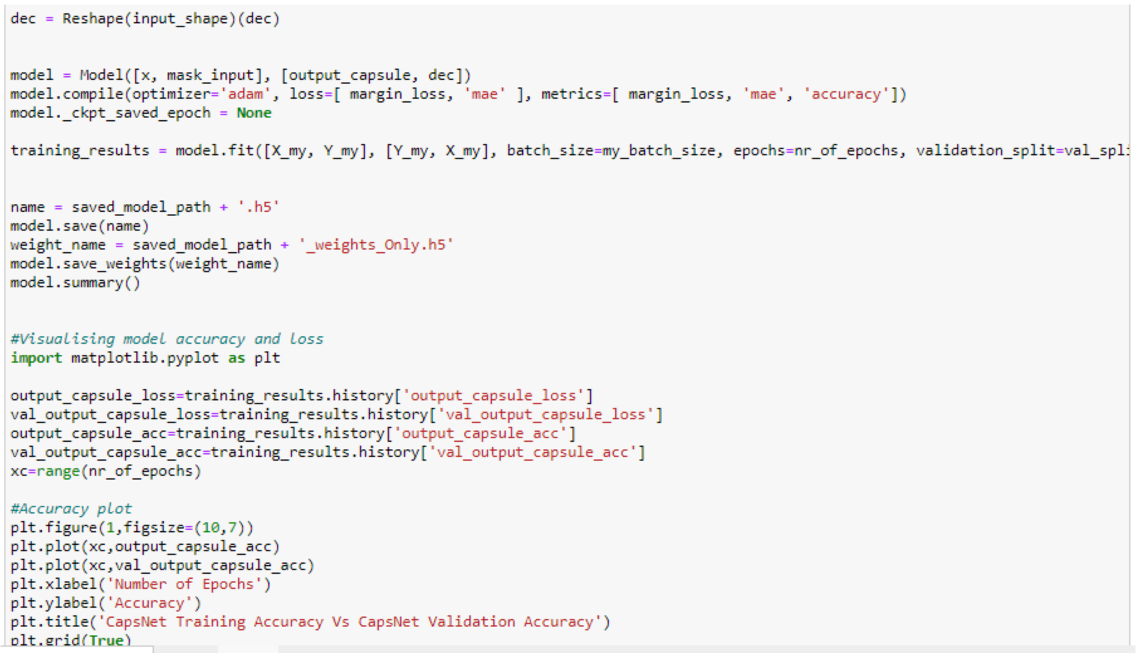
```
        |
        return Lambda(squash, name='primary_capsule_squash')(output)
    return builder



class CapsuleLayer(Layer):
    def __init__(self, n_capsule, n_vec, n_routing, **kwargs):
        super(CapsuleLayer, self).__init__(**kwargs)
        self.n_capsule = n_capsule
        self.n_vector = n_vec
        self.n_routing = n_routing
        self.kernel_initializer = initializers.get('he_normal')
        self.bias_initializer = initializers.get('zeros')

    def build(self, input_shape): # input_shape is a 4D tensor
        _, self.input_n_capsule, self.input_n_vector, *_ = input_shape
        self.W = self.add_weight(shape=[self.input_n_capsule, self.n_capsule, self.input_n_vector, self.n_vector], initializer=se
        self.bias = self.add_weight(shape=[1, self.input_n_capsule, self.n_capsule, 1, 1], initializer=self.bias_initializer, nar
        self.built = True

    def call(self, inputs, training=None):
        input_expand = tf.expand_dims(tf.expand_dims(inputs, 2), 2)
        input_tiled = tf.tile(input_expand, [1, 1, self.n_capsule, 1, 1])
        input_hat = tf.scan(lambda ac, x: K.batch_dot(x, self.W, [3, 2]), elems=input_tiled, initializer=K.zeros( [self.input_n_
        for i in range(self.n_routing): # routing
            c = tf.nn.softmax(self.bias, dim=2)
            outputs = squash(tf.reduce_sum( c * input_hat, axis=1, keep_dims=True))
            if i != self.n_routing - 1:
                self.bias += tf.reduce_sum(input_hat * outputs, axis=-1, keep_dims=True)
        return tf.reshape(outputs, [-1, self.n_capsule, self.n_vector])

    def compute_output_shape(self, input_shape):
        # output current layer capsules
        return (None, self.n_capsule, self.n_vector)
```



```
class LengthLayer(Layer):
    def call(self, inputs, **kwargs):
        return tf.sqrt(tf.reduce_sum(tf.square(inputs), axis=-1, keep_dims=False))

    def compute_output_shape(self, input_shape):
        *output_shape, _ = input_shape
        return tuple(output_shape)


def margin_loss(y_ground_truth, y_prediction):
    _m_plus = 0.9
    _m_minus = 0.1
    _lambda = 0.5
    L = y_ground_truth * tf.square(tf.maximum(0., _m_plus - y_prediction)) + _lambda * ( 1 - y_ground_truth) * tf.square(tf.maxi
    return tf.reduce_mean(tf.reduce_sum(L, axis=1))


#Loading the training data
if image_type == "RGB":
    len_of_file_name = 41
else:
    len_of_file_name = 47


data_name = MODEL_NAME[7:-12] #len_of_file_name]

x_data_name = 'X_' + data_name + '_.pickle'
y_data_name = 'Y_' + data_name + '_.pickle'

x_data_path = r"C:\Users\dismas\anaconda3\envs\gputest2\Dataset\{}".format(x_data_name)
y_data_path = r"C:\Users\dismas\anaconda3\envs\gputest2\Dataset\{}".format(y_data_name)
```

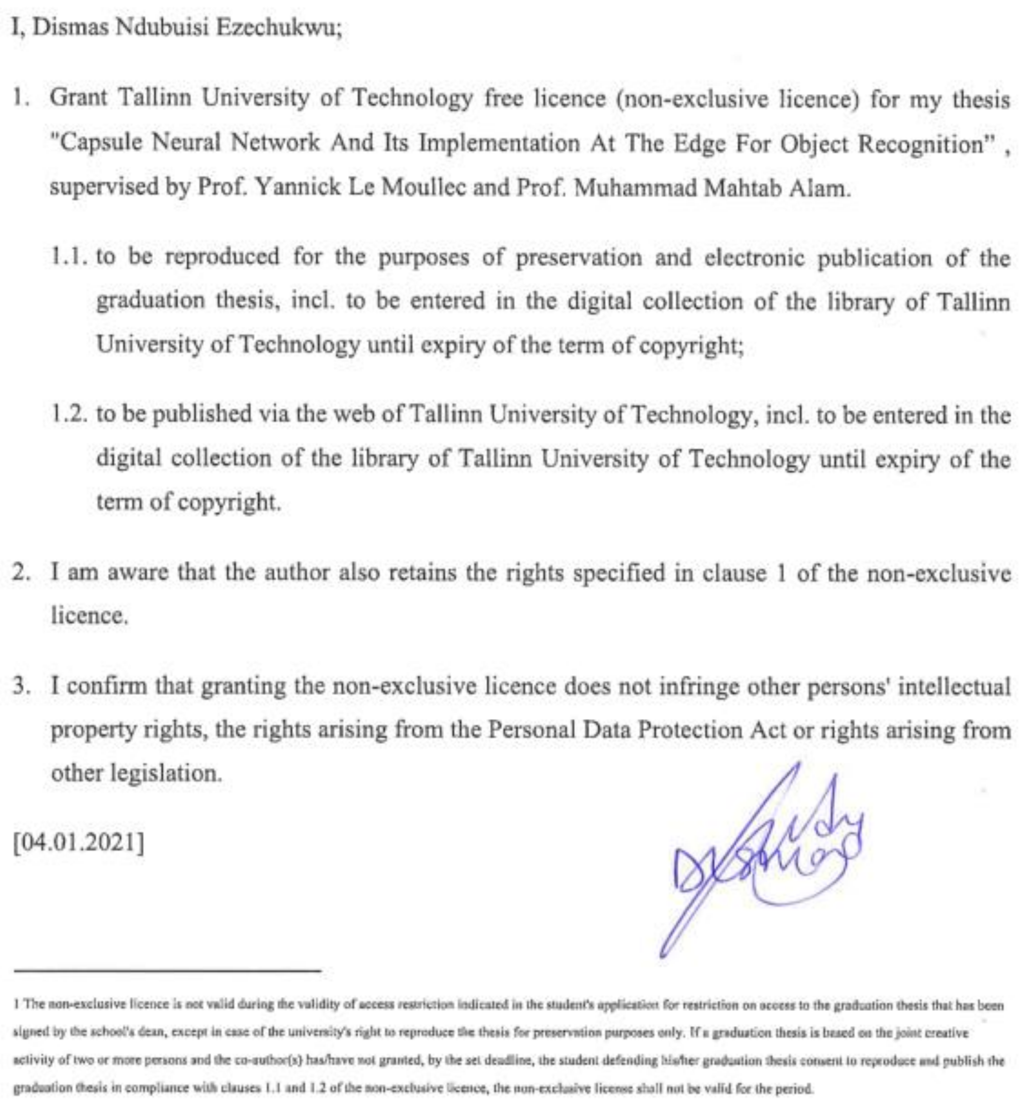


```
pickle_in = open(x_data_path,"rb")
X = pickle.load(pickle_in)

pickle_in = open(y_data_path,"rb")
Y = pickle.load(pickle_in)

#X = X/255.0
X_my = X.reshape(-1, IMG_SIZE, IMG_SIZE, NUM_CHANNEL).astype('float32') / 255.0
Y_my = np.array(Y, dtype=np.float32)
Y_my = to_categorical(Y_my.astype('float32'))



input_shape = [IMG_SIZE, IMG_SIZE, NUM_CHANNEL]
n_class = 10
n_routing = 3

x = Input(shape=input_shape)


conv1 = Conv2D(filters=512, kernel_size=9, strides=1, padding='valid', activation='relu', name='conv1')(x)
conv2 = Conv2D(filters=256, kernel_size=9, strides=1, padding='valid', activation='relu', name='conv2')(conv1)
primary_capsule = PrimaryCapsule( n_vector=8, n_channel=32, n_kernel_size=9, n_stride=2)(conv2)
animal_capsule = CapsuleLayer(n_capsule=n_class, n_vec=16, n_routing=n_routing, name='digit_capsule')(primary_capsule)
output_capsule = LengthLayer(name='output_capsule')(animal_capsule)

mask_input = Input(shape=(n_class, ))
mask = MaskingLayer()([animal_capsule, mask_input])  # two inputs
dec = Dense(512, activation='relu')(mask)
dec = Dense(1024, activation='relu')(dec)
dec = Dense(IMG_SIZE*IMG_SIZE*NUM_CHANNEL, activation='sigmoid')(dec)
```



```
dec = Reshape(input_shape)(dec)


model = Model([x, mask_input], [output_capsule, dec])
model.compile(optimizer='adam', loss=[ margin_loss, 'mae' ], metrics=[ margin_loss, 'mae', 'accuracy'])
model._ckpt_saved_epoch = None

training_results = model.fit([X_my, Y_my], [Y_my, X_my], batch_size=my_batch_size, epochs=nr_of_epochs, validation_split=val_spl


name = saved_model_path + '.h5'
model.save(name)
weight_name = saved_model_path + '_weights_Only.h5'
model.save_weights(weight_name)
model.summary()


#Visualising model accuracy and loss
import matplotlib.pyplot as plt

output_capsule_loss=training_results.history['output_capsule_loss']
val_output_capsule_loss=training_results.history['val_output_capsule_loss']
output_capsule_acc=training_results.history['output_capsule_acc']
val_output_capsule_acc=training_results.history['val_output_capsule_acc']
xc=range(nr_of_epochs)

#Accuracy plot
plt.figure(1,figsize=(10,7))
plt.plot(xc,output_capsule_acc)
plt.plot(xc,val_output_capsule_acc)
plt.xlabel('Number of Epochs')
plt.ylabel('Accuracy')
plt.title('CapsNet Training Accuracy Vs CapsNet Validation Accuracy')
plt.grid(True)
```

Appendix



I, Dismas Ndubuisi Ezechukwu;

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis "Capsule Neural Network And Its Implementation At The Edge For Object Recognition" , supervised by Prof. Yannick Le Moullec and Prof. Muhammad Mahtab Alam.

   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;

   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.

2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.

3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

[04.01.2021]

[illegible]

1 The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.