VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES

Nguyen Hoang Anh Minh

COMPUTER VISION DEPLOYMENT ON EDGE DEVICES

YOLOv3 on Movidius Raspberry Pi

Technology and Communication
2019

VAASAN AMMATTIKORKEAKOULU
UNIVERSITY OF APPLIED SCIENCES
Information Technology

ABSTRACT

Author	Nguyen Hoang **Anh Minh**
Title	Computer Vision Deployment on Edge Devices
Year	2019
Language	English
Pages	87
Name of Supervisor	Jukka Matila

Artificial intelligence has been receiving tremendous progress in the past recent years thanks to the advancement of technology. Nowadays it is the most active field of RnD with huge resources and efforts to further it fast forwards. Uncountable varieties of applications are leveraging its capabilities to solve problems that were impossible to tackle before. However, most of the artificial intelligence application systems developed at the moment require desktop or server class of computing power to run on and hence, being able to scale these systems to more resource restricted devices is highly desirable and would pave the path to many intelligent solutions. As the technology community is very sensitive and is sharing the same vision, major organisations are quickly coming in with supports of their products to provide solutions for the development of relating ideas. Among many of those solutions, one is the popular hardware series offered by Intel, the Movidius Neural Compute Stick for computation acceleration of deep learning at inference.

Since new technology are always based on other advancements, the Movidius is not an exception. It supports most of the famous open source machine learning frameworks and among which, the TensorFlow libraries, developed by Google is increasing adopted by the machine learning community and is also the developers' and scientists' favourite. You Only Look Once (YOLO) is a state-of-the-art object detecting system capable of real-time processing at extremely high accuracy. It is created in the DarkNet framework with limited scalability across different platforms but rather for just researching.

This work is attempting to enable a Raspberry Pi, aided by the Movidius Neural Compute Stick hardware to run the latest YOLO application. In the results, the TensorFlow of YOLOv3 were created and reetained the original accuracy while maintaining real-time performance on a laptop setup. On the Raspberry Pi, it was able to preserve the original accuracy with trade-off to some real-time performance.

CONTENTS

LIST OF FIGURES AND TABLES

LIST OF SNIPPETS

TABLE OF ABBREVIATIONS

DL	Deep learning
HI	Human intelligence
IQ	Intelligence Quotient
EQ	Emotion Quotient
AI	Artificial intelligence
ML	Machine Learning
CNN	Convolutional neural network
RNN	Recurrent neural network
BN	Batch Normalisation
NCS	Neural compute stick
VPU	Vision processing unit
G**PU**	Graphics processing unit
SDK	Software development kit
CUDA	Compute unified device architecture
SOM	System-on-module
SDK	Software development kit
TPU	Tensor processing unit
USB	Universal serial bus
RPI	Raspberry pi

USB	Universal Serial Bus
PC	Personal Computer
YOLOv3	You only look once version 3
TF	TensorFlow
IR	Intermediate representation
OpenVINO	Open Visual Inference and Neural Network Optimisation
ReLU	Rectified Linear Unit
API	Application Programming Interface
BERT	Bidirectional Encoder Representations from Transformers
RGB	Red – Green – Blue
NLP	Natural Language Processing
NCE	Neural Compute Engine
TOPS	Trillion Operation per Second
NCSDK	Neural Compute Software Kit
CV	Computer Vision
FPGA	Field Programmable Gate Array
IPU	Image Processing Unit
IR	Intermediate Representation
XML	Extended Mark-up Language
SoC	System on Chip

RAM	Random Access Memory
ARM	Advanced RISC Machine
CPU	Central Processing Unit
HDMI	High Definition Multimedia Input
LAN	Local Area Network
GPIO	General Purpose Input Output
GFLOP	Giga Floating Point operations per second
DMA	Direct Memory Access
POP	Package on Package
OS	Operating System
HD	High Definition

1 INTRODUCTION

1.1 Objectives

Deep learning (DL) powered systems at this point in time are mostly using the conventional cloud-based setup with servers powered by capable parallel computing hardware to process client inference requests from user applications. This enforces data collection and data constraint within centralised data centres being under control by the service providers. But as data is becoming the new oil, this has brought up many serious concerns in the society about privacy, security, data exploitation and misuse. At the same time, recent development of computer vision has taken advantages of DL integration and were able to achieve significant results. Complementing to this, object detections systems are increasingly gaining huge attentions for accomplishing real-time performance. Despite that, these applications still have to trade off with very intensive processing power. Thankfully, several solutions have become available in terms of both hardware and software to speed up DL based applications onto low computing power devices. With this being said, the ultimate goal of this work is to build a decentralised embedded system for effective object detection that can run in real-time and powered by the state-of-the-art DL techniques. The result of this work will be significantly advantageous to future advancement of human vision augmentation, humanoid technology as well as other similarly related fields where instant perception to the real world is crucial.

1.2 Human intelligence vs. artificial intelligence

Human intelligence (HI) is a combination of extraordinarily complex cognitive processes that are both objective and subjective. Most of the objective processes include, oral and verbal communication and comprehension, reasoning, concept formation, decision making, and problem solving. Yet any of these also incorporates subjective factors such as emotions and motivation. Other subjective processes are kinaesthetic awareness, perception, and prejudice. Furthermore, education plays a crucial role to improve this type of intelligence and the higher quality the education, the better the intelligence. However, human education is still bound to dictatorially imposing the truths without providing convincing information (data) to allow for self – internalising the essence of the concepts. As a result, learners only memorise the knowledge without a thorough understanding which

leads to failing to generalise it into solving novel problems as they are likely to require very skilful manipulation of ideas and concepts. In addition, the level of HI is evaluated through standardised IQ and IE tests. On the other hand, artificial intelligence (AI) is a mathematical construct designed and created by human intelligence to do a specific task at superhuman level of performance. It is a conglomeration of many fields of sciences including philosophy, psychology, neuroscience, computer science, statistics, linguistic and more. this kind of intelligence is typically approached in two ways. The first one has been traditionally utilised for years in the pass where a rule-based system of algorithms is hard coded, although this can be still considered the case in today's smart applications. The other modern approach is self – learning, i.e. machine learning, where algorithms are designed to look at data and learn the underlying patterns, relationships, and transformations all by itself without being explicitly told about anything beforehand at all. In order to improve AI, the prior way requires a lot of design and engineering efforts to implement detailed rules. While in the latter, these efforts are spent on sophisticated system structures along with advanced optimising algorithms to enable for capturing complex information hidden in the nature of data. The intelligence level of a system can be evaluated using crafted performance metrics that define several components related to the fields for which the system is built. [1]

1.3 Machine learning and deep learning

The latter approach has become extremely effective in recent years and ubiquitously outperforms all traditionally prior approaches. The elementary factor behinds this success come down to the concept of machine learning (ML). while this is actually been around for a long time, it was not until today could it only become effective. This is thanks to the evolution of new computing technologies that are now capable of processing faster and also significantly bigger amount of data than ever before. Therefore, ML is being increasingly researched and large numbers of state-of-the-art research papers are actively being published which are immediately being adopted to every aspect of life, some of them include finance, healthcare, computer vision, speech recognition robotics, and recently adversarial. There are two popular definitions to ML given by two famous people in the field. The first one was offered by Arthur Samuel, an Artificial Intelligence research pioneer, he described machine learning as "the field of study that gives computers the ability

to learn without being explicitly programmed.". Later on, Tom Mitchell, a computer scientist, provided a more modern definition which is: "a computer programme is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance in tasks T, as measured by P, improves with experience E.". In general, any machine learning problem can be divided into two classifications: Supervised Learning and Unsupervised Learning. Giving explicitly the input data and its output, the idea of supervised learning is to find a relationship in the form of a mathematical equation that maps between them. There are two categories in supervised learning which are regression and classification. A problem is categorised as regression if the predicted outputs are continuous. Instead, the problem is categorised as classification if the predicted outputs are discrete values (categories). In terms of unsupervised learning, the predicting goal is to derive the structure of the data by clustering them based on the relationships among the variables rather than considering the effect of each variable. Therefore, there is no feedback on the prediction results. Most of the technology applications are significantly benefitted from supervised learning, specifically the methods of deep learning. They generally comprise a diligently designed neural inspired network architecture being optimised by an advanced algorithm that allow the network to learn (find the best set of rules) and predict on structured (i.e. labelled) data. The network improves by being iteratively trained on an enormous amount of data with a large extent of diversity in the targeting field to become expert in it. Therefore, not only does deep learning systems thirst for data, they also requires tremendous amount of training time and rely heavily on large computational resources. For that reason, they tend to be developed on very capable machine with powerful hardware and deployed to a cloud-based system for use. However, this setup introduced a lot of drawbacks such as a required network connection and low latency, it started to become more and more desirable to distribute deep learning systems to individual devices.

1.4 Computer vision with deep learning

DL has shot up a tremendous progress in various areas in the field of computer vision, many of them include image classification, object detection, semantic segmentation, image generation, and pose estimation. Generally, tasks in any of these areas utilise either the convolutional neural network (CNN) or the recurrent neural network (RNN) in deep

learning and train on a large dataset of images at the pixel level, hence the larger the image size, the better the accuracy but the more expensive the computational cost becomes. Although they have achieved impressive results, most of the systems run very slowly and have to require the empowerment of parallel computing capable hardware such as the GPU to process. This has become a great driving force for the technology industry to push forwards the hardware capability and further optimise the software algorithms. As a result, many of the state-of-the-art computer vision systems were designed to effectively leverage the computing hardware and able to achieve real-time performance. At the same time, many hardware manufacturers have acknowledged the trend of needs and have been making great improvements in terms of the processing power to support the advancement of deep learning. For Intel, the company introduced the Neural Compute Stick (NCS) series, a highly efficient deep learning compute accelerator features its own Vision Processing Unit (VPU), to enable developments of applications for edge devices along with its powerful framework to heterogeneously share computations across other intel's hardware [2]. Nvidia, besides releasing ever more powerful (GPUs) packing more CUDA cores that increase parallel computation capability, also provides the Jetson System-on-module (SOM) computers that package those GPU performance to power all kind computer vision and graphics application for embedded systems. The Jetson devices are programmed by the Jetpack SDK [3]. Similarly, Google offers paid services for use of its cloud Tensor Processing Unit (TPU) infrastructure and Google Coral, featuring dedicated Edge TPU cores. The Coral contains the development board similar to the Raspberry Pi (RPI) micro-computer but with a removable SOM, and an USB accelerator similar to the NCS, and with other extension components. The Coral runs on its official operating system Google's Mendel Linux but can also run on any Debian based Linux distributions (distros) and is programmed in Google's TensorFlow framework [4].

1.5 Computer vision at the edge.

Conventionally, ML systems are deployed in centralised cloud computing paradigm where services at the user-end send requests of inference over to the server and wait to receive the result. This helps taking advantages of the computational resources of the server-end and allows for scalability of software that utilise the system to a wider range of devices, from PCs to mobile devices and especially embedded. However, this setup is

dependent on a stable internet connection and communication protocols badly penalise the latency. This is the case for applications operating on a constant stream of video input, which would over-load the network medium during data transfer processes and slow down runtime performance. To overcome this challenge, great efforts need to be put in to extending the power of edge computing and decentralising ML processes to the edge of the network. This will significantly benefit delay-sensitive and critical mission applications such as autonomous driving. In self-driving cars, immediate decision making is extremely crucial as they need to be fully aware of the surrounding and the state of all the other cars nearby to execute the right decisions, if the traffic light turns red or pedestrians are crossing in the front, the reaction to stop and notification to the vehicle behind must be executed immediately. Having gained a lot of attentions in ML decentralisation at the edge, there are currently several edge solutions available from leading technology organisations, which when being merged with state-of-the-art real-time computer vision models would become a very powerful module that paves the way to many innovations of technology, in which robotics would have significant impact in developing artificial visual perception.

1.6 Real-time object detection on embedded system

With the current trend of AI application, the development is shifting to the edge and DL powered applications for computer vision are getting more efficient and are breaking each other's achievement in real-time performance, this thesis introduces an attempt to deploy the state-of-the-art real-time object detection system You Only Look Once version 3 (YOLOv3) onto the RPI 3 B+ with computing boosts from multiple Intel's Movidius NCS2 accelerators. The implementation includes first re-building YOLOv3 model in TensorFlow (TF), then freezing it to a saved model and converting to the intermediate representation (IR) model graph which was compatible with the OpenVINO toolkit. Developed object detection programmes used this toolkit to utilise the NCS to process predictions on input images captured by a camera module. In case of the RPI, a Pi camera module was used.

2 BACKGROUND KNOWLEDGE

2.1 Artificial Neural Networks

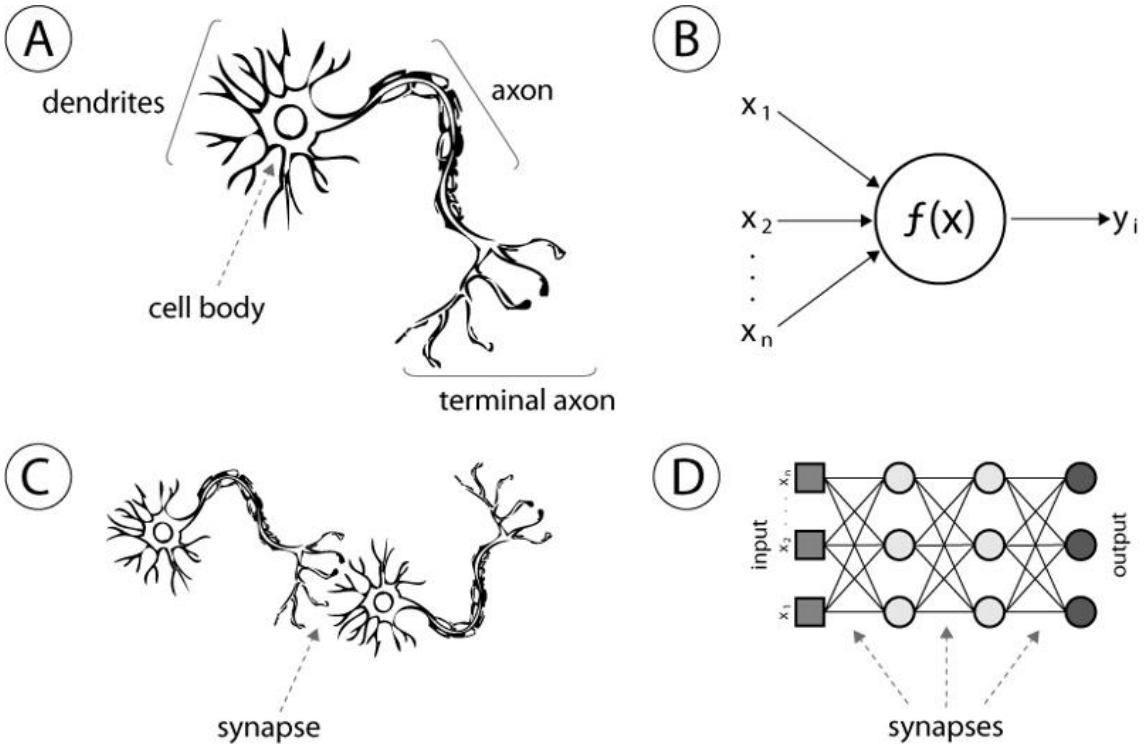

Figure 1. Biological neuron and artificial neuron comparison [5]

Artificial Neural networks (ANN), also often referred to as neural networks, are information processing paradigms inspired by the human brain system with the motivation of replicating its remarkable capabilities of human cognition to solve computer tasks. The very first neural network was called *perceptron* and was invented by psychologist Frank Rosenblatt in 1958 [6]. Neural networks then were able to address many problems that were too complex or impossible to solve by standard computational and statistical methods, it started to gain popularity among researchers. By the late 1980s, many institutions adopted this algorithm into a variety of applications [7]. The structure of neural networks stimulates the network of neurons in the brain. Biologically, a neuron contains a cell body and is connected by dendrites (i.e. input wires) to receive electric signals (i.e. spikes) from other neurons. This neuron has an output wire called axon to connect to other neurons and send signals away. The axon has many terminals and those other neurons are connected to these axon terminals via synapses.

2.1.1 Architecture Overview

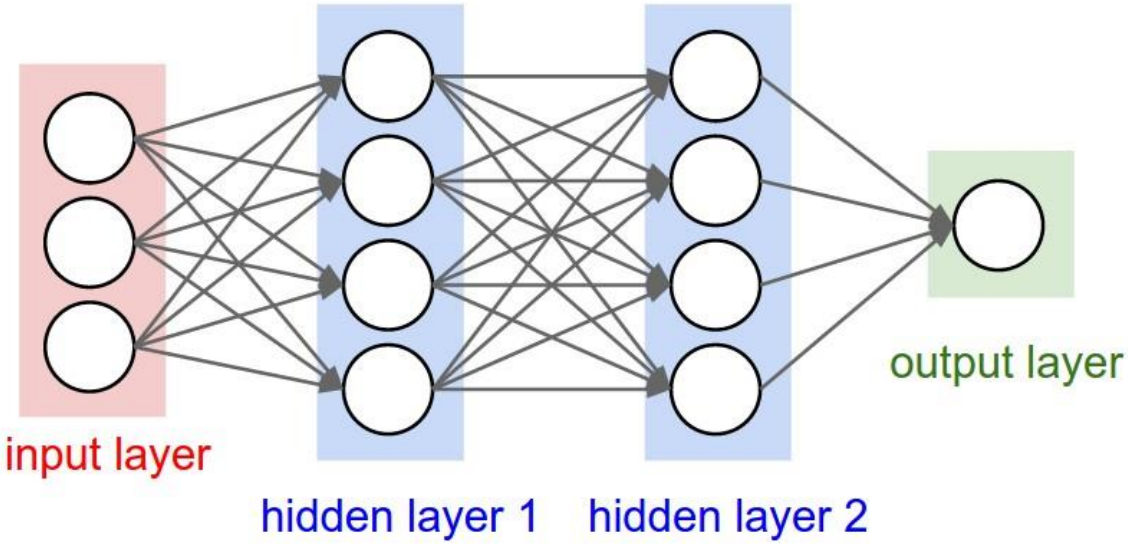

Figure 2. A 3-layer neural network with 2 hidden layers with 4 neurons each and 1 output layer with 1 neuron [8]

In neural networks, each neuron corresponds to a computational node which receives information from a certain number of input nodes (features) multiplied by a set of weights (i.e. weights simulate the role of synapses that allows for signal transfer between neurons) and perform some non-linearity computations, then sends outputs away to other nodes. Neural networks are usually composed of a set of neurons (i.e. nodes) for many layers. The more nodes the layers have, the wider the layers are. Similarly, the more layers there are, the deeper the networks will be. The last layer is usually the output layer and depends on the problems the neural networks are trying to solve, this output layer can either has one output node, if the problem is classification, or more than one output nodes if the problem is regression.

2.1.2 Algorithm Overview

The mathematical representation of one or more nodes is expressed in the following set of functions.

Hypothesis function: $$z = h_{w,b}(x) = w^T x + b \quad (0)$$

Where:

w – the 2-dimensional weight matrix belongs to the node

b – the bias constant belongs to the node

x – the 2-dimensional input matrix coming to the node

z – the 2-dimensional output matrix of the hypothesis function

If the current layer that is under consideration is layer i, then what the hypothesis function (0) computes simulates the flow of information coming from nodes in layer $i-1$ to the nodes in layer i, similar to when electrical signals travel from a set of neurons to another set through synapses. The existence of the bias b is to allow the nodes to learn some preference over certain kind of information, yet if the neural networks would implement a batch normalisation (BN) algorithm [9] to normalises the output z then b would become unnecessary as this constant would be cancelled out.

Activation function: $$a = g(z) \quad (1)$$

Where:

g – the activation function of choice (e.g. ReLU)

a – the 2-dimensional output matrix of the activation function

the output z of function (0) would then be applied by an activation function to perform a non-linearity transformation which results in probabilistic values. Many common activation functions include Sigmoid [10], Tanh [11], ReLU and leaky-ReLU [12], and Softmax [13].

Loss function:

$$J(w,b) = \frac{1}{m}\sum_{i=0}^{m}\mathcal{L}(\hat{y}_i, y_i) \quad (2)$$

Where:

m – the number of training examples.

$\mathcal{L}$ - the loss function applied, dependent on the of solving problem (i.e. regression or classification).

$\hat{y}_i$ – the prediction of the neural network corresponding to example i.

y_i – the ground truth value corresponding to example i.

The loss function is used to training neural networks. When performing inference on the neural network model on a set of inputs, every layer performs computations on the output of the previous layer except the first layer which does it on the input, this takes place from left to right. This activity is called forwards propagation. The prediction of the neural network (denoted as $\hat{y}$) is the output of the activation function of all the nodes in the last layer. When training neural networks, forward propagation occurs for many times over *m* training examples, this number of times is called *epoch.*

As neural networks computation take place over a set of identical-shaped matrices and constants, it is very often that the programming implementation will be done in the vectorisation manner to take advantage of the hardware architecture more efficiently and accelerate computing speed. Therefore, if function (0) is vectorised with respect to the number of nodes in layer i, w will be become a 3 – dimensional weight matrix and b will become a bias vector. In addition, most of the machine learning libraries like TF have already integrated vectorisation very effectively to the core of their APIs and dedicated processing units has also been created to push machine learning computation even faster.

2.2 Convolutional Neural Networks.

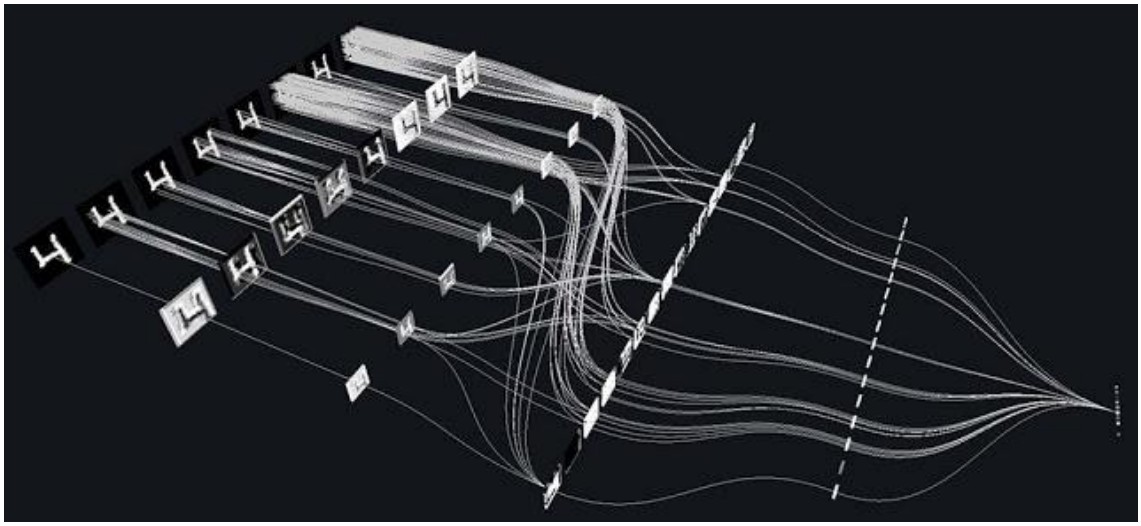

Figure 3. Convolutional Neural Network architecture

CNN often referred to as ConvNet, is a special artificial neural network that was created specifically to process high dimensional data, such as a digital image. A CNN contains a number of convolutional layers to filter the high dimensional input for useful information. At every convolution layer, the operation involves convolving the input data (feature maps) with many convolution kernels (filters) to output new transformed feature maps. The filters in the convolutional layers are modified and learned from the training process to extract the most useful feature information for a specific task. CNN is capable of filtering out information about both the shape and the texture of an object when confronted with a general object recognition task, in which different classes of objects tend to have different shapes, but sometimes some different objects hold the same shape and they are characterised by their appearances and CNN is powerful enough to generalise these well. CNN has proved their effectiveness in a variety of applications in both computer vision and language problems. For image processing systems, they include image classification, face recognition, object detection, object segmentation, art and style transfer. For speech processing systems, they include speech recognition, natural language processing, text classification, text analysis, image captioning and language translation. Besides, many of these applications are integrated in state-of-the-art AI systems such as robots, virtual assistants, and self-driving cars. [14]

2.2.1 Architecture Overview

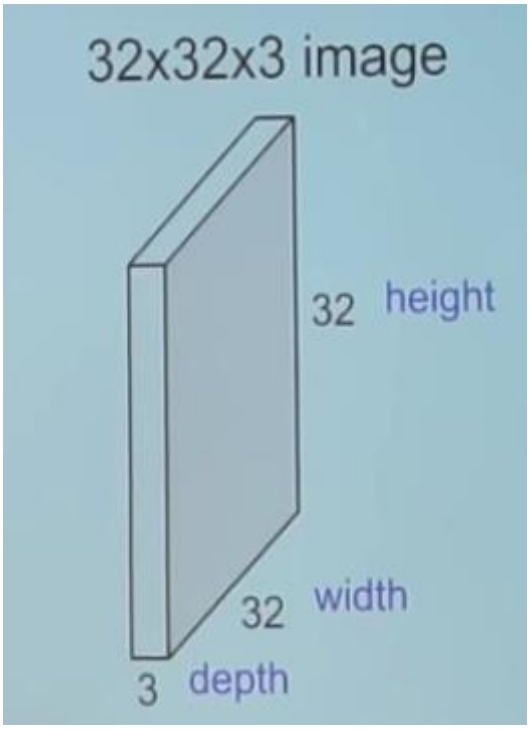

Figure 4. The shape of images in CIFAR-10 dataset [15]

In general, CNN resembles ANN in a way that it is still based on the idea of layers of artificial neurons carrying learnable weights and biases. Each neuron performs a dot product with some inputs and is optionally followed by a non – linearity transformation. The whole network is still expressed by a differentiable score function – loss function. In traditional ANN, the fully connected neuron structure does not scale well to 3-dimensional input data such as full images. For example, in the CIFAR-10 dataset where images are of size 32x32x3 (32 in width, 32 in height, and 3 colour channels), there would be 32*32*3 = 3072 2-dimensional weight matrices. Although this amount still seems reasonable, an image of more respectable size, e.g. 416x416x3 for the input shape of images in the YOLOv3 model [16], this number of weights would significantly escalate to 416*416*3 = 519,168. Moreover, as there would need to be several other hidden layers with a huge number of nodes more than this, the total parameters would add up even more! Clearly, this architecture design is so computationally wasteful, and the enormous number of parameters would quickly lead the model to overfitting. In contrast, as the architecture of CNN is designed in a sensible way that encodes certain specific properties

that make multi-dimensional data processing more efficient. This highly enhances the computational efficiency and vastly reduce the number of parameters in the network. There are typically three main layers in a CNN architecture: the convolutional layer, the pooling layer and the traditional neural network layer (or fully connected layer). Specifically, convolutional layers are made up of 3 – dimensional volume nodes called kernels (filters), each kernel has its dimensions called width, height, and depth.

2.2.2 Algorithm Overview

- **Convolutional layer**

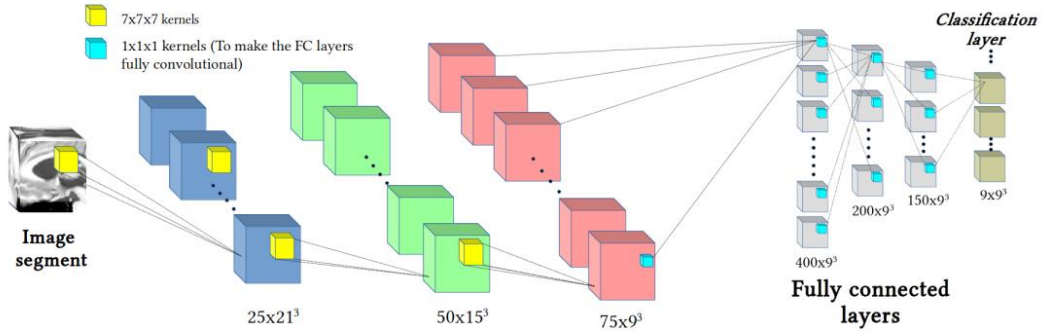

Figure 5. A convolutional neural network architecture [17]

The convolutional layer is the core building block of a CNN as it does most of the computational computation. A convolutional layer comprises of a set of filters in which each one is small in width and height but extends in full depth dimension according to that of the input volume. For example, the first convolutional layer of YOLOv3 has 32 filters of shape 3x3x3 where the last dimension matches the number of colour channels in the input image, i.e. RGB. This is inspired from the idea of the receptive field in human vision. In addition, CNN uses the parameter sharing strategy that drastically helps reduce the number of parameters by adopting the assumption that if one filter is useful to compute at one spatial location, then it will also be useful at some other locations as well. Therefore, in case of YOLOv3, the first layer contains a total of 32x3x3x3=864 unique weights.

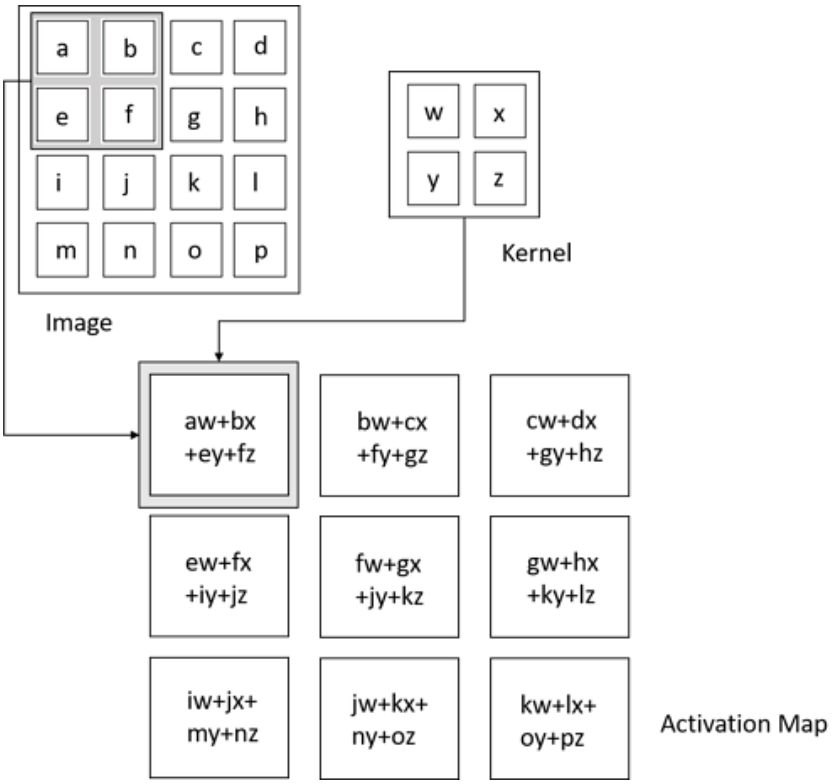

Figure 6. the 2D demonstration of a convolutional operation [18]

During the forwards propagation, all these filters convolve (i.e. slide) over the input volume from across the width dimension to along the height dimension and compute the dot product with the entries of the input volume at each particular location. The final output of this convolution operation is a 3-dimensional volume consisting of a collection of 2-dimensional feature slices stacked along the depth dimension, where each of which contains the responses of a filter in each spatial location of the input volume. Similar to ANN, all the value of the output volume will be optionally normalised and then transformed by a non-linearity function, which is typically a ReLU function. During the backwards propagation, each of the nodes in the 3 – dimensional output volume will compute the gradient for the weights in its filters. The gradients are added up across each feature slice and are used to update the weights in the filter which generated that feature slice. Trained CNN has its convolutional layers as a set of learned filters that can extract certain kinds of information such as edges in various orientations or blobs of colours in the shallow layers, and eventually the general shape and texture of the training objects in the deep layers. It is worth mentioning that a convolutional layer is specified by the following hyperparameters, meaning they need to be carefully selected: *the filter size, the number of filters, the*

stride and *the type of zero-padding*. The *stride*, as its name suggests, specifies the stride the filters make each time they convolve. A stride of 1 means that the filters will slide around by 1 pixel, and the larger the stride the smaller the output size will be. Zero-padding is a technique of padding the input volume with a border of additional zero values. This is effective since it allows the filters to convolve more frequently at the edges of the input and capture information better at these areas. In addition, it is also convenient as it gives you control over the output size. The two typical types of zero-padding are valid, meaning no padding at all, and same, meaning pad so that the output size is similar to that of the input.

As the arrangement of hyper-parameters are mutually constrained, a dimension of the output size, except for the depth dimension, can be formulated as follow:

Output dimension: $$\frac{n + 2p - f}{s} + 1 \quad (3)$$

Where:

n – a dimension of the input.

p – the number of zero-padding to use.

f – the corresponding dimension of the filter.

s – the stride of the filter.

- **Pooling layer (subsampling layer)**

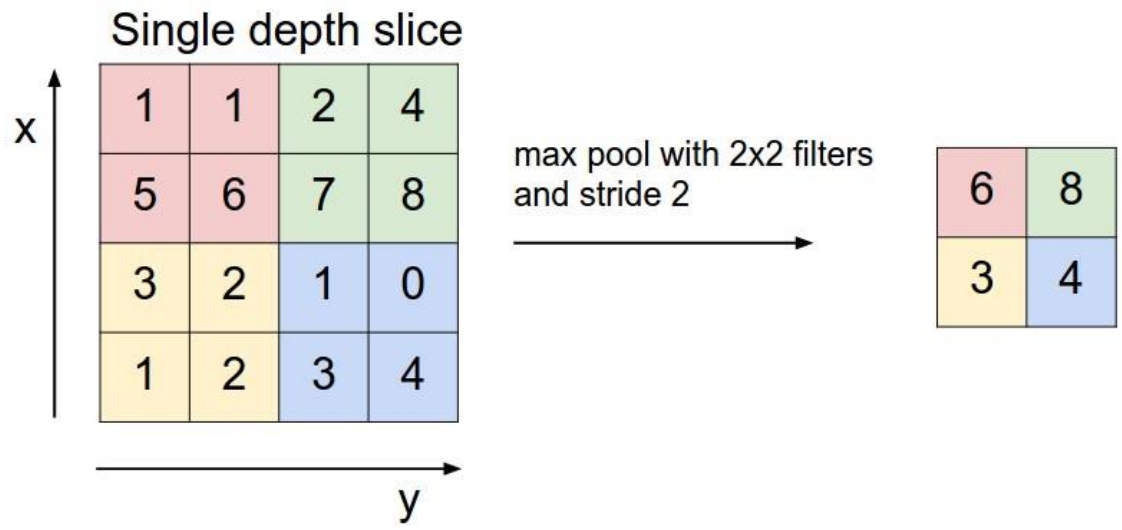

Figure 7. the demonstration of the max pooling operation. [8]

The function of the pooling layer is to progressively shrink the spatial size of the input using its convolutional pooling filter to reduce divisions in the input to a single value (down-sampling). This allows for the reduction of the number of parameters in the network, which ease out the amount of memory consumption and computation and additionally be able to control overfitting. The pooling procedure also provides simple immutability to tilt, translations and shearing and hence improves the image object detection capability of CNN. For example, the face in an image patch that is slightly translated from the centre when going through a pooling layer will be detected by its pooling filter and be funnelled into the right place. The larger the filter size, the more information is concentrated, which leads to slimmer networks that more easily fit into the GPU memory or mobile processor. However, there is a catch as setting this filter size to be too large, too much information is suppressed and leads to poor predictive performance. A typical practice in building a convolutional neural network architecture is to periodically insert a pooling layer in-between some convolutional layers. The pooling layer independently subsamples each feature slice of the input volume and downsizes it spatially using either the *Max* operation of the *average* operation. Since the pooling layer carries no weights at all, during the backwards propagation, it only routes the gradient and distributed it to the input that contributed to the operation in the forwards propagation.

- **Inception Layer**

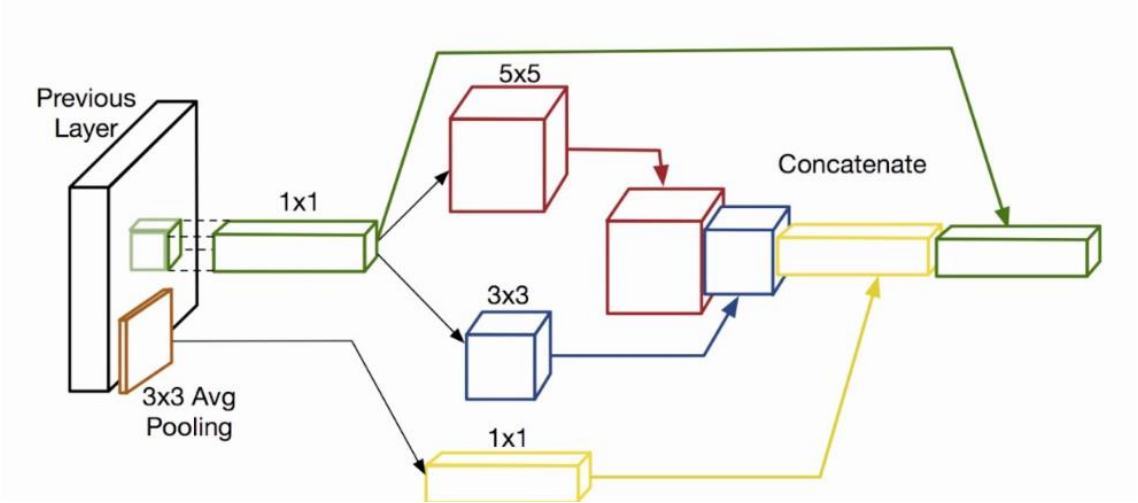

Figure 8. An inception layer, containing 1x1, 5x5 and 3x3 convolutions, and an 3x3 average pooling. The outputs of all these layers are stacked together for the final output. [19]

Inception layers in the CNN allow for using multiple convolutions with multiple filters and pooling layers simultaneously in parallel within the same layer. For example, Figure 8. demonstrates the architecture of an inception layer with usage of 3 different convolutional operations and an average pooling operation. The first convolution operation uses 1x1 filters then feed the output to 2 convolutional operations with 5x5 and 3x3 filters, and an average pooling operation. the output of all these layers are concatenated together along the depth dimension and are fed to the next layer. The intention of this implementation is to let the network learns the best weights for these filters at training and automatically extracts more useful information. This also enables deeper and larger conv layers which leads to better performance while maintaining reasonable computational cost.

- **Residual network**

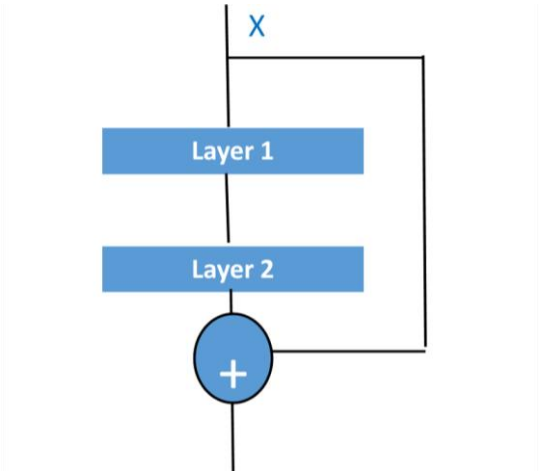

Figure 9. A demonstration of a residual connection [20]

A lot of work done by AI researchers have shown convincing theoretical analysis of the effectiveness of deep architecture of neural networks. By simply by stacking up more layers, they can effectively increase its accuracy [21]. This brought up the depth problem of vanishing gradient where the deeper the layers go, the gradients become smaller and eventually fall to zero. This adversely diminish the accuracy over layers and lead to worse performance. Kaiming et al. [22] was first to introduce this problem and proposed the ultimate solution of residual network which allowed for training of over 2000 layers. The residual network basically contains residual connections which is meant to route the output of some previous layers to the output of a later layer and perform the addition to them. This principle has been expanded into many other domains of deep learning such as speech processing systems.

2.3 Batch normalisation

BN is one of the main reasons that has made DL successful and it has become a standard feature of many machine learning libraries. The intuition behinds it concerns the problem of covariance shift. It refers to the change in the distribution of the input data used for the training algorithm which in turns affects the behaviour of machine learning algorithm. It is obvious when the train and test datasets come from different sources and they have different distributions. What BN does is to try to limit the covariance shift by normalising the activations of each layer (the output of the previous layer that is input to the next

layer) to mean 0 and unit variance. This allows each layer to learn on a more stable distribution of inputs and to use larger learning rate and thus accelerate the training of the network. [23]

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_{\mathcal{B}} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_{\mathcal{B}}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_{\mathcal{B}})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_{\mathcal{B}}}{\sqrt{\sigma_{\mathcal{B}}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Figure 10. The equation behind batch normalisation

2.4 You Only Look Once version 3 (YOLOv3)

YOLOv3 is a state-of-the-art, real-time object detection system developed based around the CNN algorithm with the adoption of many of the modern techniques proven to be effective in other state-of-the-art deep learning models for computer vision such as the residual network, inception network and multi-scale outputs. It inputs the full image, then divides it into regions and predicts bounding boxes, which are weighted by their own predicted probabilities, and object probabilities for each region at three different scales. Different from other prior systems in the same category which repurpose and combined trained classifiers and localisers and suggest detections at high scoring regions of the image. In addition, YOLOv3 and its predecessors were built in the DarkNet framework, the authors' open-source neural network framework written in C and CUDA. Despite the fact that this framework is low-level and fast, it was made to only serve researching purposes and of the author himself, documentations and third-party supports as well as scalability in this framework are very limited to none. [24]

- **Darknet-53**

	Type	Filters	Size	Output
	Convolutional	32	3 × 3	256 × 256
	Convolutional	64	3 × 3 / 2	128 × 128
1×	Convolutional	32	1 × 1	
	Convolutional	64	3 × 3	
	Residual			128 × 128
	Convolutional	128	3 × 3 / 2	64 × 64
2×	Convolutional	64	1 × 1	
	Convolutional	128	3 × 3	
	Residual			64 × 64
	Convolutional	256	3 × 3 / 2	32 × 32
8×	Convolutional	128	1 × 1	
	Convolutional	256	3 × 3	
	Residual			32 × 32
	Convolutional	512	3 × 3 / 2	16 × 16
8×	Convolutional	256	1 × 1	
	Convolutional	512	3 × 3	
	Residual			16 × 16
	Convolutional	1024	3 × 3 / 2	8 × 8
4×	Convolutional	512	1 × 1	
	Convolutional	1024	3 × 3	
	Residual			8 × 8
	Avgpool		Global	
	Connected		1000	
	Softmax			

Figure 11. Darknet-53 architecture [16]

Darknet-53 is the latest of its kind, deep neural network model for image classification. It is the feature extractor that YOLOv3 was built on top off. It was inspired by the custom network based on GoogLeNet architecture, VGG models, and residual network. Darknet-53 has 53 convolutional layers, hence the name, and a total of 18.7 billion operations compared to the state-of-the-art ResNet-152 model with 29.4 billion operations, it is more efficient in terms of speed and on par with the other classifier in terms of performance. The architecture of Darknet-53 has each layer followed by a BN layer and then the Leaky-ReLU activation function. In addition, the convolution layers are fixed padded to the input size and down-sample directly with a *stride (factor)* of 2.

- **Bounding boxes**

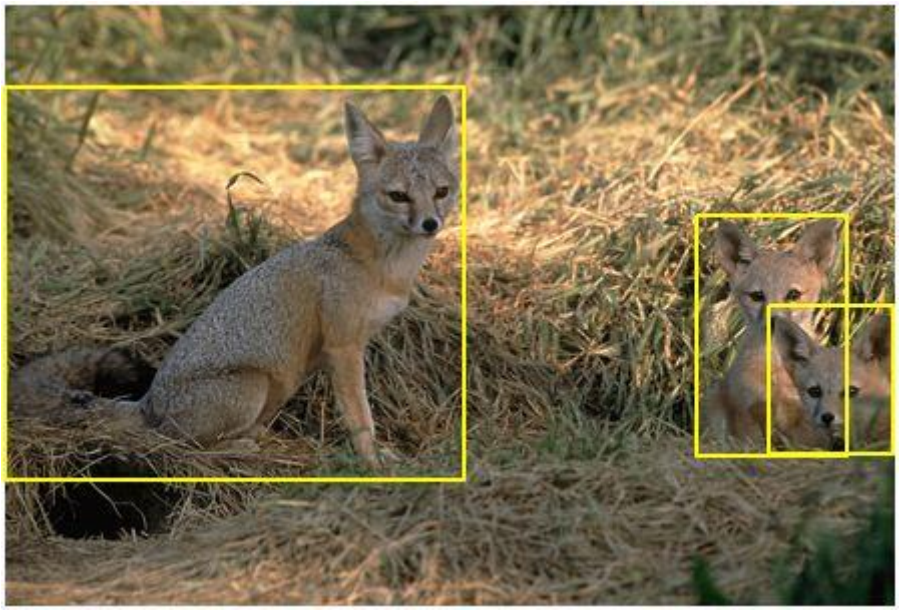

Figure 12. An example of bounding boxes displayed over objects [25]

In the object detection task, there are multiple targets in an input image that not only do they need to be identified but also to obtain their specific locations.in the image and we use bounding boxes to describe those target locations. A bounding box is an imaginary rectangular box around a predicted object and can be determined by either an x and y coordinate of the centre of the box and the factor of the width and height of the box with respect to the image size, or a pair of x and y coordinates of the top left corner and the bottom right corner. Therefore, a bounding box is presented by a set of four real numbers. In addition, collision checking of bounding boxes are sometimes used for some applications. For YOLO, as demonstrated in figure 12, each predicted bounding box has its coordinates include the centre coordinates relative to the grid using a sigmoid function, as well as the width and height. Thus, they need to be transformed using a set of non-linearity transformation functions in order to be visually interpreted.

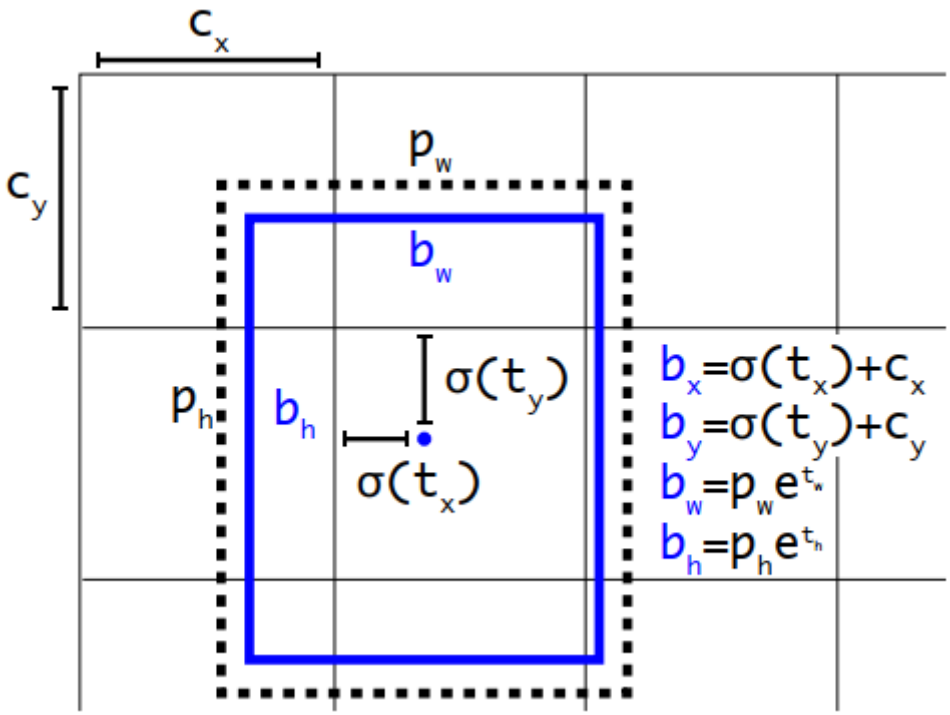

Figure 13. The demonstration of YOLO bounding box and the transformation functions

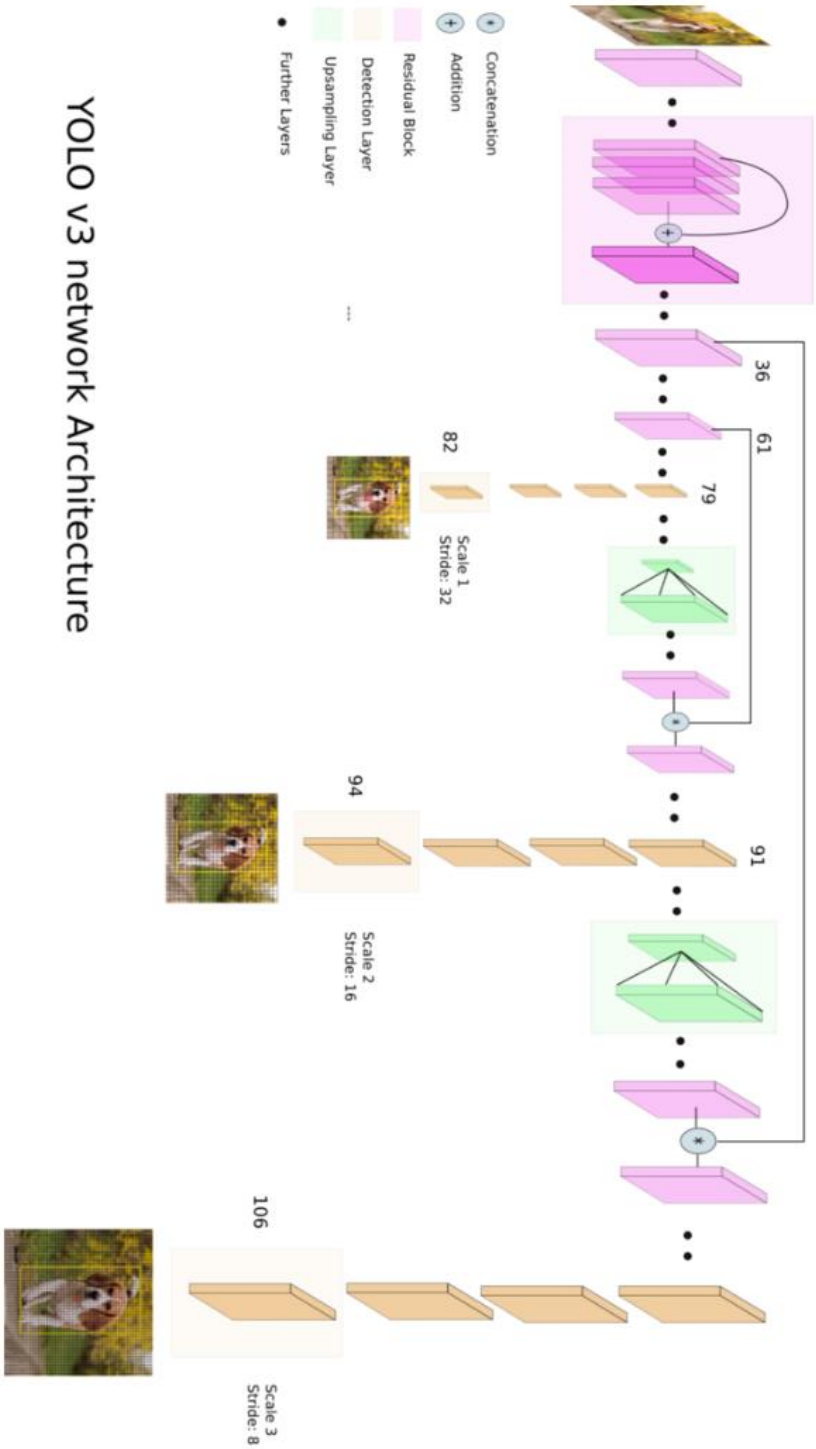

Figure 14. YOLOv3 Architecture [26]

YOLOv3 was built directly on top of the Darknet-53 model, giving it a total of 106 fully convolutional layers with another 53 layers in addition to DarkNet-53. At the outputs, the

detection is done by applying 255 1x1 convolutional filters on the feature volumes of three different scales at three different places in the network.

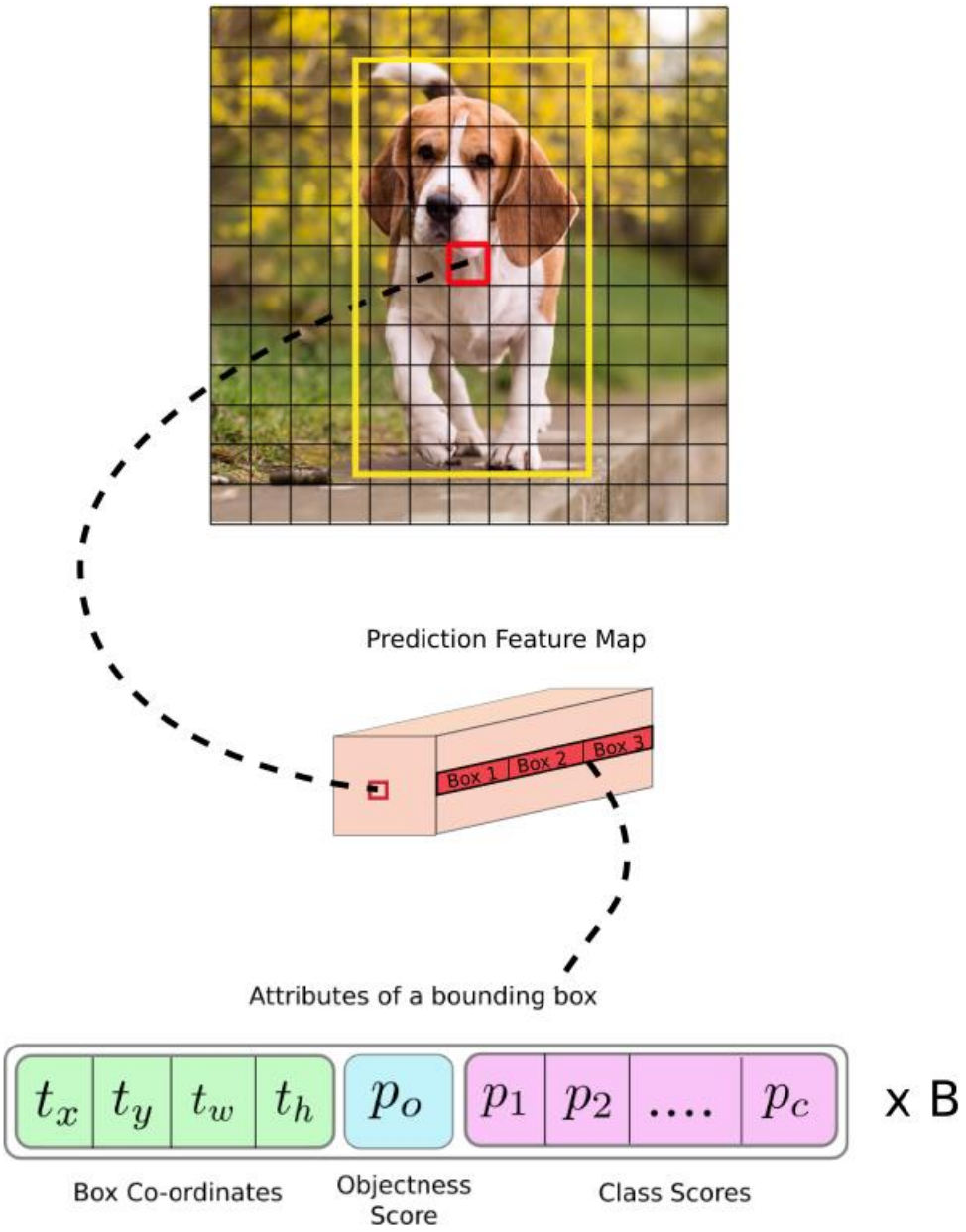

Figure 15. YOLOv3 output vector structure [26]

YOLOv3 down-samples the image to three different scales by a factor of 32, 16, and 8 to make detections. The first 81 layers of the network down-sample the image by a factor of 32 and the first detection is made at the 82nd layer. the network then continues at layer 79 and the output is later upsampled by factor of 2 before going through an inception module where it is concatenated with the output of layer 61, the second detection is made at the

94th layer. In the similar manner, the network continues again at layer 91, upsampled by a factor of 2 and then concatenated inceptionally with output of layer 36 and make the third detection at the 106th layer. If the input image has size 416x416 is taken as an example, the first detections would yield an output volume of shape 13x13x255, the second detection would have shape 26x26x255 and the third detection would be 52x52x255. Each cell of the output matrix a vector of 255 values denoting predictions with respect to three bounding boxes. Each prediction has 85 real numbers in which the first four are the bounding box coordinates, the next one is the probability of whether this bounding box contain any objects – the objectness score, and the last 80 values are the probabilities of the 80 objects that the networks learned to detect. In the end, there is a total of 10647 predictions that this network infers for an image all in one go. This technique effectively improves the detecting capability of the system compared to earlier versions, especially for small objects. Using inception modules for up-sampled layers with earlier layers in the network thoughtfully help preserving detailed information of the image and significantly improve detection capability for smaller objects. By allowing for detections to be performed at three different scales, each output entry is trained only to be responsible for being good at detecting objects at certain sizes. examining the case of the earlier example again, the first output layer of size 13x13 would be considered to detect large object, the second output layer of size 26x26 would be considered to detect medium objects, and the last output layer of size 52x52 would be considered to detect small objects.

2.5 Python

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. It allows for rapid application development by enforcing high-level built in data structures, dynamic typing and dynamic binding syntax, it can also be used as a scripting or glue language to connect existing components together. Python syntax is simple, easy to learn and emphasizes readability and therefore reduces the cost of programme maintenance. Python supports modules and packages, which encourages program modu-

larity and code reuse. The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms and can be freely distributed. Python provides increased productivity as no compilation step is required which makes the edit-test-debug cycle incredibly fast. Furthermore, debugging Python programmes is easy: a bug or bad input will never cause a segmentation fault. Instead, when the interpreter discovers an error, it raises an exception. When the programme doesn't catch the exception, the interpreter prints a stack trace. A source level debugger allows inspection of local and global variables, evaluation of arbitrary expressions, setting breakpoints, stepping through the code a line at a time, and so on. The debugger is written in Python itself, testifying to Python's introspective power. On the other hand, often the quickest way to debug a programme is to add a few print statements to the source: the fast edit-test-debug cycle makes this simple approach very effective. [27]

2.6 TensorFlow

TF is a multi-platform open source platform for machine learning. It is an end-to-end, comprehensive and flexible ecosystem of tools, libraries and community resources that allows for fast experimentation of new ideas, turning concept to code of state-of-the-art models and to publication, and it allows developers easily build and deploy machine learning powered applications. From version 2.0 onwards, TensorFlow enables easy building, training machine learning models at multiple levels of abstraction to choose from without giving up on speed and performance. For getting started, TensorFlow provides the high-level Sequential API Keras which has these advantages: user-friendly, modular, composable, and extendable. It is opted for fast prototyping, advanced researching, and production. For training very large systems and optionally in a production context, the Estimator API provides distribution capability across multiple machines (multi-servers) at the same level of simplicity to Keras. For research and development where having full control is a necessity, the TensorFlow Core API now have Eager Execution

mode enabled by default to provide a define-by-run interface for its operations. TensorFlow is built to provide a direct path to production no matter what language and platform they are and TensorFlow Extended provides a full production ML pipeline, whether it's in the cloud or in the browser where it runs in JavaScript environments, use TensorFlow.js; if it is on mobile and edge devices use TensorFlow Lite; if it is on-prem, simply use any API that is suitable. TensorFlow also supports ecosystems of extension libraries and models including Ragged Tensors – a TensorFlow implementation of nested variable-length lists for easy storing and processing non-uniform shape data, TensorFlow Probability – a Python library for easy combining probabilistic models and deep learning on TPU and GPU hardware, Tensor2Tensor – a library of designed deep learning models and datasets, and BERT **– Bidirectional Encoder Representations from Transformers**, a new method language representations which achieved state-of-the-art results on many recent Natural Language Processing (NLP) tasks. [28]

2.7 NumPy

NumPy is the mathematical library for scientific computing for the Python programming language. it supports multi-dimensional array objects (matrices) with a large collection of sophisticated (broadcasting) operation functions. It also includes tools for integrating C/C++ and Fortran code along with many useful linear algebra, Fourier transform, and random number computations. NumPy can conveniently be used as an efficient multi-dimensional container of generic data and define new arbitrary datatypes. This allows for seamless and fast integration with a wide variety of popular frameworks and databases. Since images with multiple channels (colour images) are typically represented as three – dimensional arrays, operations such as indexing, slicing and masking with other arrays at the pixel level are very efficient. In addition, the NumPy array is adopted by the computer vision library OpenCV as a universal data structure in for storing images and it significantly simplifies the workflow and debugging. On the other

hand, the limitations of this library are that algorithms that are unvectorisable will perform slowly as they must be implemented in pure Python. Furthermore, vectorisation may increase memory requirement complexity from constant to linear as the size of temporary arrays need to be as large as the inputs. [29]

2.8 OpenCV

OpenCV is an Open source software library built to provide a standard infrastructure and accelerate the use of machine learning for computer vision applications. It is widely adopted with an estimated number of exceeding 14 million downloads. It contains more than 2500 of both classic and state-of-the-art optimised algorithms with their applications range from face detection and recognition, object identification, human action classification, movements tracking, 3D model of object extraction, 3D cloud point production from stereo cameras, high resolution image reproduction by stitching images, image similarity search in database, red eye removal in images, eye tracking, scenery recognition, and overlay marker establishment for augmented reality and so on. It is developed by Intel and is free for both academic and commercial use. The library is cross-platform and supports various deep learning frameworks such as TensorFlow, Torch, PyTorch, and Caffe. It has C++, Python, Java and MATLAB interfaces and has supports for Windows, Linux, Mac OS, iOS and Android operating systems. Focusing mainly at real-time performance, the library was implemented in optimized C/C++ and was designed for computational efficiency and can make use of multi-core processing capabilities. Enabled with the OpenCL and CUDA framework, it can also take advantage of the hardware acceleration of the underlying heterogeneous compute platform. [30]

2.9 Intel's Movidius Neural Compute Stick.

Figure 16. Intel's Neural Compute Stick [2]

The Intel® Movidius NCS is a series of small, fan-less, deep learning accelerating hardware that enables learning AI programming for edge devices. The core of this hardware is powered by the first always-on high-performance vision processor in the industry**,** the Intel® Movidius™ VPU. It is the same the one used in a majority of intelligent integrated devices such as security cameras, gesture-controlled drones, industrial machine vision equipment, etc. The NCS series is made to enable rapid prototyping, validation, and deployment of deep learning inference applications at the edge. All these are made possible thanks to the low-power consumption VPU architecture that allows an entirely new segment of AI applications to no longer need to be reliant on a connection to the cloud. This helps lower latency and increase security since all the work stays on your machine rather than being sent to operate at somewhere else. Latest in the line, the NCS2 is based on the next generation of the VPU architecture, the Intel® Movidius™ Myriad™ X VPU which contains two Neural Compute Engines (NCE). The NCE is an on-chip hardware block which is optimised for a high compute-per-watt consumption for visual intelligence processing. This new VPU achieves 1 trillion operations per second (TOPS) of performance, and in machine vision and visual awareness applications, it is 8 times faster over its predecessor in extremely power-constrained environments without any accuracy compromises. [31]

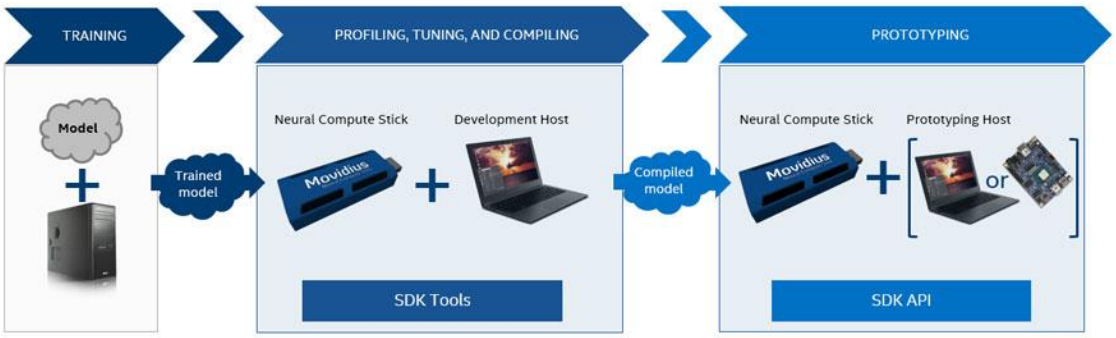

Figure 17. The development workflow diagram for the Intel's Neural Compute Stick [32]

The development process of the NCS is presented in the workflow diagram of figure 16. A model is designed and trained on a capable host machine (a development computer). Then use one of the supported APIs to profile, tune, and compile to convert the model to the format supported by the VPU hardware. With the newly formatted model, users can validate the model and prototype its applications on the same host machine or on edge devices. The hardware is physically connected to the machine and is accessed using available methods in the API.

2.10 Intel® Movidius™ Neural Compute Software Development Kit (NCSDK)

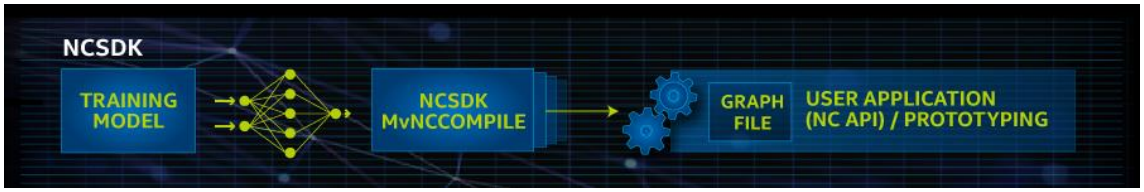

Figure 18. The workflow diagram of the NCSDK [33]

Solutions that use the NCS can be programmed using the NCSDK. It contains a set of functions to help compile, profile, and tune, which enable rapid prototyping and deployment for CNN based applications that require real-time inferencing at ultra-low power. The workflow is summarised in figure 17, DL models after being developed in libraries such as Caffe and TensorFlow can be used with the NCSDK. Functions in this framework can be used to profile, validate and compile the model to the model graph file that can be loaded by the NCS hardware. However, the NCSDK is restricted to a list of supported

CNN models and the capability of deploying custom models is still limited. To make up for that incompatible difficulty, Intel suggests transitioning to the Intel® OpenVINO™ toolkit. which is also the default software toolkit for the NCS2.

2.11 Intel® Distribution of OpenVINO™ toolkit

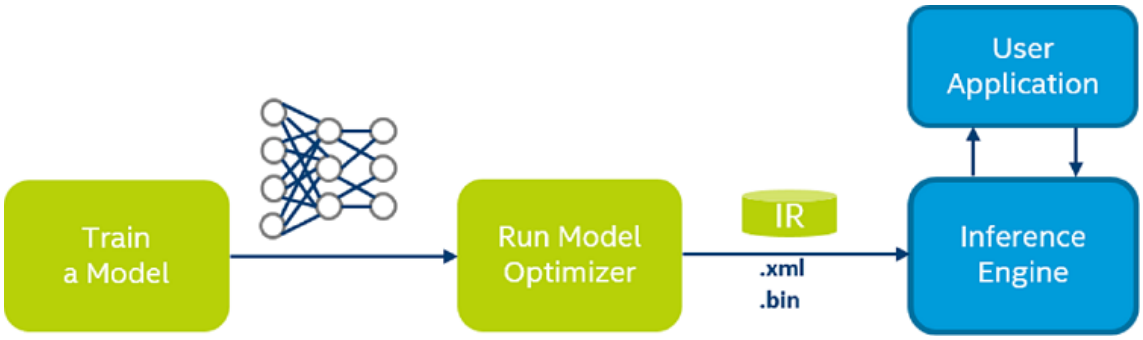

Figure 19. The workflow diagram of the OpenVINO toolkit [34]

OpenVINO is a primary development software toolkit for the NCS2 and other Intel hardware. It allows for developing and deploying machine vision solutions that delivers high inferencing speed and accuracy. OpenVINO integrates camera processing, optimized DL computation, and CV acceleration tools for heterogeneous execution environments. This means that, CNN-based solutions using this toolkit can maximise performance by extending their workloads across the Intel hardware (including CPUs, GPUs, FPGAs, VPUs, and IPUs) using a just a common API. Compared to NCSDK, OpenVINO also enables CNN-based inference at the edge but with better preoptimized kernels and calls for OpenCV API. The development workflow in OpenVINO is depicted in figure 18. It first includes training a CNN model in one of the machine learning libraries. Second, use model optimizer to produce the Intermediate Representation (IR) model graph. The IR contains two files, the topology description in XML format and the binary data of the weights the model. The IR can be used to read, load, and infer using the Inference Engine. The inference engine API contains unified functions to work across multiple Intel platforms. User applications can integrate this API to use the model IR to perform deep learning inference. [34]

2.12 Raspberry Pi

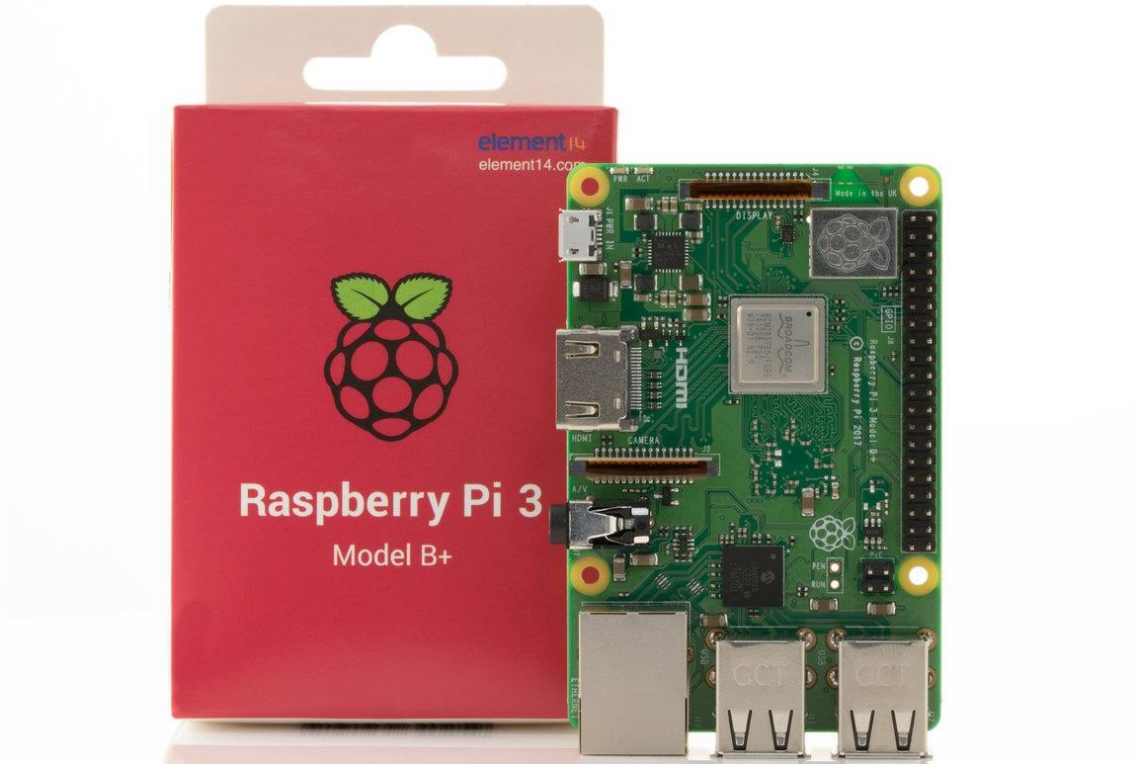

Figure 20. The Raspberry Pi 3 Model B+

The RPI is the third most popular micro-computer worldwide. This compact system comes in various models with different shape and sizes as well as the computing power. In terms of the latest most powerful model 3 B+, It has a size of a normal credit card which is 85.60mm x 56mm x 21mm and weights 45g. In terms of hardware specification, this system uses a 1.4GHz 64-bit quad-core ARM Cortex-A53 CPU, the BCM2837B0 SoC from Broadcom. In addition, it has 1 GHz of RAM, 4 USB ports, a micro USB port for power source, a microSD card slot, an HDMI port, an 1000Base-T(gigabit) Ethernet port, an integrated 802.11ac/n wireless LAN chip, Bluetooth 4.2, and 26 GPIO pins. The Raspberry Pi 3 B+ packed a GPU capable of 24 GFLOPs of general-purpose compute, features texture filtering and DMA infrastructure. In addition, it provides OpenGL ES 2.0, hardware accelerated OpenVG, H.264 high-profile encode and decode at 1080p 30fps. To compare, this micro-computer roughly matches a 300MHz Pentium 2 but has better graphics and unexpandable RAM as it is built as a Package on Package (POP) on top of the SoC. The officially recommended operating system for the Raspberry Pi is the

Raspbian OS Linux distro. It is specifically designed and optimised for the hardware across all Raspberry Pi micro-computers. Additionally, later versions of RPI computers are getting more powerful and the community has made a significant amount of efforts to compatibilise many other popular Linux distros, Android OS versions, and even Windows to many of the Raspberry Pi models. The RPI devices are powered by 5V power source and the recommended SPU current capacity is specific to each model and for the use case. For the latest model B+, the recommended SPU is 2.5A. The more interfaces there is in the device, the more power is required. To connect high-power USB devices, it is recommended to connect them to a powered USB hub then connect it to the Pi. In addition, very high-current devices or devices which draws a surge current such as network modems and USB hard disks will require an external power supply. In terms of storage, the Raspberry Pi uses a microSD card of 8GB at minimum and up to 128GB at maximum. The storage contains the operating system, additional packages and programs, and files. The Raspberry Pi 3 B+ can be extended with a number of modules. There includes HD camera, touchscreen display, TV signal controller, as well as standard computer peripherals such as mouse and keyboard. [35]

3 IMPLEMENTATION

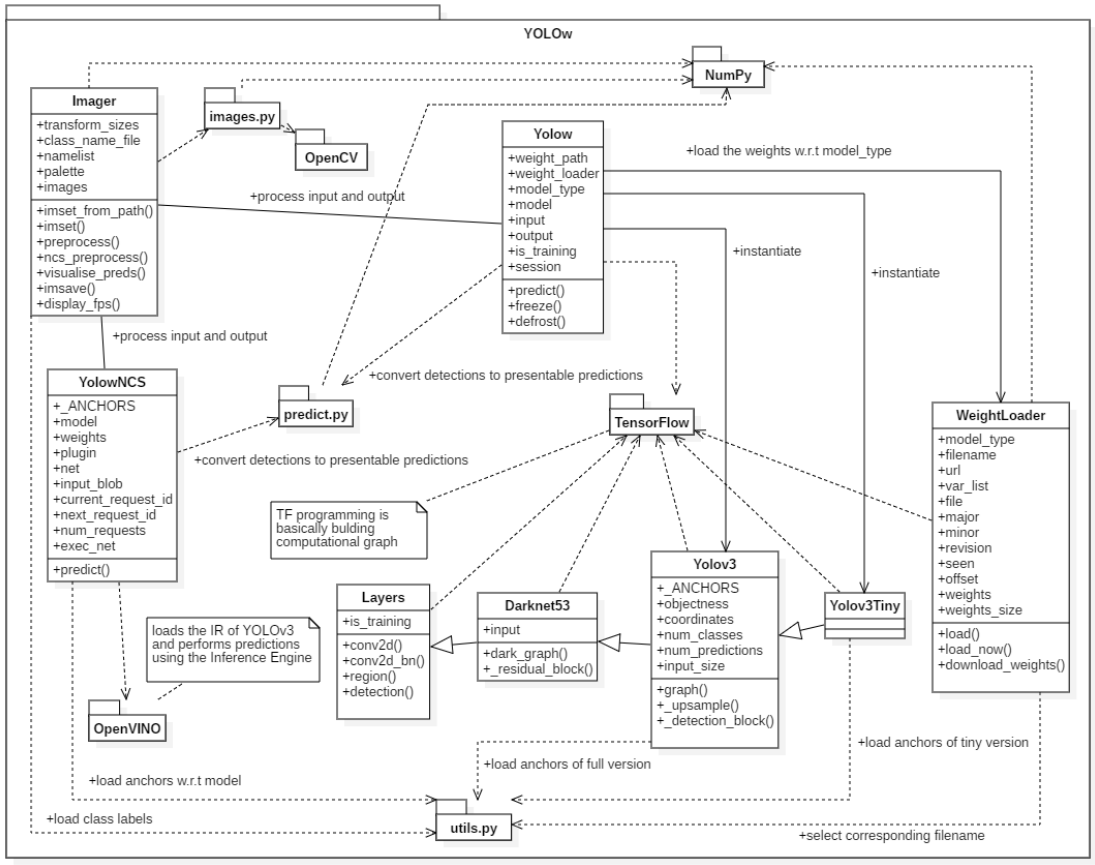

Figure 21. The code hierarchy diagram of YOLOw

Figure 20. describes the general structure of code. It can be simplistically divided into 2 main class objects. The first one is, based on TF, called *Yolow* and the other one is, based on OpenVINO, called *YolowNCS*. Primarily, *Yolow* contains many modules responsible to do the following things. Firstly, it instantiates the TF operational graph of either *YOLOv3 or YOLOv3-tiny* network architecture. Secondly, it loads the target model of either of these networks trained in the *Darknet* framework to the TF graph. Thirdly, it freezes the graph into a model file in *protobuf* format, it can be easily and quickly loaded back to run or integrate to other projects. Additionally, this frozen model file is required to create the IR model graph for *YolowNCS*. Finally, *Yolow* takes as input pre-processed visual data and performs detection inferencing. Being designed to opt for a similar using experience, *YolowNCS* first handles the connection with the NCS device hardware, then

initialises the network from the IR graph and load it to the NCS, finally it handles detection inferencing in either synchronous or asynchronous manner.

3.1 TensorFlow

3.1.1 Class *Layers*

In detail, all of the network building blocks of the YOLO architectures are sub-classes of the elementary class *Layers*. It has two rudimentary methods that actually builds YOLOv3 architectures.

```
class Layers(object):

    def __init__(self, is_training=False):
        self.is_training = is_training
```

Snippet 1. The constructor of Layers class

In the constructor, class *Layers* requires the argument is_training whose value is default to *False*. It is for *the* batch_normalisation() function to determine the appropriate operations based on whether the computations are meant for training or inferencing. If it is the prior, is_training should be set to *True* and the returned output is normalised with statistics of the currently computing batch of input. Otherwise, the returned output is normalised using moving statistics for a single input.

```
    def conv2d(self, input, num_kernels, kernel_size=1, strides=1,
with_bias=False):
        with tf.variable_scope(name_or_scope=None,default_name='conv2d'):
            if kernel_size > 1:
                paddings = tf.constant([[0,0], [1,1], [1,1], [0,0]])
                input = tf.pad(input, paddings)
            W = tf.get_variable(name='weights', shape=[kernel_size, ker-
nel_size, input.shape[-1], num_kernels], dtype=tf.float32)
            z = tf.nn.conv2d(input=input,
                             filter=W,
                             strides=[1, strides, strides, 1],
                             padding='VALID')
            if with_bias:
                bias = tf.get_variable(name='biases',
                                       shape=num_kernels,
                                       dtype=tf.float32,
                                       initializer=tf.zeros_initializer)
```

```
                z += bias
            return z
```

Snippet 2. The implementation of the conv2d() method

The conv2d() method represents a convolutional layer of YOLOv3, in which the input tensor is custom padded by a border of thickness 1 and the kernel tensor *w* is created using the passed in arguments. They are then passed to the tf.nn.conv2d() function to perform the convolutional operations, its output is then added by the bias term according to formula (0).

```
    def conv2d_bn(self, input, num_kernels, kernel_size=3, strides=1):
        z = self.conv2d(input=input,
                        num_kernels=num_kernels,
                        kernel_size=kernel_size,
                        strides=strides)
        z_tilder = tf.layers.batch_normalization(inputs=z,
                                                 momentum=.9,
                                                 epsilon=1e-05,
                                                 training=self.is_training)
        return tf.nn.leaky_relu(z_tilder, alpha=.1)
```

Snippet 3. The implementation of the conv2d_bn() method

The conv2d_bn() method represents another convolutional layer in the architectures that is similar to the other one but is extended with a BN and a activation operations. The BN operation is computed by Function tf.layers.batch_normalization() which takes its hyperparameters *momentum* and *epsilon* set according to the Darknet configuration of the architecture and the provided argument is_training. The activation operation is computed by function tf.nn.leaky_relu() with the hyperparameter *alpha* followed the Darknet configuration, it applies the Leaky ReLU non-linearity transformation function to the input.

3.1.2 Class Darknet53

```
class Darknet53(Layers):
    def __init__(self, input, is_training=False):
        super().__init__(is_training=is_training)
        self.input = input
```

Snippet 4 the constructor of the Darknet53 class

The Darknet53 class sub-classes Layers to effectively inherit its methods and construct the DarkNet-53 model part of the YOLOv3 architecture. The class has its constructor requires two arguments: *input* of the visual data and is_training for the superclass.

```
    def _residual_block(self, input, num_kernels):
        shortcut = input
        input = self.conv2d_bn(input=input,
                               num_kernels=num_kernels,
                               kernel_size=1)
        input = self.conv2d_bn(input=input,
                               num_kernels=num_kernels * 2)
        input += shortcut
        return input
```

Snippet 5. The implementation of the _residual_block() method

Darknet-53 has a helper method _residual_block() which allows easily building the repetitive residual module of the architecture. This module comprises two conv2d_bn() layers with a shortcut (skipped) connection, it is an addition operation of the output of at the second conv2d_bn() layer and the input to the first conv2d_bn() layer.

```
    def dark_graph(self):
        input = self.conv2d_bn(input=self.input,
                               num_kernels=32)
        input = self.conv2d_bn(input=input,
                               num_kernels=64,
                               strides=2)
        input = self._residual_block(input=input,
                                     num_kernels=32)
        input = self.conv2d_bn(input=input,
                               num_kernels=128,
                               strides=2)
        for _ in range(2):
            input = self._residual_block(input=input,
                                         num_kernels=64)
        input = self.conv2d_bn(input=input,
                               num_kernels=256,
                               strides=2)
        for _ in range(8):
            input = self._residual_block(input=input,
                                         num_kernels=128)
        route_3 = input
```

```
        input = self.conv2d_bn(input=input,
                               num_kernels=512,
                               strides=2)
        for _ in range(8):
            input = self._residual_block(input=input,
                                         num_kernels=256)
        route_2 = input
        input =self.conv2d_bn(input=input,
                               num_kernels=1024,
                               strides=2)
        for _ in range(4):
            input = self._residual_block(input=input,
                                         num_kernels=512)
        return input, route_2, route_3
```

Snippet 6. The implementation of the dark_graph() method

The other method is dark_graph() which actually builds the TF graph of the model without the final layer for classification (52 layers). The graph is an consecutive arrangement of conv2d_bn() layers and _residual_block() that effectively down-sample the data. The end output is three nodes at three difference scales of the process.

3.1.3 Class *Yolov3*

```
class Yolov3(Darknet53):
    num_anchors = 3
    objectness = 1
    coordinates = 4
    def __init__(self, input, num_classes, input_size, anchor_file,
is_training):
        super().__init__(input=input, is_training=is_training)
        self._ANCHORS = load_anchors(anchor_file)
        self.num_classes = num_classes
        self.num_predictions = self.objectness + self.coordinates +
self.num_classes
        self.input_size = self.input.get_shape().as_list()[1:3]
```

Snippet 7. The constructor of the Yolov3 class

The same design philosophy is applied to the *Yolov3* class to inherit from Darknet-53. In the constructor, Yolov3 requires 5 arguments: input of visual data, is_training for its super class, num_classes for the number of classes, input_size for the size of the visual input, anchor_file for the path to the anchor file. In addition, the *Yolov3* class also defines three

constant values: num_anchors for the number of anchors in use, *objectness* for the confidence of the detection existence, and coordinates for the number of coordinates defining bounding boxes. The anchor_file argument is used to load the list of anchors using load_anchors() function imported from utils module. The class includes the graph building method graph(), the block module method _detection_block(), the two layer methods _upsample() and _region().

```
    def graph(self):
        with tf.variable_scope('darknet53'):
            route_1, route_2, route_3 = self.dark_graph()
        with tf.variable_scope('yolov3'):
            predictions_1, route = self._detection_block(route_1, 512,
self._ANCHORS[6:])
            route = self.conv2d_bn(input=route,
                                   num_kernels=256,
                                   kernel_size=1)
            route = self._upsample(route)
            route_2 = tf.concat([route, route_2], axis=-1)
            predictions_2, route = self._detection_block(route_2, 256,
self._ANCHORS[3:6])
            route = self.conv2d_bn(input=route,
                                   num_kernels=128,
                                   kernel_size=1)
            route = self._upsample(route)
            route_3 = tf.concat([route, route_3], axis=-1)
            predictions_3, route = self._detection_block(route_3, 128,
self._ANCHORS[:3])
        return tf.concat([predictions_1, predictions_2, predictions_3],
axis=1, name='output')
```

Snippet 8. The implementation of the graph() method

```
    def _region(self, input, anchor_list):
        output_shape = input.get_shape().as_list() # if output_shape=(m,
13, 13, 255)
        grid_sz = output_shape[1:3] # grid_sz = 13, 13
        grid_dim = np.prod(grid_sz) # grid_dim = 169
        strides = np.array(self.input_size) / np.array(grid_sz)

        input = tf.reshape(input, [-1, grid_dim*self.num_anchors,
self.num_predictions]) # predictions = (m, 507, 85)

        bb_xy, bb_hw, confidence, classes = tf.split(input, [2, 2, 1,
self.num_classes], axis=-1) # split along the last dimension
```

```
        bb_xy = tf.sigmoid(bb_xy)
        confidence = tf.sigmoid(confidence)
        classes = tf.sigmoid(classes)

        grid_x_range = tf.range(grid_sz[0], dtype=tf.float32)
        grid_y_range = tf.range(grid_sz[1], dtype=tf.float32)
        grid_x, grid_y = tf.meshgrid(grid_x_range, grid_y_range)
        grid_x = tf.reshape(grid_x, [-1, 1])
        grid_y = tf.reshape(grid_y, [-1, 1])
        grid_offset = tf.concat([grid_x, grid_y], axis=-1)
        grid_offset = tf.tile(grid_offset, [1, self.num_anchors])
        grid_offset = tf.reshape(grid_offset, [1, -1, 2])
        bb_xy = bb_xy + grid_offset
        bb_xy = bb_xy * strides
        anchors = [tuple(a/strides) for a in anchor_list]
        anchors = tf.tile(anchors, [grid_dim, 1])

        bb_hw = anchors * tf.exp(bb_hw) * strides
        return tf.concat([bb_xy, bb_hw, confidence, classes], axis=-1)
```

Snippet 9. The implementation of the _region() method

Under graph(), method dark_graph() is called within the variable scope ‘darknet53’ which returns the three route output. Within the variable scope ‘yolov3’, every route is in turn combined with the shortcut output of the last route, except for the first one, and is fed to a scheme of layers with each one uses a different kernel for a certain scale. A route goes into a _detection_block() and outputs the detections at that scale along with the shortcut route. The shortcut is then subject to a conv2d_bn() layer before being upscaled by the _upsample() and is concatenated with the next _dark_graph() route. At the end of the method, predictions from all three branches are concatenated and returned.

```
    def _detection_block(self, input, num_kernels, anchor_list):
        output_depth = self.num_anchors * self.num_predictions
        for i in range(3):
            input = self.conv2d_bn(input=input,
                                   num_kernels=num_kernels,
                                   kernel_size=1)
            if i == 2:
                route = input
            input = self.conv2d_bn(input=input,
                                   num_kernels=num_kernels*2)
        input = self.conv2d(input=input,
                            num_kernels=output_depth,
                            with_bias=True)
```

```
        input = self._region(input, anchor_list)
        return input, route
```

Snippet 10. The implementation of the _detection_block() method

In terms of the _detection_block() method, it implements six conv2d_bn() layers and one conv2d() layer, which is meant to transform the feature maps to the proprietary shape to be processed by _region(). In addition, output of the fifth conv2d_bn() layer is returned as a shortcut to be concatenated with other feature maps in the later layer in the network whilst the main output goes to the _region() method. This method splits the input into volumes of detection attributes: anchors_xy – centres of bounding boxes, anchor_hw – size of bounding boxes, confidence – the detection confidence score for bounding boxes, and classes – the confidence scores of every class with respect to every bounding box. Based on figure 12, the bounding box attributes are transformed into standardised values and re-concatenated to be returned. This way of doing saves some computing as each attribute volume does not have to be concatenated separately in method graph().

```
    def _upsample(self, input, stride=2):
        grid_shape = input.get_shape().as_list()
        grid_sz = grid_shape[1:-1]
        output_dims = grid_sz*stride
        return tf.image.resize_nearest_neighbor(images=input,
                                                size=output_dims,
                                                name='upsampled_' +
str(output_dims))
```

Snippet 11. The implementation of the _upsample() method

Layer method _upsample() returns the up-sampled feature maps of input, up-sampling uses a factor (stride) of 2 and the k-nearest neighbour algorithm for interpolation.

3.1.4 Class Yolov3Tiny

The next class to inherit Yolov3 is the tiny version of the Yolov3 architecture, Yolov3Tiny. This compact architecture carries the same terminology but with fewer layers and has only two output branches. This version strives for light computing power and commits to sacrifice some accuracy and detection capability for small objects compared to the full version. This architecture will be more friendly for small devices to run more efficiently. The implementation details are as follow.

```
class Yolov3Tiny(Yolov3):
    def graph(self):
        with tf.variable_scope('yolov3_tiny'):
            input = self.conv2d_bn(input=self.input,
                                   num_kernels=16)
            for i in range(6):
                input = tf.layers.max_pooling2d(inputs=input,
                                                pool_size=2,
                                                strides=(1 if i == 5 else
2),
                                                padding='same' if i == 5
else 'valid')
                input = self.conv2d_bn(input=input,
                                       num_kernels=pow(2, 5+i))
                if i == 3:
                    route_1 = input

            input = self.conv2d_bn(input=input,
                                   num_kernels=256,
                                   kernel_size=1)
            route_2 = input
            predictions_1 = self._detection_block(input=input,
                                                  num_kernels=512,
                                                  anchor_list=self._AN-
CHORS[3:6])
            input = self.conv2d_bn(input=route_2,
                                   num_kernels=128,
                                   kernel_size=1)
            input = self._upsample(input)
            input = tf.concat(values=[input, route_1],
                              axis=-1)
            predictions_2 = self._detection_block(input=input,
                                                  num_kernels=256,
                                                  anchor_list=self._AN-
CHORS[:3])
        return tf.concat(values=[predictions_1, predictions_2],
                         axis=1,
                         name='output')
```

Snippet 12. Implementation of the graph() method of class Yolov3Tiny

Since the class has the same structure as its superclass, it overwrites the graph() method and the _detection_block() method to load its smaller network. For the first method, the feature maps are down-sampled by the factor of 2 by the max-pooling layer instead of the

convolution layers. This helps reduce the computing resources and, in the end, only two prediction volumes are concatenated and returned.

```
    def _detection_block(self, input, num_kernels, anchor_list):
        output_depth = len(anchor_list)*self.num_predictions
        input = self.conv2d_bn(input=input,
                               num_kernels=num_kernels)
        input = self.conv2d(input=input,
                            num_kernels=output_depth,
                            with_bias=True)
        input = self.region(input, anchor_list)
        return input
```

Snippet 13. The implementation of _detection_block() method for Yolov3Tiny class

For method _detection_block(), it only implements two conv2d_bn() layers before the _region() layer.

3.1.5 Class *WeightLoader*

When the network graph of YOLO is initialised, it needs to be loaded with the Darknet weights file before being used for making predictions. In order to do that, class Weight-Loader was implemented to read the binaries in the weights file to floating point numbers of bytes and assign the right numbers to the corresponding operations in the model graph of TensorFlow.

```
class WeightLoader(object):
    default_models = load_default_models()
    def __init__(self, var_list, weight_file):
        self.file = Path(weight_file)
        self.filename = os.path.basename(weight_file)
        self.var_list = var_list
        self.major = 0
        self.minor = 0
        self.revision = 0
        self.seen = 0
        self.offset = 0
        self.weights = []
        self.weights_size = 0
        print('\n\nChecking weights from {}\n'.format(self.file))
        if not self.file.is_file():
            print('\n{} was not available! Start to download from the in-
ternet.'.format(self.file))
```

```
            self.url = 'https://pjreddie.com/media/files/' + self.file-
name
            self.download_weights()

        print('\n\nLoading weights from {}\n'.format(self.file))
```

Snippet 14. The constructor of the WeightLoader class

An instance of class WeightLoader requires the list of TF graph operations and the path to the Darknet weights file. In the constructor, it tries to load the weights file and in case of failure, it will automatically try to download that file from the internet on the based on the file name. The class has three methods load(), load_now(), and download_weights(), in which the first two methods is used to perform the assignment operation and the third one is used to download the weights file.

```
    def load(self, var):
        var_shape = var.shape.as_list()
        var_size = np.prod(var_shape)
        read_from = self.offset
        read_to = read_from + var_size
        val = self.weights[read_from:read_to]
        if 'weights' in var.name:
            val = val.reshape(var_shape[3], var_shape[2], var_shape[0],
var_shape[1])
            val = np.transpose(val, (2, 3, 1, 0))

        else:
            val = val.reshape(var_shape)

        self.offset = read_to
        return tf.assign(var, val, validate_shape=True)
```

Snippet 15. The implementation of the load() method

Method *load()* is intendedly for *load_now()* to use as what it does it actually sign the bytes of values in the weights file to the passing operation variable argument.

```
    def load_now(self):
        with open(self.file, 'rb') as f:
            self.major, self.minor, self.revision = np.fromfile(f,
dtype=np.int32, count=3)
            self.seen = np.fromfile(f, dtype=np.float64, count=1)
            self.weights = np.fromfile(f, dtype=np.float32)
            self.weights_size = self.weights.shape[0]
```

```
        load_ops = []
        now = 0
        while now < len(self.var_list):
            var_now = self.var_list[now]
            if 'weights' in var_now.name:
                next = now + 1
                var_next = self.var_list[next]
                if 'batch_normalization' in var_next.name:
                    num_bn_vars = 4
                    gamma, beta, moving_mean, moving_variance =
self.var_list[next:next+num_bn_vars]
                    bn_vars = [beta, gamma, moving_mean, moving_variance]
                    for var in bn_vars:
                        load_ops.append(self.load(var))
                        print('{} variable loaded -- read {}/{} total
bytes.'.format(var.name, self.offset, self.weights_size))
                    now += num_bn_vars
                elif 'biases' in var_next.name:
                    load_ops.append(self.load(var_next))
                    print('{} variable loaded -- read {}/{} total
bytes.'.format(var_next.name, self.offset, self.weights_size))
                    now = next
                else:
                    mess = 'Encountered unexpected next variable
{}.'.format(var_next.name)
                    assert Exception(mess)
                load_ops.append(self.load(var_now))
                print('{} variable loaded -- read {}/{} total
bytes.'.format(var_now.name, self.offset, self.weights_size))
                now += 1
                print('total loaded variables = ' + str(now))
            else:
                mess = 'Encountered unexpected variable {}'.for-
mat(var_now.name)
                assert Exception(mess)
        print('Done!')
        return load_ops
```

Snippet 16. The implementation of the load_now() method

In terms of load_now(), it cycles through each of the TF graph operations, takes into consideration of how the order of variables in the Darknet weights file are laid out, and assign their values correspondingly to the variables of that operation.

```
    def download_weights(self):
        def reporthook(blocknum, blocksize, totalsize):
            readsofar = blocknum * blocksize
            if totalsize > 0:
                percent = readsofar * 1e2 / totalsize
                s = "\r%5.1f%% %*d / %d" % (
                    percent, len(str(totalsize)), readsofar, totalsize)
                sys.stderr.write(s)
                if readsofar >= totalsize: # near the end
                    sys.stderr.write("\n")

            else: # total size is unknown
                sys.stderr.write("\r_%% %d/Unknown\n" % (readsofar))

        print('Downloading ' + self.filename + ' from ' + self.url)
        urlretrieve(self.url, filename=self.file, reporthook=reporthook)
        print('Download complete!')
```

Snippet 17. The implementation of the download_weights() method

Lastly, final method download_weight() is directly call by the constructor when the passing file is not available, which makes it more convenient for the user.

3.1.6 Class *Yolow*

To serve as a high-level programming interface, the Yolow module was implemented to wrap the code of all the functionalities for ease of use and future upgradability under the hood while keeping the application code the same. The module contains an argument parser function and the actual Yolow class itself.

```
class Yolow(object):
    sess = tf.Session()
    default_models = load_default_models()
    model_types = list(default_models.keys())
    freeze_dir = 'data/pb/'
    tf_models = {model_types[0]: Yolov3,
                 model_types[1]: Yolov3Tiny}

    def  __init__(self, model_type, model_file, anchor_file, num_classes,
input_size, is_training=False):
        if model_type not in self.model_types:
            raise ValueError('model_type can only be either \'full\' or
\'tiny\'.')
```

```
        elif not model_type:
            model_type = self.model_types[0]
        self.model_type = model_type

        if not model_file:
            model_file = './data/bin/{}'.format(self.default_mod-
els.get(model_type))

        elif not os.path.exists(model_file):
            raise ValueError('model file {} does not exist.'.for-
mat(model_file))
        self.model_file = model_file

        if '.pb' not in self.model_file:
            self.frozen_filename = '_'.join(['frozen',
os.path.basename(self.model_file).split('.')[0]])
            self.frozen_filename = self.freeze_dir + self.frozen_filename
+ '.pb'
        if os.path.exists(self.frozen_filename):
            self.defrost()
            self.input = tf.get_default_graph().get_tensor_by_name('im-
port/input:0')
            self.output = tf.get_default_graph().get_tensor_by_name('im-
port/detections/output:0')

        else:
            if not anchor_file:
                anchor_file = 'data/anchors/' + self.model_type + '.txt'

            elif not os.path.exists(anchor_file):
                raise ValueError('{} anchor file does not exist.'.for-
mat(anchor_file))


            self.anchor_file = anchor_file
            if not input_size:
                input_size = 416

            if type(input_size) is int:
                input_size = input_size, input_size

            self.input_size = input_size
            self.num_classes = num_classes
            self.is_training = is_training
            self.input = tf.placeholder(tf.float32,
                                        [None, self.input_size[0],
```

```
                                        self.input_size[1], 3],
                                        'input')
            self.model = self.tf_models[self.model_type](self.input,
                                                          self.num_classes,
                                                          self.input_size,
                                                          self.anchor_file,
                                                          self.is_training)

            with tf.variable_scope('detections'):
                self.output = self.model.graph()
            self.loader = WeightLoader(tf.global_variables('detections'),
self.model_file)
            # self.sess.run(tf.global_variables_initializer())
            self.sess.run(self.loader.load_now())
            self.freeze()
```

Snippet 18. The constructor of the Yolow class

The Yolow class first defines some major constant variables sess – TensorFlow session, default_models – a dictionary of the model names and their default weight files, model_types – a list of YOLOv3 model architectures, freeze_dir – the path to the directory where frozen models will be stored, and tf_models – a dictionary of the YOLOv3 models. In the constructor, it first checks the validity of the parameters and whether the frozen model already exists. if so, it will be defrosted to use directly which save time and resources. Otherwise, Yolow will initialise the model by building the network graph and create a WeightLoader instance to load the Darknet weights file from the passing information. When done it will automatically make the frozen model.

```
    def predict(self, input_list, confidence_theshold=.6, iou_thresh-
old=.5):
        feed_dict = {self.input: input_list}
        batch_detections = self.sess.run(self.output, feed_dict)
        return predict(batch_detections, confidence_theshold, iou_thresh-
old)
```

Snippet 19. The implementation of the predict() method

```
    def freeze(self):
        graph_def = tf.graph_util.convert_variables_to_con-
stants(sess=self.sess,
input_graph_def=tf.get_default_graph().as_graph_def(),
output_node_names=['detections/output'])
        if not os.path.exists(self.freeze_dir):
            os.makedirs(self.freeze_dir)
        with tf.gfile.GFile(self.frozen_filename, 'wb') as f:
             f.write(graph_def.SerializeToString())
```

Snippet 20. The implementation of the freeze() method

```
   def defrost(self):

        print('Found frozen model {}, defrost and use!'.format(self.fro-
zen_filename))
        with tf.gfile.GFile(self.frozen_filename, 'rb') as f:
            graph_def = tf.GraphDef()
            graph_def.ParseFromString(f.read())
        tf.import_graph_def(graph_def)
```

Snippet 21. The implementation of the defrost() method

The class also implements three methods, those are predict() – performs predictions and transforms them into valid detections on a batch of input , freeze() – write the graph definition of the model to a file, defrost() – loads the graph definition of the model and import it to the default graph for use.

To provide a mean of supports to help Yolow handle image related operations, there is Imager class. This is a convenient object wrapper of all functions in the images module. The implementation details are as follow:

```
class Imager(object):

    def __init__(self, transform_sizes, labels):
        if type(transform_sizes) is int:
            self.transform_sizes = (transform_sizes, transform_sizes)

        else:
            self.transform_sizes = transform_sizes

        self.labels = labels
```

```
        self.palette = np.random.randint(0, 256, (len(self.labels),
3)).tolist()
```

Snippet 22. The constructor of the Imager class

This class requires two auguments transform_sizes – the YOLOw input size to pre-process and visualise detections for images, and labels – a list of labels for visualisation. It has seven methods, those are:

imset_from_path() – loads all the images from the provided path and set it to the class instance, it is handled by the imread_from_path() function.

```
def imread_from_path(im_path):
    p = Path(os.path.abspath(im_path))
    if os.path.isdir(p):
        ims = [p.joinpath(imname) for imname in os.listdir(p)]
    else:
        ims = [p]
    return [cv2.imread(str(im), cv2.IMREAD_COLOR) for im in ims]
```

Snippet 23. Implementation of function imread_from_path()

This function conveniently opens a directory of images and stores them in a Python list. imset() – set image input to class instance; preprocess() – calls function improcess() to prepare input batch for feeding to YOLOv3 models, it includes resizing and padding the image; ncs_preprocess() – also calls improcess() but sets the function not to perform BGR2RGB operation as well as normalisation.

```
def imresize(im, to_sizes):
    if type(to_sizes) is int:
        to_sizes = (to_sizes, to_sizes)

    im_h, im_w, _ = im.shape
    to_w, to_h = to_sizes
    scale_ratio = min(to_w/im_w, to_h/im_h)
    new_im = cv2.resize(im,(0, 0), fx=scale_ratio, fy=scale_ratio, inter-
polation=cv2.INTER_CUBIC)
    new_h, new_w, _ = new_im.shape

    padded_im = np.full((to_h, to_w, 3), 128)
    x1 = (to_w-new_w)//2
    x2 = x1 + new_w
    y1 = (to_h-new_h)//2
    y2 = y1 + new_h
```

```
    padded_im[y1:y2, x1:x2, :] = new_im

    return padded_im
```

Snippet 24. The implementation of the imresize() function

```
def improcess(ims, to_sizes, to_rgb=True, normalise=True):
    imlist = []
    for im in ims:
        if to_rgb:
            im = cv2.cvtColor(im, cv2.COLOR_BGR2RGB)
        im = imresize(im, to_sizes)
        imlist.append(im)
    imlist = np.array(imlist)
    if normalise: imlist = imlist / 255
    return imlist
```

Snippet 25. Implementation of the impreprocess() function

visualise_preds() – takes predicted detections pred_list and call function visualise() on them to displays on the input set of images; this function in turn makes calls to function rescale_vertex() to scale the predicted bounding box coordinates according to the scale of the input images, and function add_overlays() to draw those bounding boxes from the coordinates.

```
def rescale_vertex(vtx, from_wh, to_wh):
    if from_wh is int:
        from_wh = (from_wh, from_wh)
    if to_wh is int:
        to_wh = (to_wh, to_wh)
    from_wh = np.array(from_wh)
    to_wh = np.array(to_wh)
    scale_ratio = min(from_wh[0]/to_wh[0], from_wh[1]/to_wh[1])
    pad = (from_wh - scale_ratio*to_wh) // 2
    vtx = (vtx - pad) / scale_ratio
    return vtx.astype(np.int32)
```

Snippet 26. The implementation of the rescale_vertex() function

```
def add_overlays(frame, preds, pred_wh, labels, palette):
    tops, bots, scores, classes = preds
    if not tops:
        return frame
    frame_wh = frame.shape[:-1][::-1]
    vtcs = np.concatenate([tops, bots], axis=0)
    vtcs = rescale_vertex(vtcs, pred_wh, frame_wh)
```

```
    tops, bots = np.split(vtcs, 2)
    b_thick = np.int(np.sum(frame_wh) // 1000)
    t_thick = (b_thick//3)+1
    t_scale = 8e-4*np.min(frame_wh)
    font_face = cv2.FONT_HERSHEY_SIMPLEX
    for top, bot, cls in zip(tops, bots, classes):
        colour = palette[cls]
        top = tuple(top)
        bot = tuple(bot)
        frame = cv2.rectangle(frame, top, bot, colour, b_thick)
        txt = '{}'.format(labels[cls])
        t_size = cv2.getTextSize(txt, font_face, t_scale, t_thick)[0]
        t_box_bot = top
        t_box_top = (t_box_bot[0] + t_size[0] + b_thick*4, t_box_bot[1] -
t_size[1] - b_thick*6)
        t_orig = top[0]+b_thick*2, top[1]-b_thick*4
        if t_box_top[1] < 0:
            t_box_top = top
            t_box_bot = (t_box_top[0] + t_size[0] + b_thick*4,
t_box_top[1] + t_size[1] + b_thick*6)
            t_orig = top[0] + b_thick*2, top[1] + t_size[1] + b_thick*2
        frame = cv2.rectangle(frame, t_box_top, t_box_bot, colour, -1)
        frame = cv2.putText(frame, txt, t_orig, font_face, t_scale, (255,
255, 255), t_thick)
    return frame
```

Snippet 27. The implementation of the add_overlays() function

```
def visualise(orig_imlist, pred_list, pred_ranges, labels, palette):
    '''
    Visualise predictions on original images.
    '''
    imlist = []
    for im, preds in zip(orig_imlist, pred_list):
        imlist.append(add_overlays(im, preds, pred_ranges, labels, pal-
ette))
    return imlist
```

Snippet 28. The implementation of the visualise() function

Lastly, imwrite() – calls function imsave() to write all the output images to directory output/; and display_mess() – display a text message on the top left corner of set images.

```
def imwrite(ims):
    p = Path('.').absolute()
    save_dir = p.joinpath('outputs').as_posix()
```

```
    if not os.path.exists(save_dir):
        os.makedirs(save_dir)
    [cv2.imwrite(save_dir + '/{}.jpg'.format(i), im) for i, im in
zip(range(len(ims)), ims)]
    print('Images have been saved to {}'.format(save_dir))
def display_mess(frame, mess):
    cv2.putText(frame, mess, (2, 20), cv2.FONT_HERSHEY_SIMPLEX, .65,
(255, 255, 255), 2, 2)
    return frame
```

Snippet 29. Implementation of functions imwrite() and, display_mess()

3.1.7 Example programmes

With all these components built up, it can be used as in this example detect_images.py which performs detections on all images in directory images/. An Imager instance imer takes care of opening images and pre-processing them. Then a Yolow instance gets fed that input for predicting. The returned detections get passed back to imer to visualise on the original images and write to output/.

```
yl = Yolow(model_args['type'],
           model_args['model'],
           model_args['anchors'],
           model_args['num_classes'],
           input_sz)
input_list = imer.preproces()
start = time.time()
pred_list = yl.predict(input_list)
print('prediction takes {}s'.format(time.time() - start))
ims = imer.visualise_preds(pred_list)
imer.imsave(ims)
```

Snippet 30. The sample programme detect_images.py

Another example programme is live.py in which visual frames from camera input is used for prediction and get displayed directly. The frame rate is also calculated and get printed every 3 seconds as well for performance evaluation.

```
def main(args):
    imer = Imager(transform_sizes=(args['width'], args['height']),
                  labels=args['labels'])
    yl = Yolow(args['type'],
               args['model'],
               args['anchors'],
```

```
            args['num_classes'],
            (args['width'], args['height']))
    cam = cv2.VideoCapture(0)
    fps_display_interval = 3
    frame_rate = 0
    frame_count = 0
    start_time = time()
    while True:
        _, frame = cam.read()
        imer.imset(frame)
        input_list = imer.preproces()
        pred_list = yl.predict(input_list)
        imer.visualise_preds(pred_list)
        duration = time() - start_time
        if duration >= fps_display_interval:
            frame_rate = round(frame_count/duration)
            start_time = time()
            frame_count = 0
        fps_txt = '{} fps'.format(frame_rate)
        frame = imer.display_mess(fps_txt)
        cv2.imshow('YOLOv3 Live', frame)
        frame_count += 1
        if cv2.waitKey(1) & 0xFF==ord('q'):
            break
    cam.release()
    cv2.destroyAllWindows()
```

Snippet 31. The sample programme live.py

3.2 OpenVINO

3.2.1 Convert frozen model with Model Optimiser for TensorFlow

In order to run with the NCS, the frozen .pb model has to be converted to OpenVINO IR using the TF Model Optimiser module. The conversion is done by running the following command:

```
…/openvino/deployment_tools/model_optimizer/mo_tf.py --input_model
data/pb/frozen_model.pb --tensorflow_use_custom_operations_config
data/config/frozen_model_ir_config.json –input_shape=[1,416,416,3] –scale
255 –data_type FP16 –output_dir data/ir
```

Snippet 32. Model optimiser run command to convert .pb model to OpenVINO IR

The TF model optimiser is a Python programme that takes as input a .pb TF model along with a variety of information to tell about how the model should be converted. In the

command of snippet 12, besides the path to the input model, a config file also needs to be provided as YOLOv3 is not yet an officially supported architecture. This config file is in json format and it specifies the output graph nodes as well as other model parameters to build into the IR output and the path for the output, which is data/ir. The content of the yolov3_config.json for the full YOLOv3 is as follow:

```
[
  {
    “id”: “TFYOLOV3”,
    “match_kind”: “general”,
    “custom_attributes”: {
      “classes”: 80,
      “54ords”: 4,
      “num”: 9,
      “mask”: [0, 1, 2],
      “entry_points”: [“detections/yolov3/Reshape”, “detec-
tions/yolov3/Reshape_4”, “detections/yolov3/Reshape_8”]
    }
  }
]
```

Snippet 33. Content of Yolov3_config.json for the TF model optimiser

3.2.2 Class YolowNCS

The YolowNCS class is a programming interface for writing programmes that uses the YOLOw models on the NCS using OpenVINO. It only requires creating the object instance and call for the predictions on the passing input, the class abstracts away all the handlings of initializing, loading and predicting under the hood. The details are as follow:

```
class YolowNCS(object):

    def __init__(self, model_name_path, anchor_file, num_classes, in-
put_size, num_requests=2):
        self.model = model_name_path + '.xml'
        self.weights = model_name_path + '.bin'
        if not os.path.exists(self.model) or not os.path.ex-
ists(self.weights):
            raise ValueError('model files {} does not exist.'.for-
mat(model_name_path))
        if not os.path.exists(anchor_file):
            raise ValueError('anchor file {} does not exist.'.format(an-
chor_file))
```

```
        self._ANCHORS = load_anchors(anchor_file)
        self.num_classes = num_classes
        if not input_size:
            input_size = 416
        if type(input_size) is int:
            input_size = input_size, input_size
        self.input_size = input_size
        self.plugin=IEPlugin(device='MYRIAD')
        log.info('Loading network files:\n\t{}\n\t{}'.format(self.model,
self.weights))
        self.net=IENetwork(model=self.model, weights=self.weights)
        log.info('Preparing inputs')
        self.input_blob=next(iter(self.net.inputs))
        self.net.batch_size=1
        log.info('Loading model to the plugin')
        self.current_request_id = 0
        self.next_request_id = 1
        self.num_requests = num_requests
        self.exec_net=self.plugin.load(network=self.net, num_re-
quests=self.num_requests)
```

Snippet 34. The constructor of the YolowNCS class

An instance of YolowNCS takes in the path to the IR model, path to the anchor file, the number of classes, the input size, and the number of requests to process asynchronously. Basically, the Inference Engine (IE) handles loading the model and perform inference using the NCS. In the initialisation of an YolowNCS object instance, it first creates an IE plugin of type *Myriad* as this is the name of the chip in the NCS, second it creates an IE network which represents the IR model from the provided path, third it retrieves the input node in the model graph, and finally it load the IE network on the IE plugin and gets hold of the network object to call inference on.

```
    def predict(self, input_list, confidence_theshold=.6, iou_thes-
hould=.5, async_mode=False):
        batch_predictions = []
        get_from = 0
        input_dict = {self.input_blob: input_list}
        request_handle = self.exec_net.requests[self.current_request_id]
        if async_mode:
            next_request_id = self.current_request_id + 1
            if next_request_id == self.num_requests:
                next_request_id = 0
        else:
            next_request_id = self.current_request_id
```

```
        self.exec_net.start_async(request_id=next_request_id,
                                  inputs=input_dict)
        if async_mode:
            self.current_request_id = next_request_id
        request_handle.wait()
        pred_dict = request_handle.outputs
        for preds in pred_dict.values():
            preds = np.transpose(preds, [0, 2, 3, 1])
            get_to = get_from + 3
            batch_predictions.append(region_np(preds, self._AN-
CHORS[get_from:get_to], self.input_size, self.num_classes + 5))
            get_from = get_to
        batch_predictions = np.concatenate(batch_predictions, axis=1)
        return predict(batch_predictions, confidence_theshold, iou_thes-
hould)
```

Snippet 35. The implementation of the predict() method

The class has only one method *predict()* which takes in the input image and requests the accelerator to predict it in order of an assigned request ID. It can be run in two modes: synchronous and asynchronous. In the prior mode, the whole programme waits for the hardware to return the output. Whilst in the latter mode, it still allows the system to have control during the wait-for-processing time to run other jobs i.e. read in new input and pushes it on queue. In this way, the hardware will immediately predict the next input right after the other is finished, which increases throughput and results in better overall performance in most cases. Due to the limited support of OpenVINO on TF operations, the output is returned from the last convolution layer of *_detection_block()* and therefore, it has to be process by a *Numpy* implementation of *_region()*, *the _region_np()*. To maintain the consistency, the predictions are still in the same structure and can be process by the same function *predict.predict()* for detections.

3.2.3 Example programmes

YolowNCS can be used in a similar way to Yolow with minimal difference of a few specific parameters which have already been default to optimal values.

```
    yl = YolowNCS(model_args['model_name_path'],
                      model_args['anchors'],
                      model_args['num_classes'],
                      input_sz,
                      num_requests=2)
```

```
    input_ims = imread_from_path(args.image_path)
    ims = []
    start = time.time()
    for im in input_ims:
        imer.imset(im)
        input_list = imer.ncs_preprocess()
        pred_list = yl.predict(input_list)
        im = imer.visualise_preds(pred_list)[0]

        ims.append(im)

    print('prediction takes {}s'.format(time.time() - start))
```

Snippet 36. Usage example of YolowNCS in demo programme detect_images.py

As an aside, since the IR model does not take the same input as the TF .pb model, pre-processing is done slightly differently, and Imager has specified method for it as detailed below.

```
    def ncs_preprocess(self):
        ims = improcess(self.ims, self.transform_sizes, to_rgb=False,
normalise=False) # ims are normalised by the ncs.
        return np.transpose(ims, [0, 3, 1, 2])
```

Snippet 37. implementation of ncs_preprocess()

Method ncs_preprocess() actually still make call to the same function improcess() but with argument to_rgb set to False to keep the values of the channel dimension in the input in BGR format instead of RGB. the input dimension is also permuted to [number_of_input, channels, width, height].

Furthermore, it is also possible to run object detection on the NCS using input stream from camera. The following example programme uses multi-processing for independently running the camera process and NCS inferencing process, in which multi-threading is used to control multi NCS devices at the same time. Additionally, the programme takes as arguments a configuration file specifying requiring information of the model to be loaded, the number of NCS devices to use, and the request capacity for each of those devices.

```
def live_job(model_args, input_buffer, output_buffer, async_mode,
quit_event):
```

```
    imer = Imager(transform_sizes=(model_args['width'],
model_args['height']), labels=model_args['labels'])
    camera = PiCamera()
    camera.resolution = (720, 480)
    camera.framerate = 30
    rawCapture = PiRGBArray(camera)
    fps_display_interval = 1
    frame_rate = 0
    frame_count = 0
    processing_frame_rate = 0
    processing_frame_count = 0
    start_time = time()
    pred_list = None
    for frame in camera.capture_continuous(rawCapture, format="bgr",
use_video_port=True):
        frame = frame.array
        imer.imset(frame)
        if input_buffer.full():
            input_buffer.get()
        input_list = imer.ncs_preprocess()
        input_buffer.put(input_list)
        if not output_buffer.empty():
            pred_list = output_buffer.get()
            processing_frame_count += 1
        if pred_list is not None:
            imer.visualise_preds(pred_list)
        duration = time() - start_time
        if duration > fps_display_interval:
            frame_rate = round(frame_count/duration)
            frame_count = 0
            processing_frame_rate = round(processing_frame_count/dura-
tion)
            processing_frame_count = 0
            start_time = time()
        mode = "async" if async_mode.value else "sync"
        status = '{} fps (detection: {} fps) - mode: {}'.for-
mat(frame_rate, processing_frame_rate, mode)
        frame = imer.display_mess(status)
        cv2.imshow('NCS YOLOv3 Live', frame)
        frame_count += 1
        rawCapture.truncate(0)
        key = cv2.waitKey(1)
        if key & 0xFF==ord('q'):
            quit_event.set()
            break
        if key & 0xFF==ord('m'):
```

```
            async_mode.value = not async_mode.value
    cv2.destroyAllWindows()
```

Snippet 38. The details of live_job() function run by a separate process

```
def ncs_worker_thread(yl_ncs, input_buffer, output_buffer, async_mode,
quit_event):
    while not quit_event.is_set():
        if not input_buffer.empty():
            input_list = input_buffer.get()
            pred_list = yl_ncs.predict(input_list,
async_mode=async_mode.value)
            output_buffer.put(pred_list)
```

Snippet 39. The details of ncs_worker_thread() function run by multi-threading

```
def infer_job(model_args, sticks, requests, input_buffer, output_buffer,
async_mode, quit_event):
    threads = []
    for _ in range(sticks):
        th = Thread(target=ncs_worker_thread,
                    args=(YolowNCS(model_args['model_name_path'],
                                   model_args['anchors'],
                                   model_args['num_classes'],
                                   (model_args['width'],
model_args['height']),  requests),
                    input_buffer,
                    output_buffer,
                    async_mode,
                    quit_event))
        th.start()
        threads.append(th)
    for th in threads:
        th.join()
```

Snippet 40. The details of infer_job() function run by a separate process

4 TESTING RESULTS

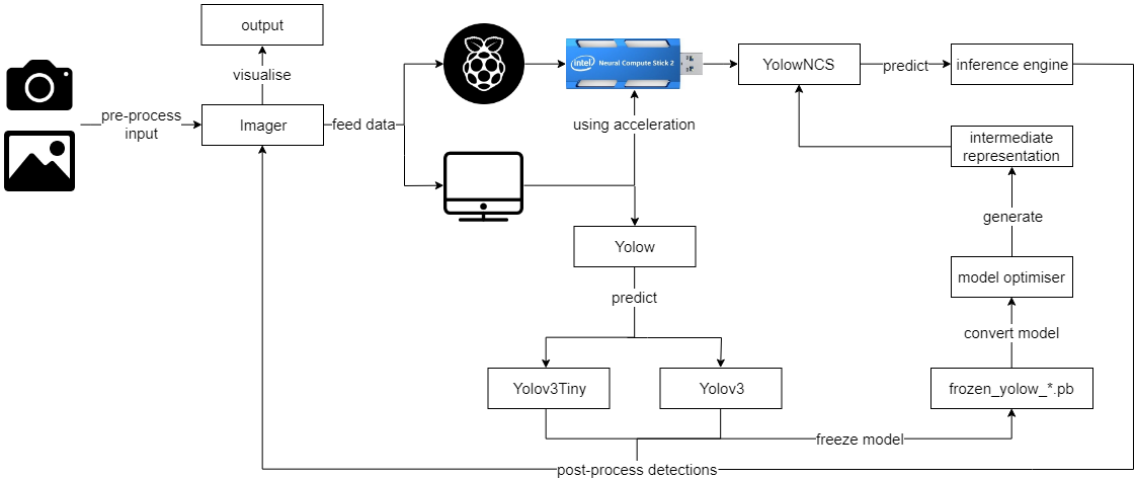

Figure 22. YOLOw workflow

As in figure 21, YOLOw workflow can be divided into two following sections:

a) For using on a PC
   1. Instantiate an Imager object to preprocess the input image.
   2. Feed the input data to a Yolow object that load the target YOLOv3 model for inference.
   3. Forward output detections to Imager for visualisation on the original image.

At this end, the Yolow object will have conveniently helped you create the frozen Protobuf model of the used Yolov3 model.

b) For use with the NCS2, the frozen model is required to generate the IR model graph. The workflow is as follows:
   4. Use the OpenVINO Model Optimizer with the frozen model to generate the IR model graph.
   5. Instantiate an Imager object to preprocess the input image.
   6. Feed data to a YolowNCS object that loads the target IR model graph for inference.
   7. Forward output to Imager to display.

YOLOw supports both the YOLOv3 and tiny YOLOv3 architectures of not only the default models trained on the COCO dataset for 80 object classes, but any models trained

on any custom datasets, on either architecture, that might output any number of classes. This is thanks to the WeightLoader module that was correctly implemented the pattern of the Darknet weights and can read its binaries with empirically minimal loss. YOLOw and YOLOw-NCS both have demo programmes to run detection for an input folder/directory of images and live detection on camera. In addition to those, YOLOw-NCS has a demo implemented with multi-processing and multi-threading to make use of running multiple NCS2 for performance reinforcement by effectively make use of more resources on some capable machine. Last but not least, all the test results of YOLOw-NCS are recorded with the asynchronous inference mode turned on to further improve the framerate. The test subjects are the HP OMEN 15 model 2018 laptop PC and the latest RPI 4. The laptop PC is equipped with following hardware: Intel Core i7 8750H CPU, 16GB of DDR4 RAM, NVIDIA GTX 1070Max-Q GPU. The RPI 4 hardware has the specifications of Broadcom BCM2711, Quad core Cortex-A72 (ARM v8) 64-bit SoC @ 1.5GHz, 4GB of LPDDR4-2400 SDRAM.

Hardware	**Frame Rate (FPS)**
Laptop PC	16 - 20
Laptop PC + 1 NCS2	2 - 3
Laptop PC + 2 NCS2	3 - 5
RPI4 + 1 NCS2	2 - 3
RPI4 + 2 NCS2	2 - 3

Table 1. The performance result summary of YOLOv3

Hardware	**Frame Rate (FPS)**
Laptop PC	30
Laptop PC + 1 NCS2	16

Laptop PC + 2 NCS2	27
RPI4 + 1 NCS2	10 - 13
RPI4 + 1 NCS2	6

Table 2 . The performance result summary of YOLOv3-tiny

On the laptop PC, running YOLOw with the COCO YOLOv3 model achieved about 16 to 20 FPS depending on the scenes. For COCO YOLOv3-tiny, it achieved a consistency performance of 30 FPS with some trade-off to detection capability and accuracy. With the NCS2, YOLOw-NCS was put into use. When pairing just one NCS2, the performance was 2 FPS on COCO YOLOv3 and 15 FPS on COCO YOLOv3-tiny. When paring two NCS2 devices with PC, it achieved around 3-5 FPS on average on COCO YOLOv3 and was still able to reach roughly 28 FPS on YOLOv3-tiny, which is comparable with the PC-class hardware. Although the NCS2 processes only values of 16-bit floating-point precision while the other is 32-bit floating-point values. It can be understood that the performance loss due to the quantisation is optimally small. These statistics in table 1 show that each NCS2 device is capable of pushing out about 2 FPS on average when there is redundant CPU resource power.

When pairing the NCS2 devices with RPI 4, the performance is acceptedly applicable depending on which the application it is applied to. With one NCS2 and COCO YOLOv3 model, the performance reached 2 to 3 FPS, and for YOLOv3-tiny, it was about 10 to 13 FPS. However, with two NCS2 devices paired the RPI 4, a considerable amount of bottleneck is introduced to the processing due to its limited CPU, although the accuracy remained. For COCO YOLOv3, the framerate was at 2 FPS. And for COCO YOLOv3-tiny, it throttled at 6 FPS.

Simply glancing at the the system monitor, the problem seems to be due to either the CPU of the RPI 4, or the current firmware of the Raspbian OS has not yet been optimised for the hardware of the new machine, or the programme code needs to be refactored. It is also possible that more than one causes could contribute to this problem as well. In general, the causing problems prevents to push out performance any further with multiprocessing and multithreading method over running just one NCS2. It can be

observed in figure 35 where YOLOv3-tiny was run with 2 NCS2, and figure 36 where that same model was run with 1 NCS2. Almost all four cores of the CPU were busy at runtime when multiple NCS2 were running while there was still a lot of CPU performance available when just one NCS was running.

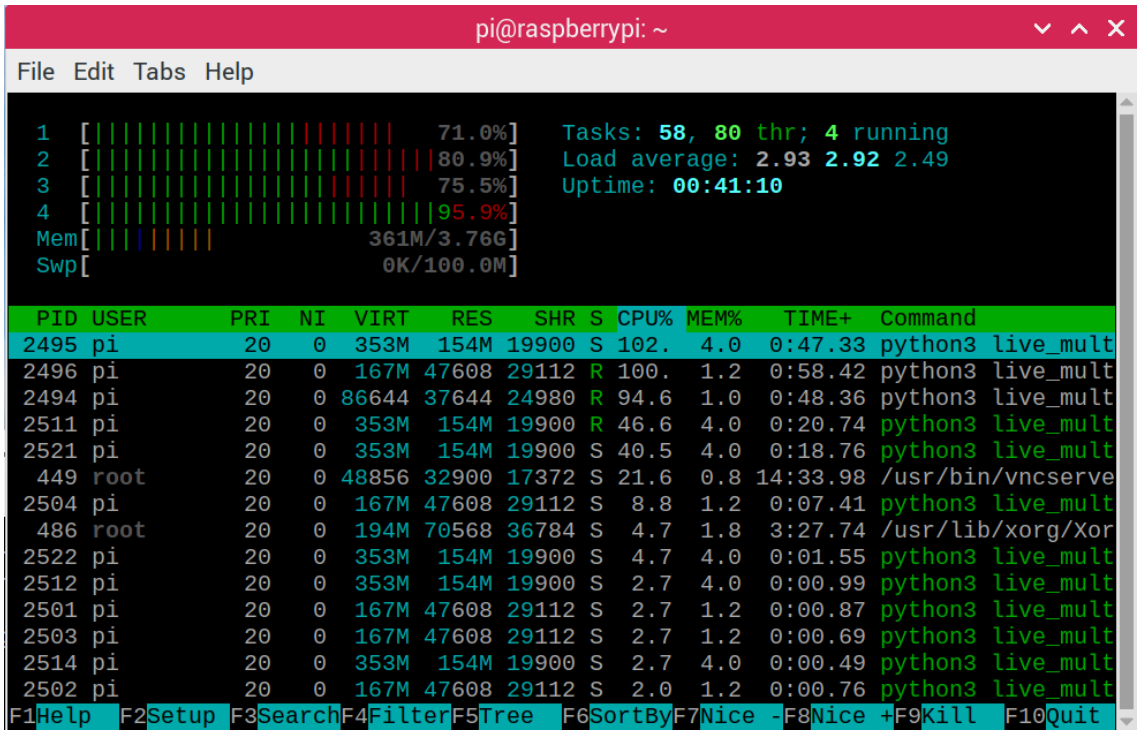

Figure 23. RPI 4 CPU bottlenecked from running two NCS2

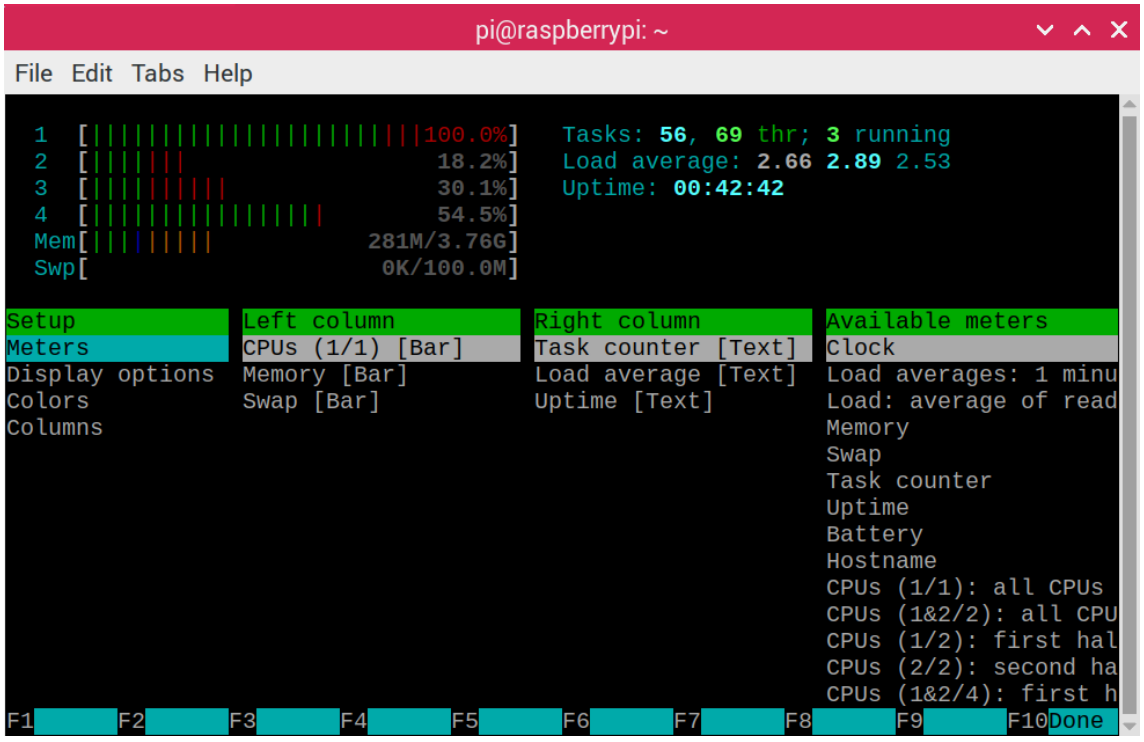

Figure 24. RPI 4 did not bottleneck from running one NCS2

Snips of the output results are below:

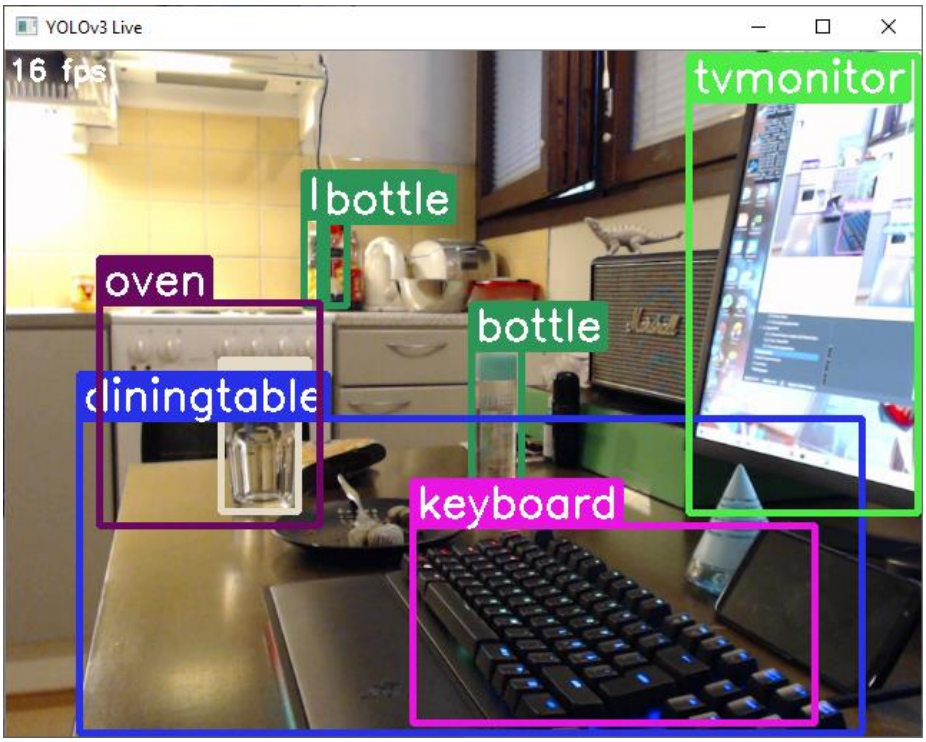

Figure 25. YOLOw runs COCO YOLOv3 on PC hardware

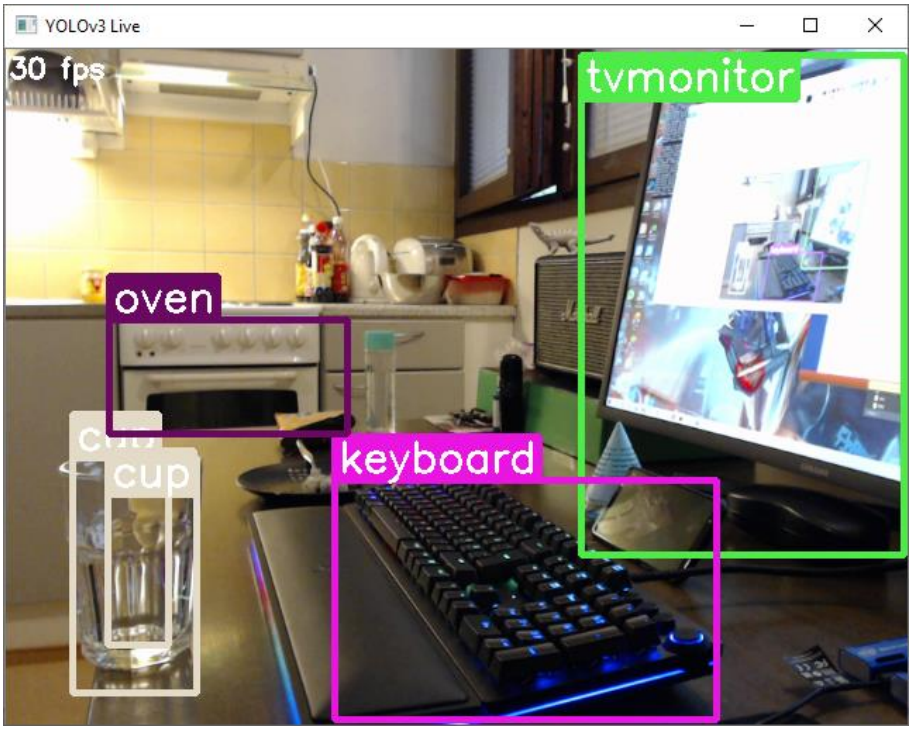

Figure 26. YOLOw run COCO YOLOv3-tiny on PC hardware

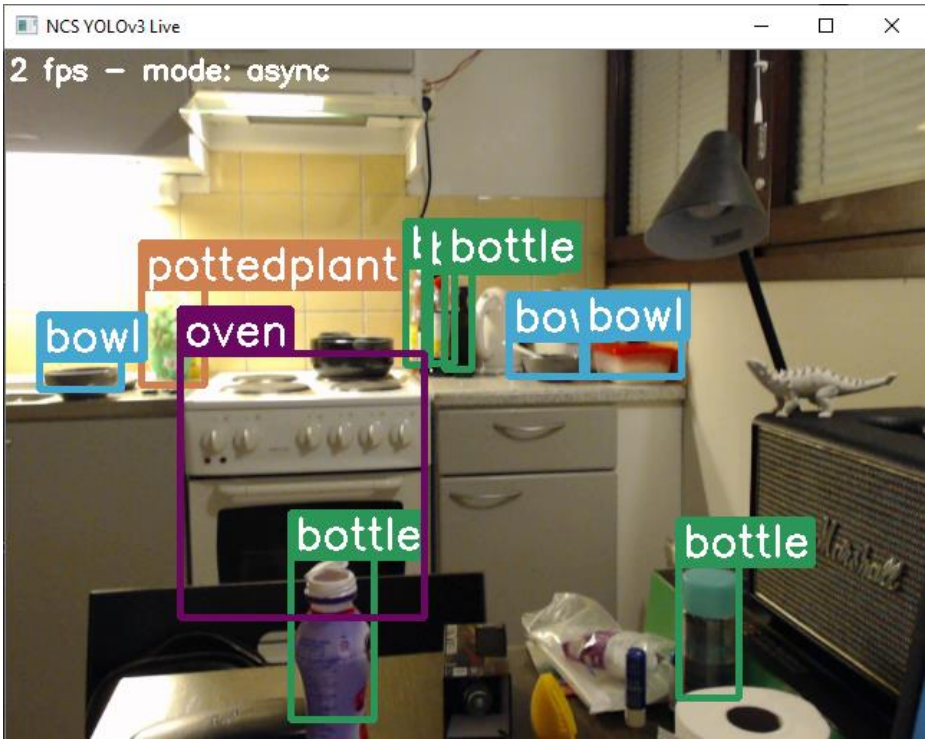

Figure 27. YOLOw-NCS runs COCO YOLOv3 on PC with one NCS

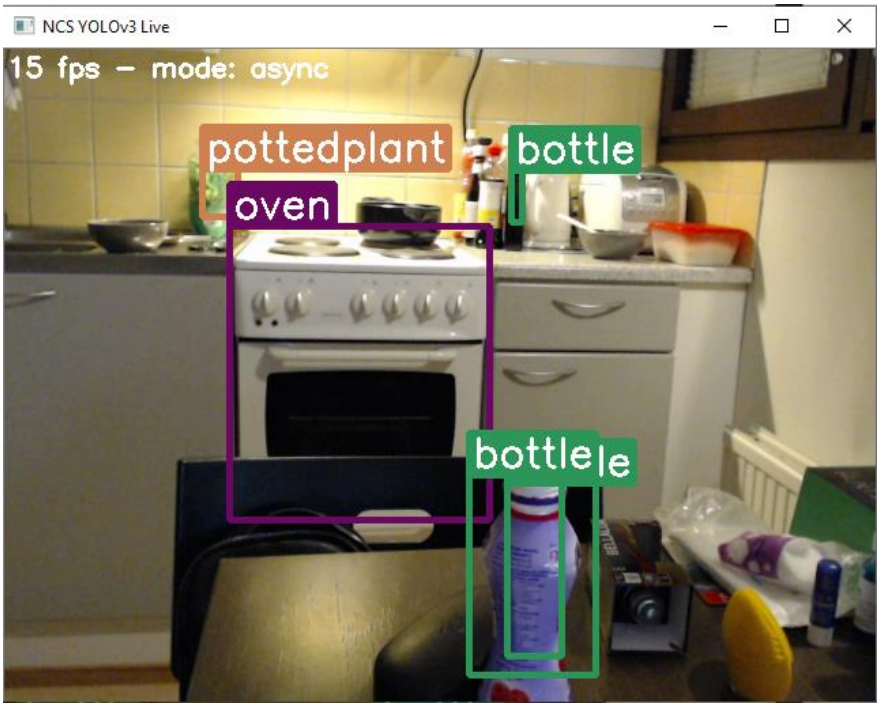

Figure 28. YOLOw-NCS runs COCO YOLOv3-tiny on PC with one NCS2

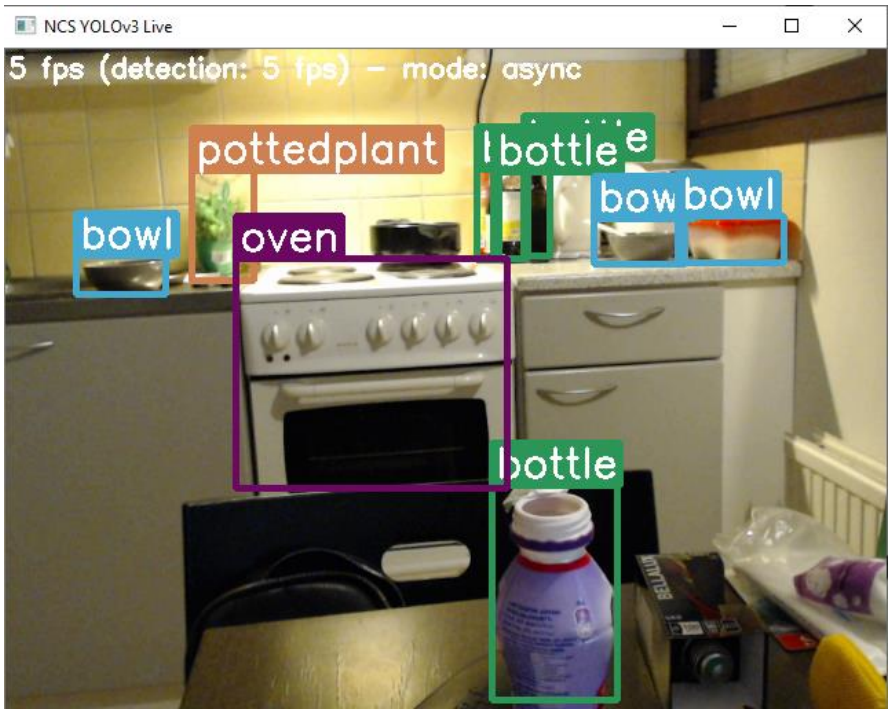

Figure 29. YOLOw-NCS runs COCO YOLOv3 on PC with two NCS2 (1)

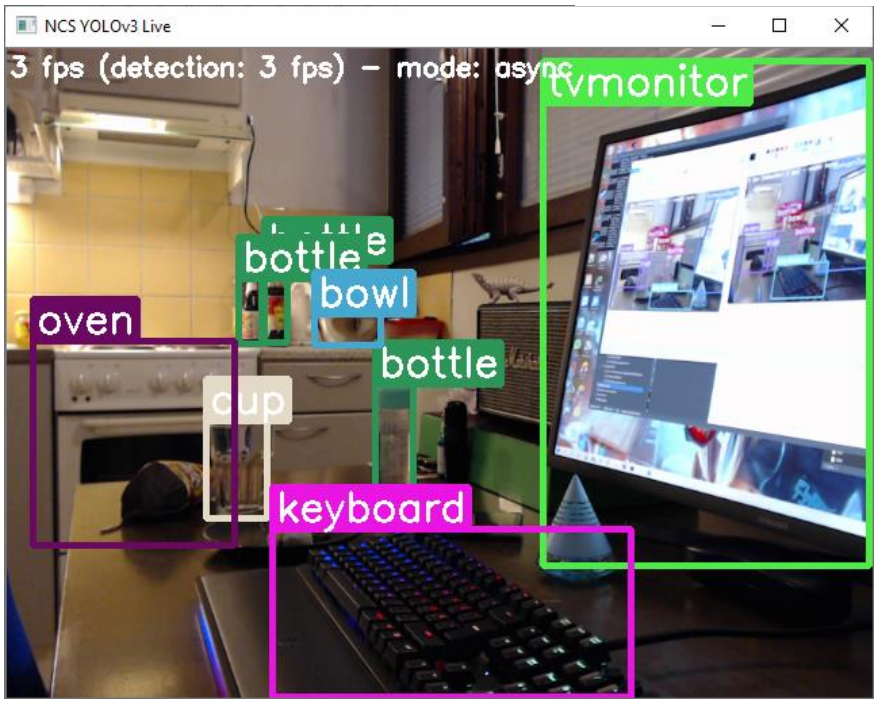

Figure 30. YOLOw-NCS runs COCO YOLOv3 on PC with two NCS2 (2)

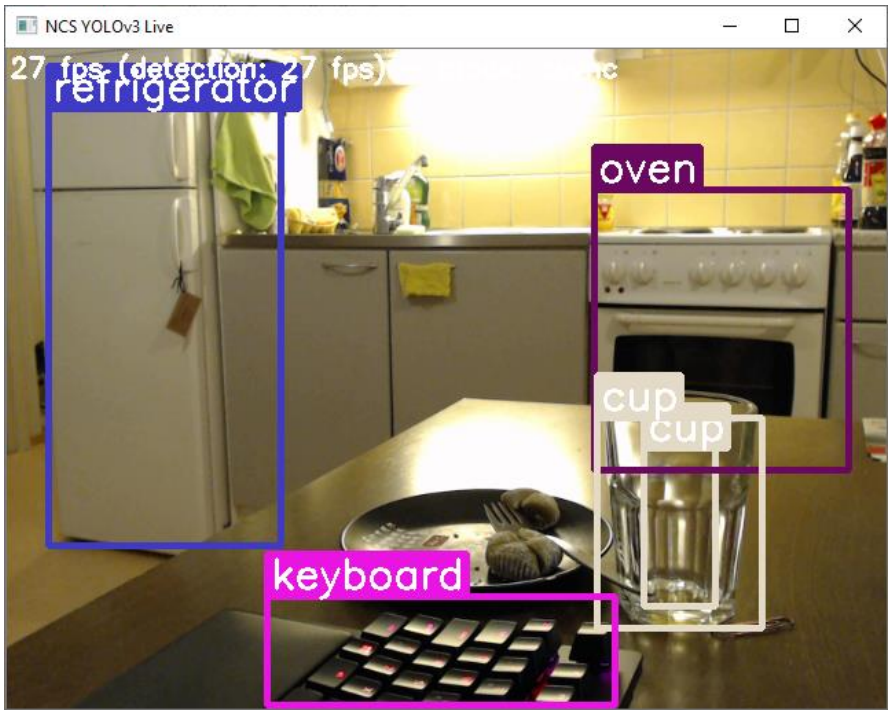

Figure 31. YOLOw-NCS runs COCO YOLOv3-tiny on PC with two NCS2

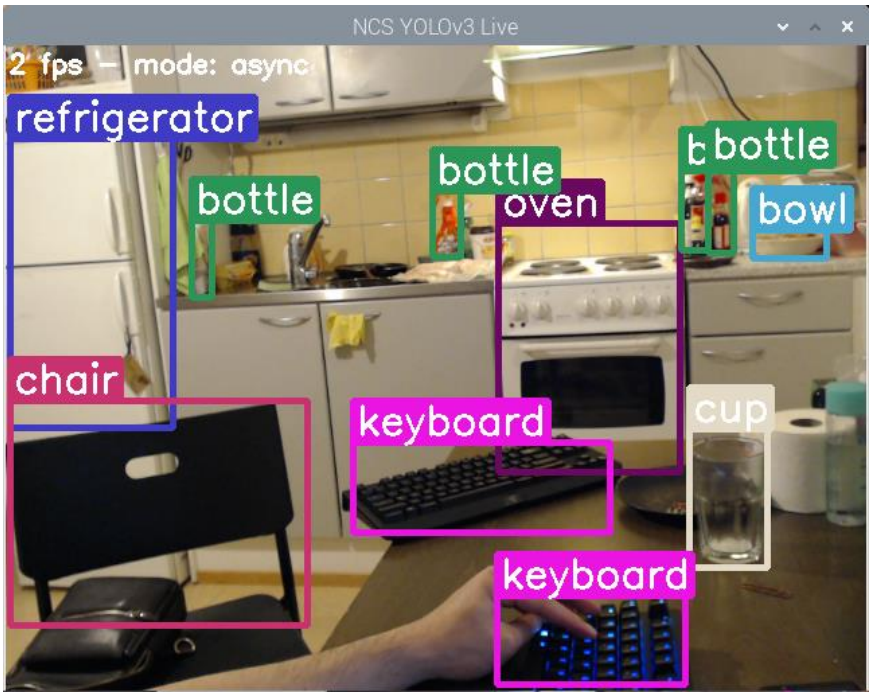

Figure 32. YOLOw-NCS runs COCO YOLOv3 on RPI 4 with one NCS2

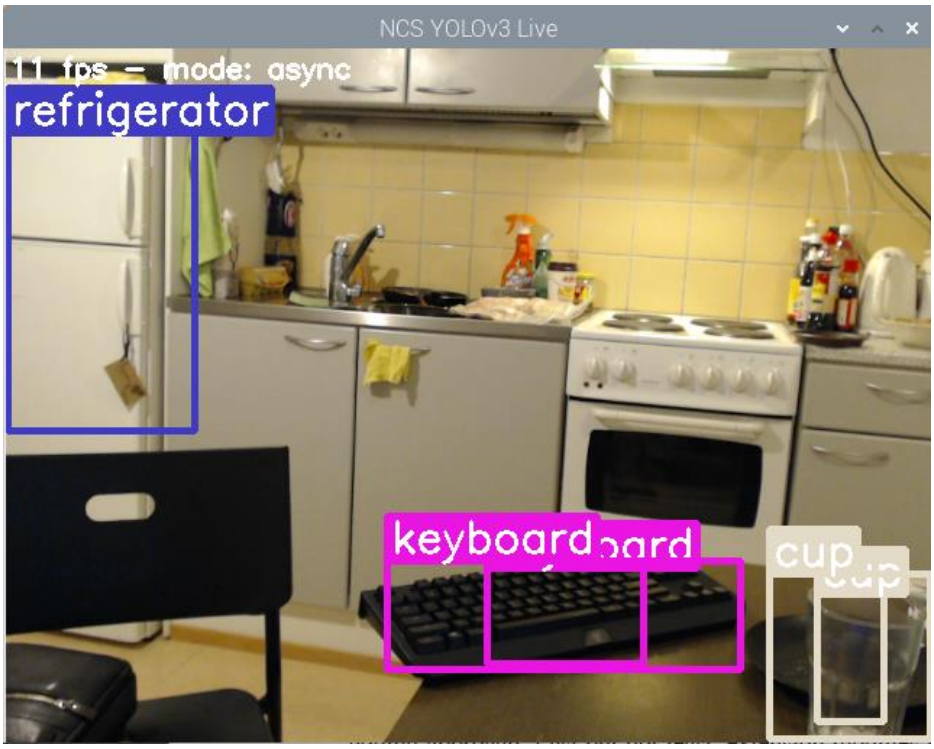

Figure 33. YOLOw-NCS runs COCO YOLOv3-tiny on RPI 4 with one NCS2 (1)

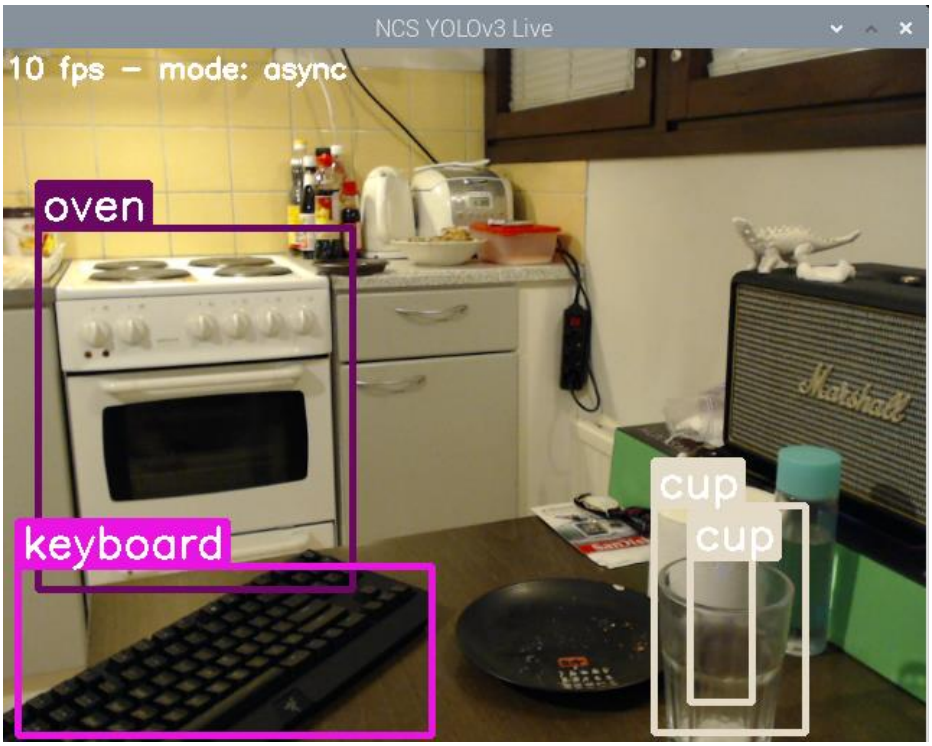

Figure 34. YOLOw-NCS runs COCO YOLOv3-tiny on RPI 4 with one NCS2 (2)

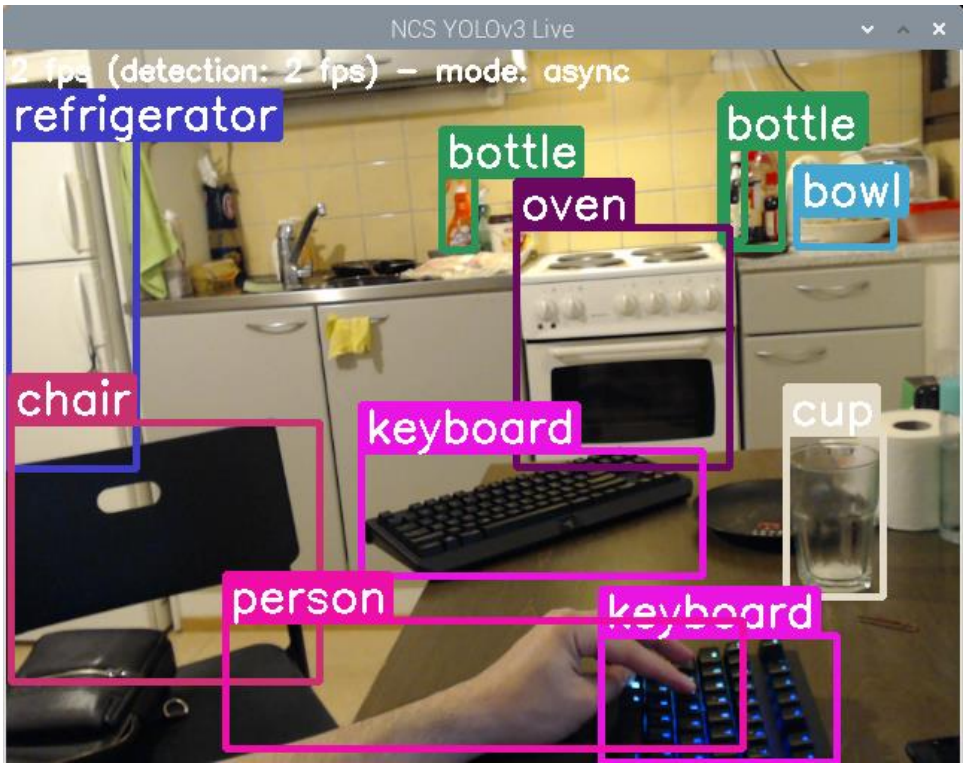

Figure 35. YOLOw-NCS runs COCO YOLOv3 on RPI 4 with two NCS2

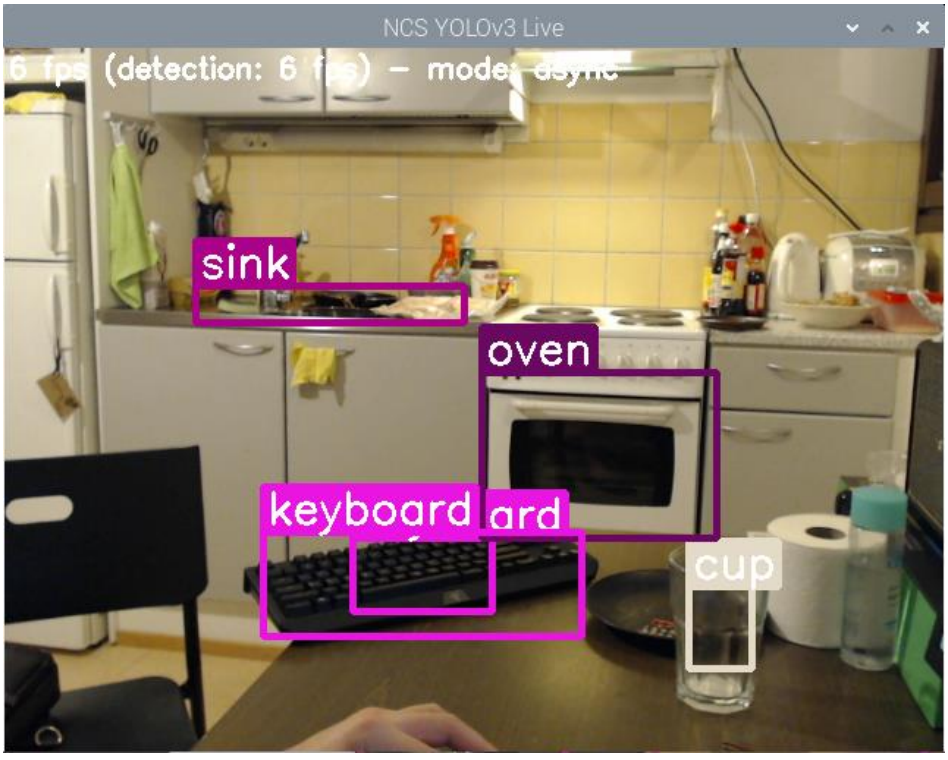

Figure 36. YOLOw-NCS runs COCO YOLOv3-tiny on RPI 4 with two NCS2

5 FURTHER IMPROVEMENTS

For the future versions of YOLOw, it is desirable to expand the feature set with a training module, which will allow training YOLOv3(-tiny) models, including the custom ones, to no longer need to rely on the Darknet framework prior to converting to TensorFlow. Furthermore, it could be more valuable to improve the kernel code of YOLOw so that it could achieve polymorphism in model graph building from just a config file, as it is the case for Darknet. This will allow easy constructing and loading any of the passed architectures of YOLO and make use of the capability of YOLOw as well as the TensorFlow library. Additionally, it will also help achieving custom network design in a no coding approach. Last but not least, extension modules in TensorFlow Lite can be integrated to enable the ability of YOLOw to deploy to other embedded systems such as the Nvidia Jetsons and other mobile devices, this will empower this project even further for many applications. YOLOw also has large potentials to be adopted into solving numerous industrial problems as well as in creating profitable commercial products. To mention a few possible applications that this project can be applied to are sentiment analysis and for the healthcare area, object tracking for security, self-driving systems for autonomous vehicles, human proposer for pose estimator, and so on. In summary, the current version of YOLOw features model conversion from Darknet framework to TensorFlow as a *ProtoBuf* frozen file, model loading from Darknet weights files as well as frozen model files and performing detection in TensorFlow and on the Intel Neural Compute Stick using OpenVINO Inference Engine. The project also includes detection programmes running on a folder of images input and live camera.

6 CONCLUSION

This project builds the YOLOw modules that allows any YOLOv3(-tiny) models in Darknet framework to be converted to TensorFlow library and easily deploy and run with the NCS devices. This is achieved by reconstructing these two network architectures using TensorFlow and building a binary file loader that correctly read the Darknet model weights files and assign the corresponding values to the TensorFlow network operations. The module is highly integratable to many industrial applications and tools for other research to make use of. However, the main module of the project was built to be deployed to edge devices and used with the Intel's Neural Compute Sick accelerating devices, that is the YOLOw-NCS. This is based on the Intel's OpenVINO toolkit framework that allow for converting and loading YOLOv3(-tiny) models on these accelerators to run. This Python module mainly aims to facilitate the use of embedded system applications, as well as PC applications.

7 BIBLIOGRAPHY

[1] G. Balinggan, "Difference Between Artificial Intelligence and Human Intelligence," 1 March 2019. [Online]. Available: http://www.differencebetween.net/science/difference-between-artificial-intelligence-and-human-intelligence/.

[2] "The Neural Compute Stick 2," Intel®, [Online]. Available: https://software.intel.com/en-us/neural-compute-stick. [Accessed 2019].

[3] "The Jetson Family," Nvidia, [Online]. Available: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/. [Accessed 2019].

[4] "Coral," Google, [Online]. Available: https://coral.withgoogle.com/. [Accessed 2019].

[5] "The Essence of Artificial Neural Networks," [Online]. Available: https://medium.com/@ivanliljeqvist/the-essence-of-artificial-neural-networks-5de300c995d6. [Accessed 2019 5 18].

[6] "Perception," Wikipedia, [Online]. Available: https://en.wikipedia.org/wiki/Perceptron.

[7] "Artificial Neural Networks," [Online]. Available: https://www.computerworld.com/article/2591759/artificial-neural-networks.html. [Accessed 2019 5 18].

[8] "Convolutional Neural Networks (CNNs / ConvNets)," cs231n Stanford University, [Online]. Available: http://cs231n.github.io/convolutional-networks/.

[9] "Batch Normalisation," [Online]. Available: https://en.wikipedia.org/wiki/Batch_normalization.

[10] "Sigmoid Function," [Online]. Available: https://en.wikipedia.org/wiki/Sigmoid_function.

[11] "Hyperbolic Function," [Online]. Available: https://en.wikipedia.org/wiki/Hyperbolic_function.

[12] "Rectifier Linear Unit," [Online]. Available: https://en.wikipedia.org/wiki/Rectifier_(neural_networks).

[13] "Softmax Function," [Online]. Available: https://en.wikipedia.org/wiki/Softmax_function.

[14] "Convolutional Neural Network (CNN)," NVIDIA, [Online]. Available: https://developer.nvidia.com/discover/convolutional-neural-network. [Accessed 19 5 2019].

[15] "Training CNN for Image Classification," [Online]. Available: https://cv-tricks.com/tensorflow-tutorial/training-convolutional-neural-network-for-image-classification/.

[16] A. F. Joseph Redmon, "YOLOv3: An Incremental Improvement," University of Washington, [Online]. Available: https://pjreddie.com/media/files/papers/YOLOv3.pdf.

[17] C. D. I. B. A. Jose Dolza, "3D fully convolutional networks for subcortical segmentation in MRI: A large-scale study," LIVIA Laboratory, Ecole de technologie sup´erieure (ETS), Montreal, QC, Canada, 20 April 2017. [Online]. Available: https://arxiv.org/pdf/1612.03925.pdf. [Accessed 05 August 2019].

[18] Y. B. a. A. C. Ian Goodfellow, "Convolutional Networks," [Online]. Available: http://www.deeplearningbook.org/contents/convnets.html.

[19] "People detection on the Pepper Robot using Convolutional Neural Networks and 3D Blob detection," Intelligent Robotics Lab, Faculty of Science, University of Amsterdam, 2 7 2017. [Online]. Available: https://staff.fnwi.uva.nl/a.visser/research/athome/JonathanGerbscheidBachelorThesis.pdf. [Accessed 9 8 2019].

[20] "UNDERSTANDING RESIDUAL NETWORKS," [Online]. Available: https://towardsdatascience.com/understanding-residual-networks-9add4b664b03.

[21] I. S. G. E. H. Alex Krizhevsky, "ImageNet Classification with Deep Convolutional Neural Networks," University of Toronto, [Online]. Available: https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf.

[22] X. Z. S. R. J. S. Kaiming He, "Deep Residual Learning for Image Recognition," Microsoft Research, [Online]. Available: https://arxiv.org/pdf/1512.03385.pdf.

[23] keitakurita, "An Intuitive Explanation of Why Batch Normalization Really Works," 10 1 2018. [Online]. Available: http://mlexplained.com/2018/01/10/an-intuitive-explanation-of-why-batch-normalization-really-works-normalization-in-deep-learning-part-1/.

[24] "YOLO: Real-Time Object Detection," [Online]. Available: https://pjreddie.com/darknet/yolo/.

[25] "ImageNet - Download the Object Bounding Boxes," Stanford Vision Lab, Stanford University, Princeton University, 2016. [Online]. Available: http://image-net.org/download-bboxes. [Accessed 14 09 2019].

[26] A. Kathuria, "What is new in YOLOv3?," [Online]. Available: https://towardsdatascience.com/yolo-v3-object-detection-53fb7d3bfe6b.

[27] "What is Python? Executive Summary," [Online]. Available: https://www.python.org/doc/essays/blurb/.

[28] "TensorFlow," [Online]. Available: https://www.tensorflow.org/.

[29] "NumPy," [Online]. Available: https://www.numpy.org/.

[30] "OpenCV," [Online]. Available: https://opencv.org/.

[31] N. Smith, "Transitioning from Intel® Movidius™ Neural Compute SDK to Intel® Distribution of OpenVINO™ toolkit," Intel, 11 4 2019. [Online]. Available: https://software.intel.com/en-us/articles/transitioning-from-intel-movidius-neural-compute-sdk-to-openvino-toolkit. [Accessed 10 8 2019].

[32] "Intel's Neural Compute Stick," Intel, [Online]. Available: https://software.intel.com/en-us/movidius-ncs.

[33] "Intel's Neural Compute SDK," Intel, [Online]. Available: https://movidius.github.io/ncsdk/.

[34] "https://software.intel.com/en-us/openvino-toolkit/deep-learning-cv," Intel, [Online]. Available: https://software.intel.com/en-us/openvino-toolkit/deep-learning-cv.

[35] "Raspberry Pi 3 Model B+," [Online]. Available: https://www.raspberrypi.org/products/raspberry-pi-3-model-b-plus/.