Real-Time Data processing and AI for Distributed IoT

Amila Weerasinghe
Rajitha Opanayaka
Rashmi Thilakarathne

Department of Computer Engineering
University of Peradeniya

Final Year Project (courses CO421 & CO425) report submitted as a requirement of the degree of
B.Sc.Eng. in Computer Engineering

April 2021

Supervisors: Dr. Upul Jayasinghe (University of Peradeniya) and Dr.Damayanthi Herath (University of Peradeniya)

We would like to dedicate this thesis to our loving parents and “teachers” . . .
Who never gave up on us
through thick and thin. . .

Declaration

We hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is our own work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgments.

Amila Weerasinghe
Rajitha Opanayaka
Rashmi Thilakarathne
April 2021

Acknowledgements

We would like to express our sincere gratitude to our supervisors Dr.Upul Jayasinghe and Dr.Damayanthi Herath for providing their invaluable guidance, comments and suggestions throughout the course of the project. Especially the patience, motivation, and immense knowledge of Dr.Upul Jayasinghe accompanied by Dr.Damayanthi Herath guided us thorough the journey. We also place on record, our sense of gratitude to one and all, who directly or indirectly, have lent their hand in this venture.

Abstract

Artificial Intelligence has been impacted in a variety of industries, leading the world towards revolutionary applications and services that are primarily driven by high-performance computation and storage facilities in the cloud.This is mainly due to the advantage of having higher computational power, larger storage capacity, and scalability. But with the increase of millions of IoT devices, a huge amount of data is being generated by end devices. To process such data, the distributed end devices have to communicate with the cloud servers making it difficult to generate real-time decisions though it consumes a lot of resources including bandwidth, processing, and storage facilities at the cloud. On the other hand, edge computing architectures which consist of Fog computing and ROOF computing enable a distributed way to process data near the sources of data which leads to facilitate real-time processing. But with the limited resources in the end devices, it is quite challenging to perform complex AI algorithms. Thus by developing AI algorithms that can perform AI processing with limited resources with the advantage of Fog computing and ROOF computing, distributed systems like IoT can harvest to its true potentials including real-time decision making and smart systems that lead to new business models while saving computational resources, bandwidth, and storage facilities at the cloud.

Table of contents

List of figures

List of tables

Nomenclature

Acronyms / Abbreviations

AI	Artificial Intelligence
CNN	Convolutional Neural Network
CPE	Cognitive Processing Elements
CPU	Central Processing Unit
DRL	Deep Reinforcement Learning
GPU	Graphical Processing Unit
IoT	Internet of Things
IOU	Intersection over union
ML	Machine Learning
RAM	Random Access Memory
ROOF	Real-time Onsite Operations Facilitation
YOLO	You Only Look Once

Chapter 1

Introduction

Today IoT has a huge impact upon the large scope of fields. Millions and billions of data generated with the development of the Internet of things [3]. Also the development of Artificial Intelligence has extended the technology capabilities. To achieve the maximum capabilities, IoT is enriched with Artificial intelligence. But most of the existing Artificial algorithms require more computational power, Storage, bandwidth and resources. When it comes to distributed architecture IoT, basically cloud computing provides the infrastructure which fulfills the needs of distributed IoT. In some scenarios real time decision taking from the end devices is a must.

Cloud architecture provides infrastructure at a higher level. Therefore to process data in cloud computing level, data has to be transmitted from end devices to the core cloud servers [4]. Network congestion, latency in the communication causes the data processing and decision taking in real time impossible. So a new paradigm has to be introduced in order to overcome the real time restrictions in Artificial intelligence based distributed IoT. Edge computing allows computation to be done in a closer proximity to end devices where only limited resources are available. They can offload the computation when a certain threshold value exceeds, Where policy technology can be used to make decisions associated with offloading. DRL can be used jointly in making decisions based upon Deep Q-networks. Where Federated Learning allows to train the DRL agents. By using both Federated learning and Collaborative Federated learning techniques computation can be done using local data, Where combination of these techniques leads to real time processing in distributed IoT.

Edge computing allows computation to be done in a closer proximity to end devices where only limited resources are available. They can offload the computation when a certain threshold value exceeds Where policy technology can be used to make decisions associated with offloading. DRL can be used jointly in making decisions based upon

Deep Q-networks, Where Federated Learning allows to train the DRL agents. By using both Federated learning and Collaborative Federated learning techniques computation can be done using local data Where combination of these techniques leads to real time processing in distributed IoT.

1.1 Background

Accomplishing the goal of handling Real time AI and ML processing at the edge devices in distributed IoT is based on various aspects. The taxonomy of Cloud computing, Fog Computing and ROOF Computing is a major concept. When it comes to the background the learning techniques and parameter sharing plays a major role. It is vital to adhere to the limited resources in edge devices and application of policy technology in distributed IoT.

1.1.1 Edge Computing

In 2012 to address the challenges of IoT applications the concept of Fog computing was first introduced by Cisco. Compared to the cloud servers, ROOF [5] and fog [6][7] is much closer to the user. Fog computing enables which are the main features of cloud computing, IaaS(Infrastructure as a Service), PaaS(Platform as a Service) and SaaS(Software as a service), to extend towards the edge devices in more close proximity.

Fog nodes can be described under three types namely:

- Servers which are are geo-distributed and are deployed at very common places
- Networking Devices like gateway routers, switches and set-top boxes
- Cloudlets which are considered as micro clouds

Also there should be a collaboration between these nodes. Collaboration techniques can be categorised into three groups.They are cluster, peer to peer and master-slave. In clustering, Clusters can be formed by the homogeneity of fog nodes [8]. Clusters can be static or dynamic. In static clusters they are difficult to scalable in runtime and dynamic formation of clusters have to considered in terms of network overhead.Peer to peer collaboration can be in both hierarchical and flat order based on nearness. In Master-Slave approach basically master Fog node have the main control i.e it controls functionalities, processing load, resource management, data flow, etc. of underlying slave nodes However, in real-time data processing due to this kind master and the slave Fog nodes require high bandwidth to communicate with each other. Even though resources are limited compared to the cloud by reducing the transmission delays and using resources efficiently real time processing can be achieved. Offloading and caching techniques can be used to increase the real time processing capabilities at the edge.

1.1.2 Deep Reinforcement Learning

Deep Reinforcement Learning(DRL) [9][10] is used to jointly manage the communication and computation resources. Capabilities of DRL can even be used to train Deep Q-Learning algorithms [11] and use them as the machine learning model at the edge [3]. Also Federated Learning can be used as a framework for training DRL agents. In this technique, DRL can be used with basically two scenarios. They are with caching [12] and with offloading. In caching DRL, popular content is cached in the intermediate servers or routers. Caching is done so that the most accessed or popular content can be accessed easily. To determine what is to be cached exactly the approach is to define the popularity of files. Basically, the popularity is defined by the probability distribution of the number of requests from users. Caching can be modeled as a Markov Decision Process and then use Deep Reinforcement Learning to Solve it is a compatible approach. In offloading DRL the communication is modelled as a finite state discrete Markov chain [4]. Then computational tasks are formed as a bernoulli distribution. Those tasks are queued as served in FIFO(First In First Out) structure. The problem formulation is done and handled via a stationary control policy.

1.1.3 Computation offloading

Offloading techniques can be used to improve the performance of the limited resources devices and also to address the real time constraints.However the performance depends on the network bandwidth and amount of data exchange between the limited resource node and the resourceful device. In computation offloading heavy computation is transferred to the resourceful servers and received the result from the servers. This involves with decision making where a policy can be used.

1.1.4 Policy technology

In most of the works the distributed IoT is served by policy technology in various aspects. The reason for having the policy technology is that the end devices have different capabilities. In top down approaches, which the AI model is trained in the cloud using high resources and then the AI algorithm is deployed to the edge. Few variations of AI models are trained as a general. Out of these few the most suitable AI model should be deployed into the suitable device depending on the different capabilities. For the purpose at the edge the policy technology is used to apply the most appropriate model. In edge the AI algorithms have to be updated depending on the input reading from the edge devices. Depending on different environments the model might be updated in various ways.There should be a system that generalises the distributed models, So that the updated models can be used to merge together in a suitable format. That systems can be implemented using the policy technologies. The policies can be implemented on edge devices so that the different models can be learnt on the edge and offloaded to the fog computing level if more resources are needed for computing.

1.1.5 Vectorization

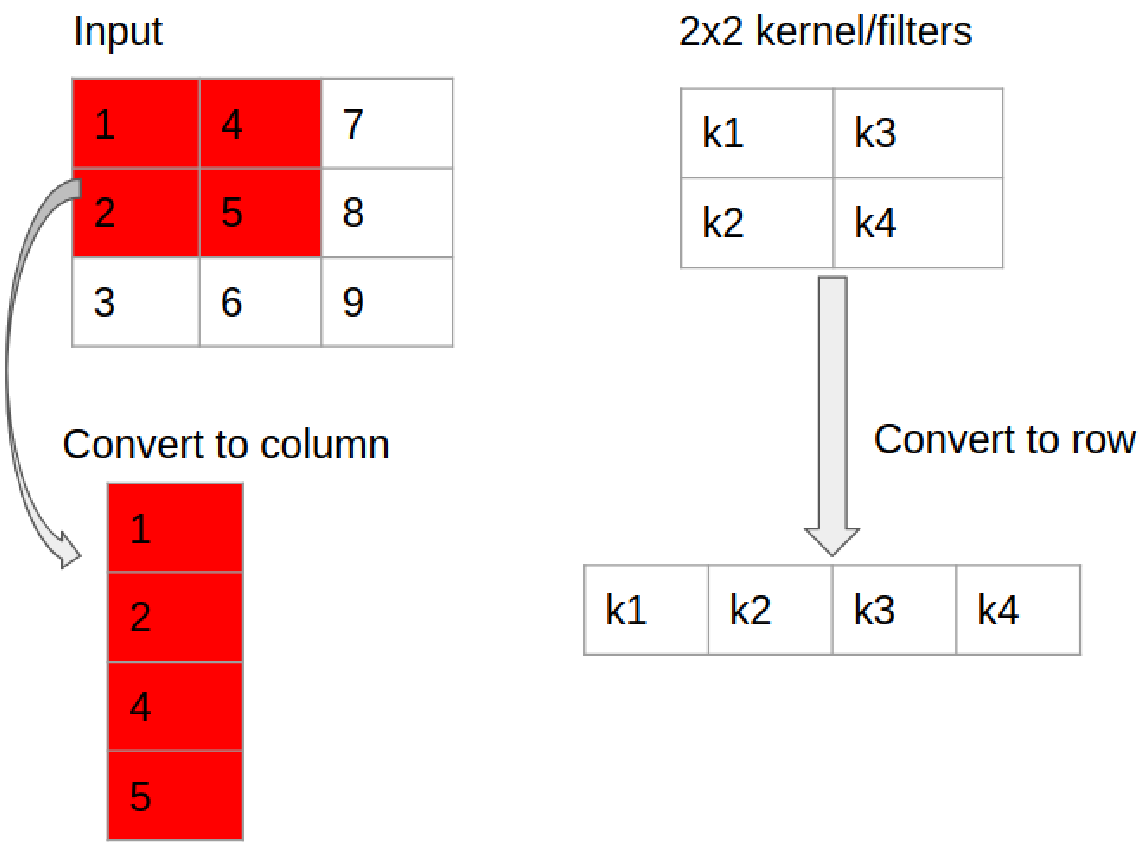

Fig. 1.1 Vectorized Convolution [1]

.

Convolutional Neural Networks consist of multiple layers

- Convolutional Layers
- Pooling Layers

There are different approaches that have been taken toward improving the CNN performance

- Pruning
- Quantization
- Vectorization
- FTT base convolution

In this approach we used Vectorization techniques.By replacing explicit for loops in above mentioned layers performance can be increased. In this research we have used numpy arrays to represent matrices. As shown in the Figure 1.1 input matrix is reshaped as a column and kernel is shaped as rows. Numpy dot product was used to get the desired output from the reshape vectors.

1.1.6 Federated learning

Federated learning is a new machine learning paradigm which allows data parties to build machine learning models collaboratively while keeping their data secure and private.[13][14][12] [15] Federated learning allows users to collaboratively train a machine learning model while keeping the user data private. Federated learning a collaborative computing framework. In here model train though model aggregation other than data aggression by keeping the local data private (within the local device) which is what is opposite to the offloading. Since all our data (images and videos) are generated from edge devices, this method is a better approach to do end to end computer vision task with annotation task moved to edge but only model parameters are sent to the central cloud for aggregations [16].

1.1.7 Multi threaded approach

Each raspberry pi 3B device in the cluster of nodes consists of four cores. We used multi-threads to utilize the four cores and perform computations simultaneously.After the computation is distributed within the nodes , each node will perform computation using multi threads. This utilization reduces the execution time and hence enables real time computations.

1.2 The problem

When Artificial Intelligence models are used in distributed IoT, edge devices have only constrained limited capabilities. Edges IoT devices have limited storage, Computational power. As Cloud computers supply IaaS (Infrastructure As A Service) clouds are rich with high computational resources. Therefore the traditional way of doing AI and ML computation is cloud base In IoT the sensors and actuators are connected to edge devices [3]. Therefore to acquire rich computational resources data has to be communicated over networks to the cloud [17]. But in the communication from edge to cloud the network congestion causes communication latency. Because millions of end devices which generate huge amounts of data need to communicate with cloud servers consuming lots of bandwidth that generate network congestion. Therefore the Real time AI is not performed on the edges of this architecture. When the computation is moved towards the edge. It reduces the network congestions. But in those levels resources are limited. With limited resources such as computation power and storage it makes difficult to generate real time decisions in AI or ML algorithms which requires high computational resources.

1.2.1 The proposed solution

We introduce a methodology for Real time processing at the edge for AI [18] algorithms harvesting the advantages of distributed architecture. Instead of using high computational resources at the cloud, processing is to be done at the edges with limited available resources. Basically the AI model will be processed at the edge, if the available resources are not enough for processing the computation will be distributed to the cluster of available edge nodes. The distribution of the computation is based upon the map reduce like architecture. If the edge nodes are incapable of processing the assigned computation, that particular task will be offloaded [19][20] to the ROOF computing level and will be distributed to the other nodes of the cluster. The offloading will be based upon policy technology that is to be defined. Use of the ROOF computing and fog computing [8] avoids the necessity of reaching the cloud servers. Therefore the processing can be done with lesser latency. That particular architecture leads to achieve goal for real time processing at the edge. When nodes in the cluster processes the federated learning approach is used to process the nodes separately as well as to use ROOF computing level to train global models without accessing the cloud computing level. The whole methodology can be used to accomplish the goal of handling AI and ML computation with limited resources while harvesting the advantages of a distributed architecture for real-time processing at the edge [3].

Chapter 2

Related work

Handling AI and ML computation with limited resources while harvesting the advantages of a distributed architecture for real-time processing at the edge has a scope which is associated with various contexts.

2.1 Use of Map Reduce to distribute the computation

The distribution of computation can be categorised basically under three main scenarios [21]:

- When the training data is large.
- When data to be classified is large.
- When the Neural network consists of a huge number of nodes.

When the training data set is larger training data has to be divided using mapper function is Map reduce [22]. Because training a huge volume of data costs high computational resources as well as a high amount of time. So different sets of data will be provided to each node in the cluster and the whole portion of data to be classified will be provided to every node, because the volume of data to be classified is small compared to the volume of training data. When each node in the cluster is trained with different chunks of training data. That will generate different AI models at the different nodes of the cluster. So federated learning techniques [15] should be used in order to define the suitable training model out of the different models that will be generated.

When data to be classified is huge that particular data will be distributed to all the nodes using the mapper function. For each node in the cluster the same chunk of training data will be provided because the training data is small compared to data to be classified in this scenario. Each node will generate its output and the reducer function will make the final output from all of these results.

When the neural network consists of a large number of neurons, computation cost increases. So using map reduce the neural network can be distributed over multiple nodes. There are a number of iterations involved with the feedforward process while the backpropagation is done in the last iteration. In each iteration reducer collects the outputs and merges all outputs from each mapper. For this approach computers with high computation power are used. In this work clusters consist of low computation power nodes.

2.2 Top Down based approach

Basically in top down approach the AI model is trained or developed at the cloud computing level using the high computational resources.Then the model is deployed to the edge devices.There are various techniques used in the scenarios. The whole process can be considered as a tree like structure. Root is considered as the central cloud and the leaves of the tree are the edge devices. A technique to look at the AI application built is that it is considered as the Cognitive Processing Elements(CPE). Then build a chain of CPE. A CPE is operated in basic four phases [23]:

- Discover phase
- Deploy phase
- Operate phase
- Retain phase

Basically in the discovery phase the basic idea is to develop the model using automated or user defined technique. In some scenarios multiple models might be generated and there should be methods to decide and select the best models out of those. After the model is developed they should be added into edges using the Deploy phase. Here the model is packed into docker containers and placed in a shared repository. Therefore edges can access that particular repository. Above mentioned chain of CPE is implemented as a Node-Red flow.

Under the operation phase microservices are implemented on edges which are responsible for instantiating the chain of CPE flow. Docker containers in the shared repository will be executed at the edges by their microservices. Retaining phase is designed such that feedback mechanisms will be provided towards the cloud and then models can be updated in the cloud computing level.

This top down approach uses high computational resources available at the cloud level and then deploys the models to edges. In contrast what we propose is to use the limited resources available at the edges, so our solution is a bottom up approach.

2.3 Edge and Fog Computing Enabled AI for IoT

By defining the edge and the fog nodes which are intermediate nodes between the edge and the cloud, these fog nodes can be used to increase the efficiency of distributed AI algorithms. This approach is based on containerization. Due to containerization, the containers can be migrated horizontally and vertically between the cloud computing layer and the edge. This architecture leads to more flexibility of Fog computing layer [24]. This method using containers it can execute and classify locally in nodes and offload to the fog and the cloud whenever additional computation is required. The approach uses the Distributed Deep Neural Networks. We can use the ideology to use fog computing. But to enable real time computations, use of both ROOF computing and Fog Computing [8] can be much efficient. This approach is focused on hardware architectural solutions in the terms of making energy efficiency rather than developing algorithmic solutions which we focused on achieving.

2.4 Fog Computing

Defining the fog nodes in Fog Computing and the methods to Collaborate the Fog Nodes [8] are described;

- Cluster
- Peer to Peer
- Master Slave

Cluster architecture describes the way to form a collaborative execution environment by forming a cluster of nodes.Peer to Peer architecture can be formed in hierarchical manner as well as in a flat order. Master slave designed such a way that master fog controls the process load, resource management of slave nodes. The collaboration methods of the Fog Computing can be used on our node collaboration in ROOF Computing and Fog Computing. Limitations of the architecture is that master-slave uses high bandwidth between master and slaves. Therefore the better approach would be to define a hybrid methodology consisting of master-slave along with peer to peer and cluster.

2.5 Caching and Communication by Federated Learning approach

Basically tries to integrate Deep Reinforcement Learning Techniques with Federated Learning Framework. The idea of exchanging the learning parameters for better training and inferencing the model is described. The approach has used the techniques to reduce the system's communication load. Deep Reinforcement Learning(DRL) is used to jointly manage communication and the computational resources. While Federated Learning is used as a Framework for training the DRL agents [4]. The distributed architecture leads to different updates to model at different nodes, because of the independent training. So the necessity of merging the updates arises. There are problems associated with the models in the nodes as the nodes are not perfectly similar. Therefore this unbalance of devices has to be figured out, As a solution this research proposes to use FedAvg algorithm [25]. We can use this algorithm to aggregate the global model at a ROOF or the fog level based on the different models developed at the edges.

2.6 Federated learning to ensure the privacy and smart models

As we are using federated learning non of our training data is moved to the cloud. so model can be trained without compromised the privacy of the user's data. so it ensures the privacy. As the updated model can be used to make predictions on the user's devices there is very low latency in the predictions. As this is going to be a collaborative training process model get smarter over time.

2.7 Vectorization

There are different types of approaches that have been taken to improve the performance of CNNs. Some are pruning, quantization and vectorization [26]. Pruning removes the number of connections of a CNN. This method weakens the CNN as some weights are removed in this process. In quantization approach word length of weights and activations are reduced. FFT based convolution and Winograd convolution are some other approaches which improve the CNN performance [27]. In FFT convolution operations perform in frequency domain. This method is suitable for larger filter sizes. For the FFT based convolution additional transformations are required which is a down side of a FFT convolution. Winograd Convolution also involves the transformation of input matrices and kernel matrices to perform the convolution [28]. In vectorization approach input matrices and kernels are transformed inorder to perform the matrix multiplication which improves the performance of the convolutional operations [29]. As the input matrices and kernels are transformed this method requires more memory.

2.8 Computation offloading

There are different studies involved in computation offloading. In offloading process computation is immigrated to the resourceful server or device from the limited resource device. This migration involves communication delays and energy consumptions so in order to perform the migration need to make decisions. Different approaches are proposed in different studies for this scenario. Some are Q learning based approaches, linear programming based approaches, and approaches based on defining cost functions for the transmission delays [30][31][32].

Q learning based implementation requires additional computation power in order to make the decisions. Linear programming and cost function approaches transmission time and energy consumption for each convolution layer need be calculated before the decision making process. Which is an additional overhead for the decision making process. Simple policy based approach is discussed in these papers [33][34]. By defining a simple policy, limited resource devices can make decisions based on the computation time and the transmission delays. The calculation of the computation time of the convolution operation using CPI gives some disadvantages. As the different devices have different CPIs with different pipelines and different architectures. GFLOPs base approach gives the advantage over the CPI methods as the manufacturers provide the information of the GFLOPs of the devices [35].

Chapter 3

Methodology

Our main goal is here to process real time data in using Distributed IoT. But the main issues in IoT devices are they have very low processing power and memory. So it's very hard to run AL algorithms in IoT devices due to processing and memory bandwidth issues. So the current approach is with the help of cloud computing [36] we can distribute the processing workload to the cloud [17] and get back the result in the edge layer, using this method we can reduce the processing time it takes to compute the algorithm. But the main downside of this approach is the time it takes to upload and data to the cloud and download data from the cloud. So this time delay may affect the real time data processing of the edge.

To overcome this problem, a fog layer is introduced [37], a new layer is added in between the cloud and the edge layers. So instead of sending data to the cloud and processing it in the cloud, now data is sent to the fog layer, this will reduce the upload and download time reducing the time it takes to send and receive data, this helps to archive near real-time data processing for IoT devices. But still the issue of real time processing is not archived. The main issue is this technique is the edge device is always offloading the data to the cloud which causes a delay.

what if we can set parameters to define when to offload the data to cloud and try to process the data in real-time if possible. This is known as policy technology [38]. By utilizing this method, the edge device can decide when to offload the data according to their capability of the IoT device. We can set different threshold parameters to when to decide the offloading. Usual threshold parameter is 80% of total processing power. When processing power reaches to 80% then the device will offload the work to the cloud, unless it will process in real time. So this method will only offload when it's necessary so we can archive near real-time results.

Few factors will come into play when it come to deciding the threshold parameter, if we use a lower threshold parameter then, then it will more often try to reach that value so the edge device will more likely to offload the data to upper layers resulting the more often data offloading which will affect the real time data processing of the edge device. And there is a resource wastage in the edge as well. Because it can go for higher processing without any bottleneck of the processing capabilities. And if we go for a very high threshold parameter that can affect the result as well. Because when the algorithm is running on the edge and it's reached the threshold parameter limit, we may run out of time to send data to the upper layers. so it causes a time delay in processing which affects the real time process. We need to find the right balance between the threshold parameter [38].

In our research we are trying to run complex algorithms on a low powered edge device in real time. We picked real time object detection algorithms to run on our edge devices [39]. Because object detection algorithms require extensive processing power and it has to be run on real time. So our goal is to run an object detection algorithm on an edge device.

We look though many object detection algorithms like Fast R-CNN(Convolutional Neural Network), Faster R-CNN, Single Shot Detector (SSD), YOLO [1], after we referring and studying all those object detection algorithms, we finally decided to use Yolo as our algorithms and trying to modify it and run it on edge level.

3.1 Why YOLO and how does it work ?

YOLO is extremely fast and accurate compared to other object detection algorithms [40]. Compared to other algorithms YOLO able to provide higher mAP(mean average precision) while consuming the same amount of computation power.

YOLO see the image globally when making predictions. Unlike sliding window and region proposal-based techniques, YOLO sees the entire image during training and test time. YOLO makes less than half the number of background errors compared to Fast R-CNN. YOLO uses the features in the entire image to predict the bounding box. And also it predicts the all the bounding box of all classes simultaneously. In YOLO it divides the image into n x n grids. If a center of an image falls into a particular grid then that grid is responsible for making the bounding box for that particular object. So each grid cell will predict the B bounding box and the confident scores for that box , confident score is how confident that image falls into a particular classifier. If nothing falls into a bounding box, then the confident score is zero. confidence score is defined as

$$Pr(Object) \times IOU_{predicted}^{true} \tag{3.1}$$

Equation 3.1 Confidence Score

IOU is Intersection over union between the predicted box and ground truth of the image box. Each bounding box consists of 5 predictions: x,y,w,h and the confidence. The (x, y) coordinates represent the center of the box relative to the bounds of the grid cell. The width and height are predicted relative to the whole image. Finally the confidence prediction represents the IOU between the predicted box and any ground truth box. Each grid cell also predicts C conditional class probabilities, $Pr(Class_i|Object)$. These probabilities are conditioned on the grid cell containing an object. We only predict one set of class probabilities per grid cell, regardless of the number of boxes B. At test time we multiply the conditional class probabilities and the individual box confidence predictions

$$Pr(Class_i|Object) \times Pr(Object) \times IOU_{predicted}^{true} = Pr(Class_i) \times IOU_{predicted}^{true} \tag{3.2}$$

Equation 3.2 conditional class probabilities and the individual box confidence predictions

which gives us class-specific confidence scores for each box. These scores encode both the probability of that class appearing in the box and how well the predicted box fits the object.

3.2 Methodological approach

As shown in the example (as seen in the figure 3.1) YOLO model detects this as a regression problem. It divides the image into an S × S grid and for each grid cell predicts B bounding boxes, confidence for those boxes, and C class probabilities. These predictions are encoded as an S × S x (B x 5 + C) tensor.

Here s = 7 and and bounding boxes are 2 , and let say we have 20 labelled classes so C = 20. Then our final reduction is a 7 x 7 x 30 tensor.

Even this kind of algorithm is very computation complex for an edge device. Even if we run an algorithm like this on an edge device it will not give real time results.

So Our first approach is to divide the computation power across multiple edge nodes in real time. In the current implementation of YOLO, they only run the algorithm in one single unit. (as an example in apple detection platform they run it on Nvidia Jetson board). So if we can distribute the computation workload [41] across the multiple

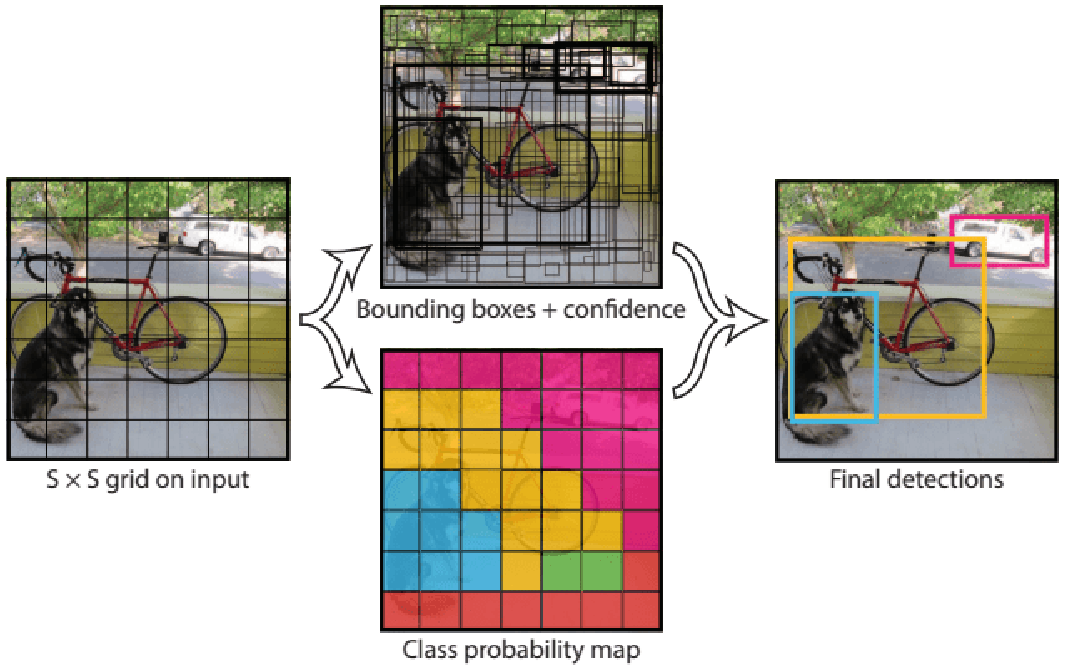

Fig. 3.1 Dividing the image into Grid and bounding boxes [1]

.

nodes (in multiple edge devices). Then we may able to archive real time results as multiple nodes will increase the computation power. We can archive this So by applying map reduce and distribute this in multiple low computational level devices(in multiple hardware devices). Using that method, we can use multiple lower powered devices to run the complex algorithms real time at edge level. Or we can distribute the multiple filters [42] of the convolutional network across the multiple edge devices. Then we will be able to parallelize the computational workload in multiple nodes which again increase the computational power.

Our second approach is to reduce the computation cost of the algorithm by simplifying the final output tensor. We can achieve this by reducing the number of class probabilities and the image division grid of the YOLO.

Using the previous example if we use 2 class probabilities(human, non human) and 9 grids. Then the output tensor can be calculated using $S \times S \times$ (B x 5 + C), where $S = 3$, $C = 2$ and $B = 2$ then the output tensor is 108, which is 92% reduction of the output tensor matrix.

3.2.1 Vectorization using im2col technique

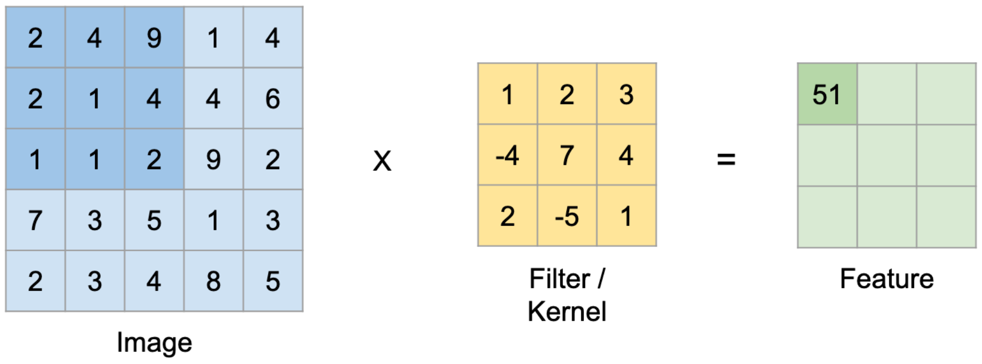

Fig. 3.2 Convolution operation.

When consider the CNN network, the major operations are,

- Convolutional operation
- Pooling operation

In convolution operation, the input matrix is multiplied by the kernel. As shown in figure 3.2 the kernel needs to move through the whole input matrix based on the given stride which defines how many columns or rows the kernel should move next. To move the kernel through the whole matrix consumes time which slows down the operation. Im2col technique can be used to overcome this problem which basically removes the need of moving the kernel through the whole input matrix. We represent input matrices and kernels using numpy matrices. So the implementation of the im2col technique is done using the numpy strides. Which gave the ability to reshape the matrices in order to do the convolution operation in a vectorized manner.

Using numpy strides each receptive field is turned into a column. As shown in the figure 1.1 2x2 receptive field is converted to a column matrix and each kernel is reshaped into a row matrix. Figure 3.3 shows the converted input matrix where each receptive field is stacked side by side in a single matrix, Then the output is multiplied by the kernel matrix which is reshaped into row matrices. For the pooling operation the same procedure is used where for a given kernel shape input matrix is reshaped and gets the output which depends on whether max pooling or average pooling.

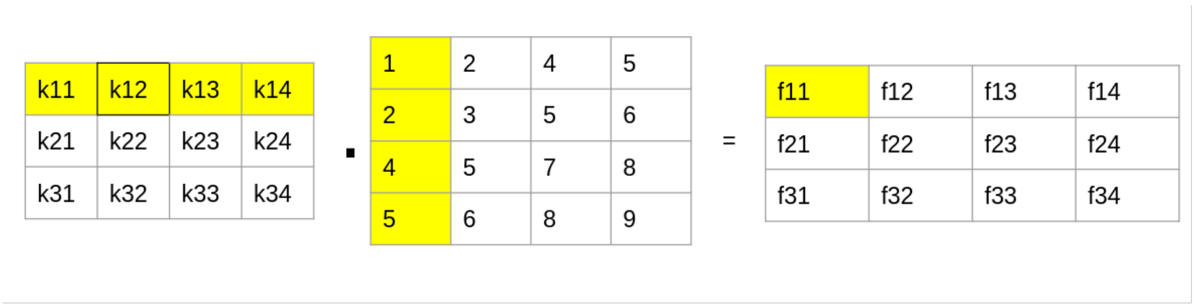

Fig. 3.3 Vectorization using Im2Col technique.

3.2.2 Computation offloading

Most of the IoT devices have limited resources compared to the cloud. When it comes to the AI and ML processing it requires more computation power and resources. Even though the computation distribution, vectorization and parallelization techniques apply some times it may not give the maximum benefits. We used offloading techniques to get the advantage of more resource full servers.We compared the execution time in raspberry pis with the execution time on the server and the communication delays based on that we defined a policy to determine whether to offload or execute locally.

Symbol	Meaning
ω	amount of computation
S_m	CPU speed of the raspberry pi
S_s	CPU speed of the server
B_u	upload bandwidth
B_d	download bandwidth
d_s	sending data
d_r	receiving data

Table 3.1 Table of Symbols.

$$d_s/B_u + d_r/B_d + \omega/S_s < \omega/S_m \tag{3.3}$$

We compute the amount of computation using the number of floating point operations and GFLOPs of the executing device. When multiplying to vectors of n elements there involves 2n-1 arithmetic operations, then we divided it by the GFLOPs and compute computation time. Then we use equation (3.3) with the communications delays to determine the offloading decisions. As these IoT devices (Raspberry pies) have limited amounts of memory we also considered the memory usage.

$$Memory\ usage > Total\ memory \times threshold \tag{3.4}$$

As shown in the equation (3.4), given a threshold value, we computed the memory usage of the operation and if the above equation satisfies then we offload the computation to the upper layers.

3.2.3 Multi threads for computation

Raspberry pi have four cores,so any computation in a raspberry pi can be utilized to compute using all the four cores.Quad core device contains four cores.Here multi threads can be applied on the computational costly parts like OpenMP but available only for C,C++ and FORTRAN. But Python Global Interpreter Lock lets only one thread at a time to be executed. As OpenMP can create threads on C ,Cython converts to a separate executed c file but again the code conversion is expensive.

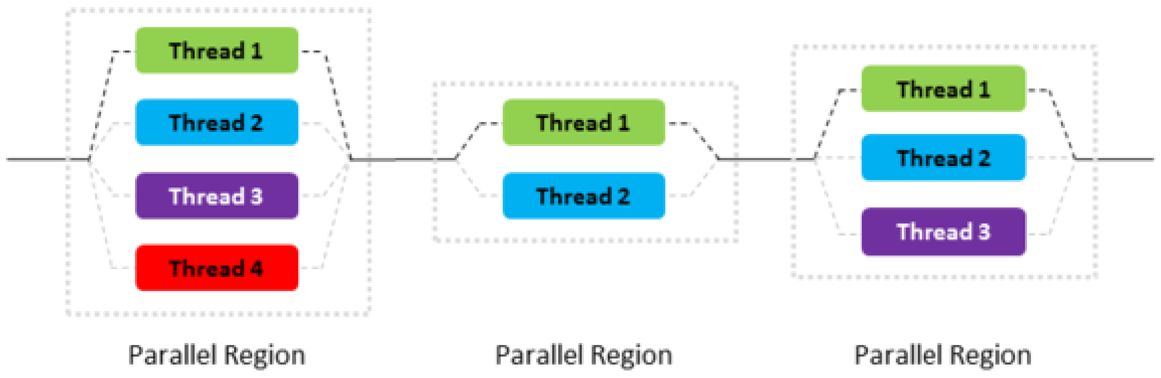

Fig. 3.4 PyMP parallel regions .

Therefore we use PyMP . It uses the Fork and join model enabling use of multiple threads. Using PyMP most computationally costly parts of the algorithm is executed parallel using threads to utilize the four available cores within a raspberry pi node. While the rest of the algorithm is executed sequentially. While the threads make certain parts of the convolutional network the shared variables of certain threads should be considered. Assigning too many threads causes overhead . Therefore optimal number of threads should be assigned. The most computationally costly parts of the YOLO algorithm are selected and then different number of threads were applied. Then the optimal number of threads were selected by measuring the best execution time.

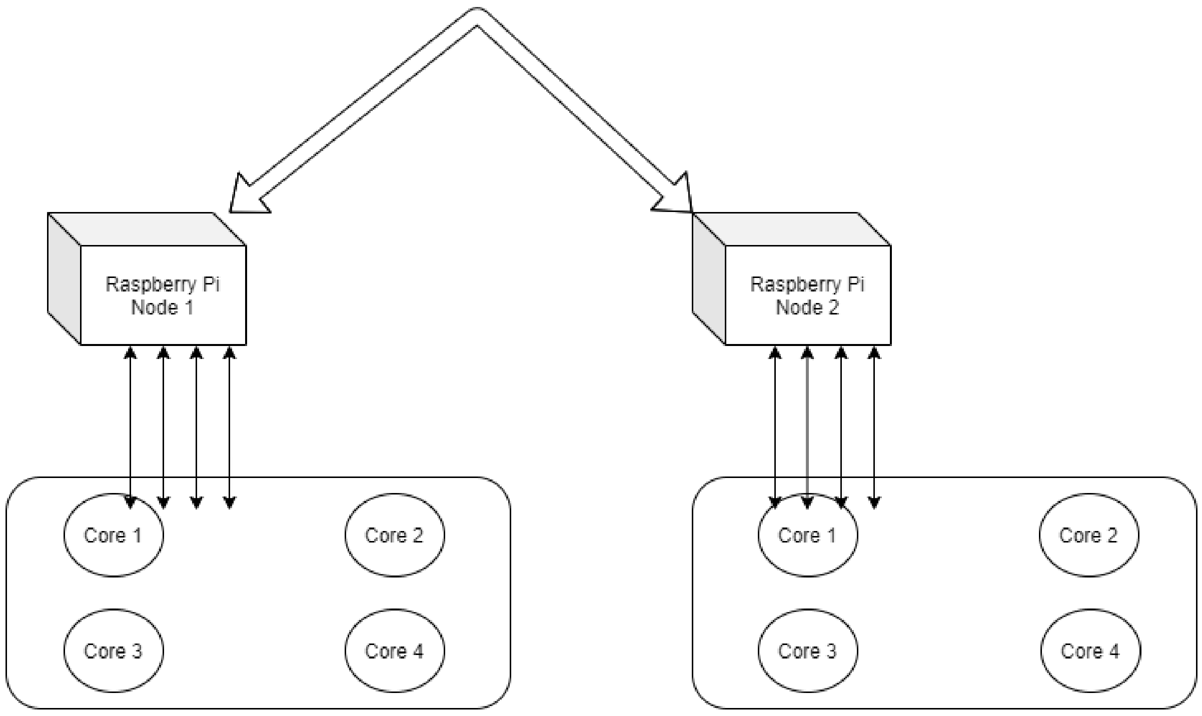

Fig. 3.5 Application of multi threads per each raspberry pi node.

3.2.4 Vectorization with multi threads

When we combine the two approaches used to optimize the computations we had to use multi threads to compute vectorized CNN computations. Here we used an optimized BLAS library. So vectorized CNN will be performed using multi threads in OpenBLAS.This methodology combines both resource utilization using multi threads and performance improvement using vectorization techniques together which we used for the optimization.

3.2.5 Federated Learning

The main advantage of the Federated learning is, it helps to train a model while preserving the privacy of the model data. In here model is train though model aggregation other than data aggression by keeping the local data private (within the local device) Since all the data is collected from an edge device, this is a better approach for doing computer vision tasks. Because all the annotations are done on the edge devices but the model parameters are aggregated in a central cloud server.[43] This method ensures the privacy of the users. Once the model parameters are aggregated, then the global model is pushed to the user's devices. So there is low latency in the predictions. As this is going to be a collaborative training process, the model gets smarter over time.

Chapter 4

Experimental Setup and Implementation

4.1 Prototype

In this work we choose object detection as our use case which needs high computation power for the training and prediction phases. Raspberry pis are used as the end devices, Which runs the YOLO algorithm. YOLO algorithm is based on convolutional neural networks. Given an image, it feeds to an convolutional network and get an output based on partitioning make on the image and number class. Figure 4.1 shows a input and output example where 600x600x3 input is feed into the CNN where output is 19x19x425.In this example number of predicting classes are 80 while 5 anchor boxes are used for each grid cell.

The convolutional neural network [44][45] consists of multiple filters in each layer so by dividing kernels among multiple raspberry pis, the computation can be distributed and parallelized [41]. After computation done parallely then the output is merged and forward to the next layer. We use the kernels with the shape of (h, w, nCprev, nC) where the h and w represent the height and the width and the nCprev represent the number of channels of the input matrix and the nC represent the number of kernel of the shape of (h ,w, nCprev). So the computation distribution we partition the nCprev depending on the available nodes and then perform the convolution on the input matrix and then combine the each output of the each node.

For the pooling layers we distribute the input matrix between the available nodes and perform the pooling operation. As shown in the figure 4.1 the master gets the input image and it distributes the computation among the slave nodes.

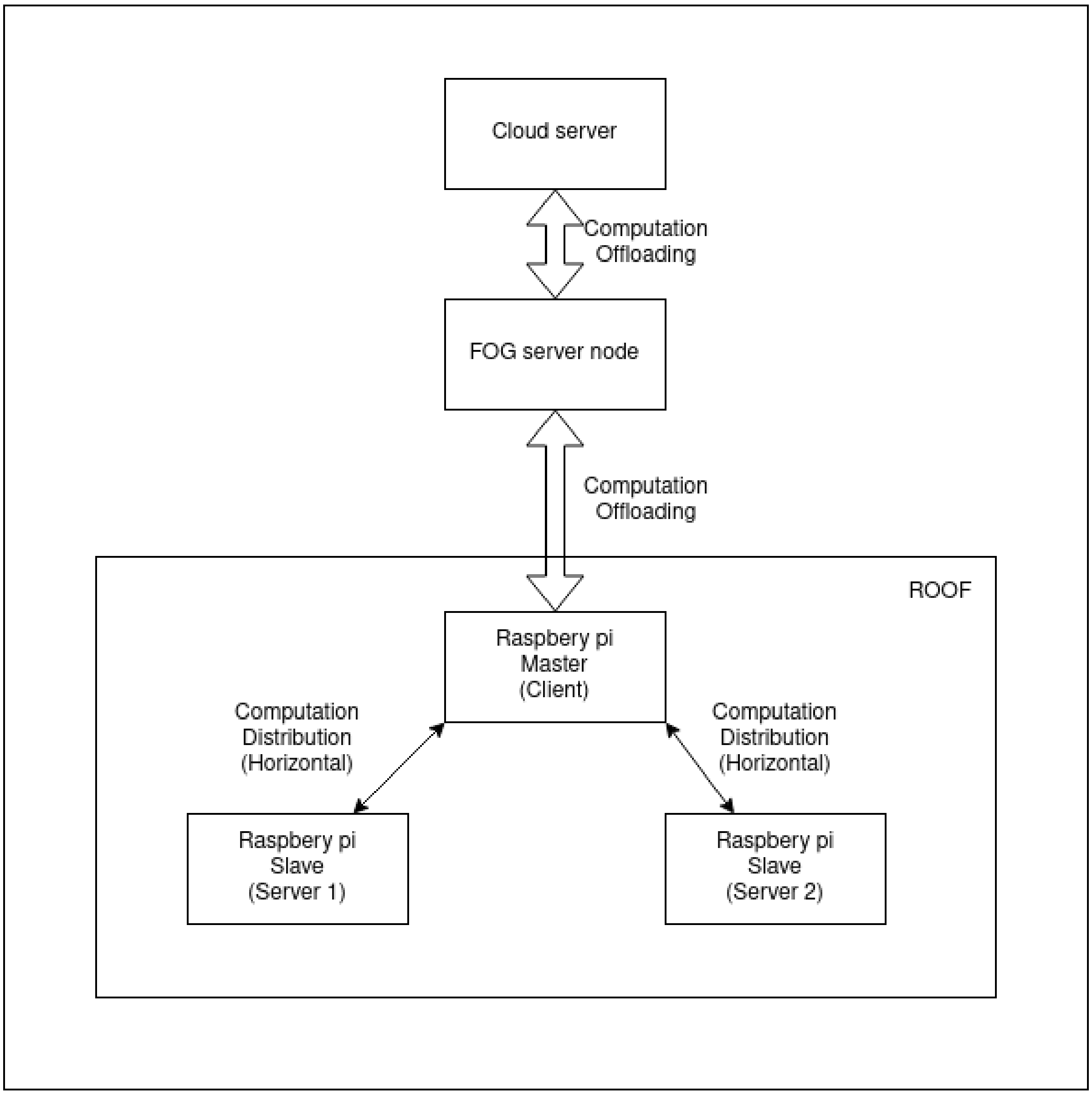
Cloud server
Computation Offloading
FOG server node
Computation Offloading
ROOF
Raspbery pi Master (Client)
Computation Distribution (Horizontal)
Computation Distribution (Horizontal)
Raspbery pi Slave (Server 1)
Raspbery pi Slave (Server 2)

Fig. 4.1 Architecture of the system.

Master acts as a client where the slave nodes act as servers. So we use client server architecture to communicate between the slave nodes and the master nodes. Server nodes are always listening to the incoming data , when the master node receives an image then it distributes the computation to the slave nodes which are always listening for the requests.

For convolution operations each node uses vectorization techniques and parallelization techniques. For matrix implementation we used numpy matrices. For the vectorization

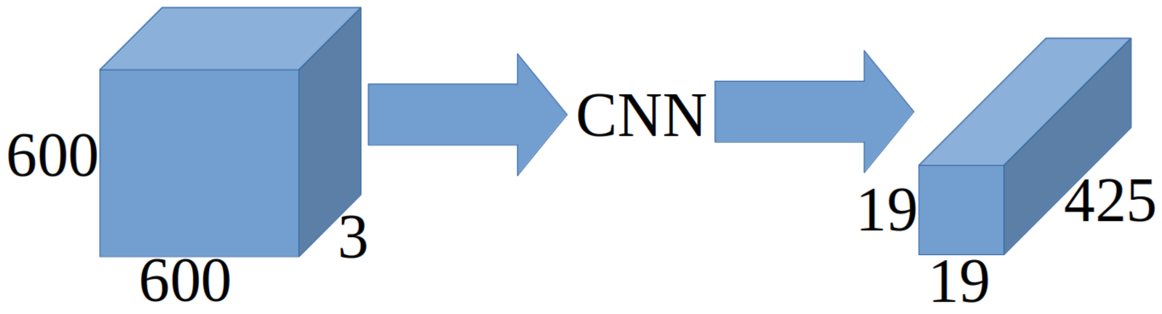

Fig. 4.2 CNN which takes 600x600x3 input and produce 19x19x425 output.

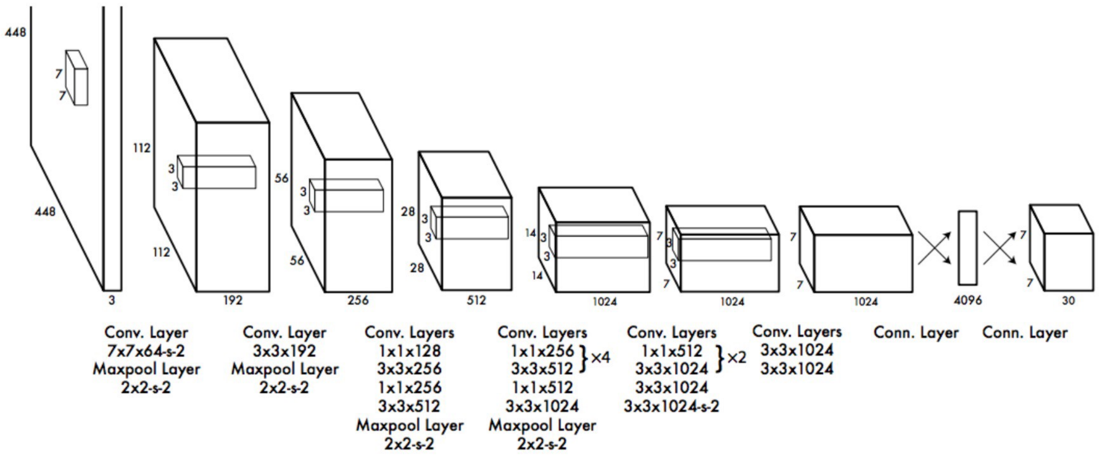

Fig. 4.3 YOLO architecture [1].

we used im2col technique and we got the advantage of numpy strides to manipulate the matrices in order to perform the convolution operations in a vectorized manner.

Federated learning methods are used for the training network and object detection, Where each raspberry pi or a raspberry pi cluster trains a model based on local data and then the learned parameters are sent to higher layers (Fog, ROOF). Then the aggregation done on that layer to train a global model.

For the data set we used We randomly captured these images of different scenes at different times from 26 street monitoring videos with 704×576 pixels. Eventually, we select a total of 2,544 items from these images with 7 object categories. Each image has at least one labeled object, and may have multiple labels of this same category in one image. The object labels are basket, carton, chair ,electromobile, gas tank,sunshade and table.

while training YOLOv3 was via Adam with an initialization learning rate of 1e-3. We adapt the original Federated Averaging (FedAvg)[25] algorithm to framework, we modified FedAvg algorithm to a pseudo FedAvg algorithm because there is an effect of data division for the Federated learning

Algorithm 1 Pseudo FedAvg

```
Input: $N$ client parties $\{c_k\}_{k=1..N}$, total rounds $T$, and Server side $S$;
Output: Aggregated Model $w$
    $S$ initializes federated model parameters, and saves as checkpoint. Client parties $\{c_k\}_{k=1..N}$ load the checkpoints.
  for $t = 1, ..., T$ do
      for $k = 1, ..., N$ do
          $w_k = w^{(t)}$
          each client $\{c_k\}$ do local training:
          for $i = 0, 1, ..., M_k$ do
              ($M_k$ is the number of data batches $b$ in the client $c_k$)
              client $\{c_k\}$ computes gradients $\nabla\ell(w_k, b_i)$
              update with $w_k = w_k - \eta\nabla\ell(w_k, b_i)$
          end for
          save $w_k$ results to checkpoints.
      end for
      $S$ loads checkpoints and get averaged model with $w^{(t)} = \frac{1}{N}\sum_{k=1}^{N} w_k$
  end for
  return $w^{(T)}$
```

Fig. 4.4 Pseudo FedAvg .

4.2 Research instruments

When it comes to object detection YOLO is a complex algorithm which consumes high computational resources [39]. DarkNet who are the founders of YOLO have trained a fairly large dataset using GPUs. Because the high complexity of the algorithm due to the complex Convolutional Network takes a very long time to train. To take an idea about the need of high resources given below is an instance of training COCO train 2014 (117,263 images) data set [2].

4.2.1 Resources used

We used Raspberry Pi 3 boards as our end devices.The specifications for the Raspberry Pi is as follows.

- Quad Core 1.2GHz Broadcom BCM2837 64bit CPU
- 1GB RAM
- BCM43438 wireless LAN and Bluetooth Low Energy (BLE) on board
- 100 Base Ethernet
- 40-pin extended GPIO
- 4 USB 2 ports
- 4 Pole stereo output and composite video port
- Full size HDMI
- CSI camera port for connecting a Raspberry Pi camera
- DSI display port for connecting a Raspberry Pi touchscreen display
- Micro SD port for loading your operating system and storing data
- Upgraded switched Micro USB power source up to 2.5A

Compare the resource availability of a raspberry pi board with a jetson board.

SpecificTION	Jetson Nano	Raspberry Pi
GPU	128 NVIDIA CUDA® cores	-
CPU	Quad-core ARM Cortex-A57 MPCore processor	Quad Core 1.2GHz 64bit CPU
Memory	4 GB	1 GB

Table 4.1 Comparision of raspberry pi with jetson nano [2].

It is clearly shown that we have used the raspberry pi which do not have GPU and also it consists of very low computational resources compared to the Jetson boards.

YOLO is tested on these jetson boards due to its high resource availability with GPU enabled environment.

Machine type preemptible n1-standard-8 (8 vCPUs, 30 GB memory)

CPU platform: Intel Skylake

GPUs: K80 ($0.14/hr), T4 ($0.11/hr), V100 ($0.74/hr) CUDA with Nvidia Apex FP16/32 HDD: 300 GB SSD

GPU	n	Batch size	img/s	Epoch time
K80	1	32x2	11	175min
T4	1	32x2	41	48min
T4	2	64x1	61	32min
v100	1	32x2	122	16min
v100	2	64x1	178	11min
2080Ti	1	32x2	81	24min
2080Ti	2	64x1	140	14min

Table 4.2 Training time for COCO data set under Cloud System [2].

The table 4.1 shows the statistical data of training the YOLO algorithm with GPUs which contains very high computational resources. The resources are allocated in the cloud environment. Even with such powerful resources available the time to train exceed 60 min at its best [2].

Our implementation targets to distribute the computation to edge devices with limited resources. We choose raspberry pi 3 boards as our edge nodes. Which have very limited resources compared to GPUs used in above scenarios. To distribute the computation among the nodes at the edge, we require a few raspberry pi boards for the cluster we instantiate the Debian OS on virtual machines. The environment is a virtually mounted Debian 10 operating system which is allocated 512 Megabyte of RAM and 1 GigaByte of Storage at the time of instantiation. This particular environment is a very constrained one compared to the GPU enabled cloud system as given above.

4.3 Data manipulation and Testing

For testing we checked our implemented CNN's operations pooling and convolution, using tensorflow. For that we gave the same input to the both tensorflow implementation and our implementation. We checked each operation separately and verified the implementation.

4.4 Pitfalls and workarounds

When the use case was to determine the AI algorithm as Object detection we went for YOLO [1]. It is a Complex convolutional network based algorithm. We had to learn how the Convolutional network works [44][45]. Then we went through how Convolutional Neural Networks are applied on object detection instead of basic Neural Network [40]. Meanwhile we learnt the map reduce techniques so we can apply that to distribute the computation [22]. Applying the Map reduce on ROOF nodes was a challenging task. Distributing the computation has to be done with several nodes. We tested the compatibility of running YOLO on different IoT devices. In our research we are using the raspberry pi as the edge node. We required several raspberry pi boards to process to use in our distributed architecture. We optimized our algorithm using both multi threads and vectorization. Then we had to combine both optimization techniques together. Here merging together done using a BLAS library . The among the BLAS libraries we got the best performance using OpenBLAS library which is an optimized Basic Linear Algebra Subprograms library. Here the combination of two techniques directed our research direction from PyMP multi threads towards use of BLAS library.

Chapter 5

Results and Analysis

5.1 Results

The Darknet's different YOLO versions were instatiated on a single Raspberry Pi node.

Algorithm	Result
YOLOv3	Segmentation Fault(on calculating weights)
Tiny YOLOv3	Segmentation fault(on CNN)

The Segmentation faults occur when the program tries to access memory beyond its reach. That implies Darknet's YOLO is computationally excessive for Raspberry Pi devices. Also YOLOv3 stops at calculating weights while Tiny YOLOv3 stops at CNN. Which shows optimized version can do better computation, but even Tiny YOLO can not complete its task.

Then our custom YOLO implementation was tested on single node with Map reduced version on multiple nodes.

Number of Nodes	Total Execution time (s)
Single node	270.51
Two nodes	89

The distribution of computation with multiple nodes reduces execution time. After the computation is divided among the cluster nodes using the Map Reduce techniques, we used multi threads to utilize the resources.The computation is parallized within the cores of each device.

No.threads	Node1(s)	Node2(s)	Join(s)	Total(s)
2	38.012	8.4214	31.9943	70.0063
4	24.21	1.9367	23.57	47.78
8	24.12	1.034	25	49.12
10	20.74	1.0015	21.0055	41.74
15	21.9844	0.973084	22.3142	44.29
20	25.9405	1.0015	21.0554	46.9959
50	23.20	1.105	23.2363	46.43

Here a variation of total execution time can be seen with respect to the number of threads. Therefore to find the optimal number of number of threads the results were tabulated.

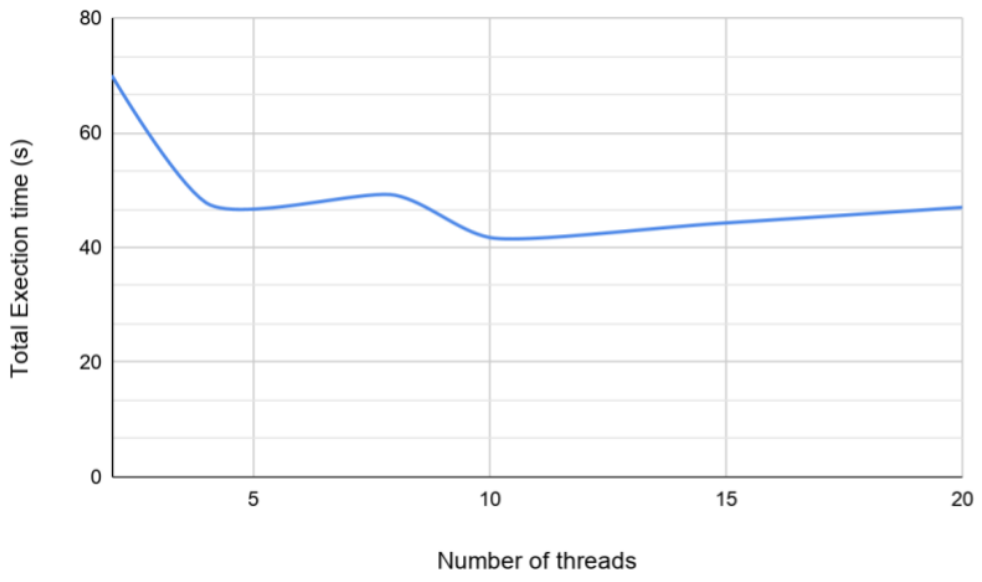

Fig. 5.1 Number of threads Vs the Total execution time

Here increasing the number of threads reduces the execution time. But after a point the execution time increases, because the synchronization overhead happens in a limited cores available environment. According to the diagram the optimal number of threads per node is 10. Then the optimized code was compared with the same algorithm tested on Cloud with very high resources.

Algorithm	Environment	Result
YOLOv3	Raspberry Pi	Segmentation fault
Tiny YOLOv3	Raspberry pi	Segmentation fault
Optimized YOLO	2 Raspberry Pi nodes	41.74(s)
Optimized YOLO	Colab cloud (GPU enabled)	29.430(s)

The resource utilization with multi threads was combined with the vectorization approach and measured the performance gain in the distributed computation in raspberry Pi.

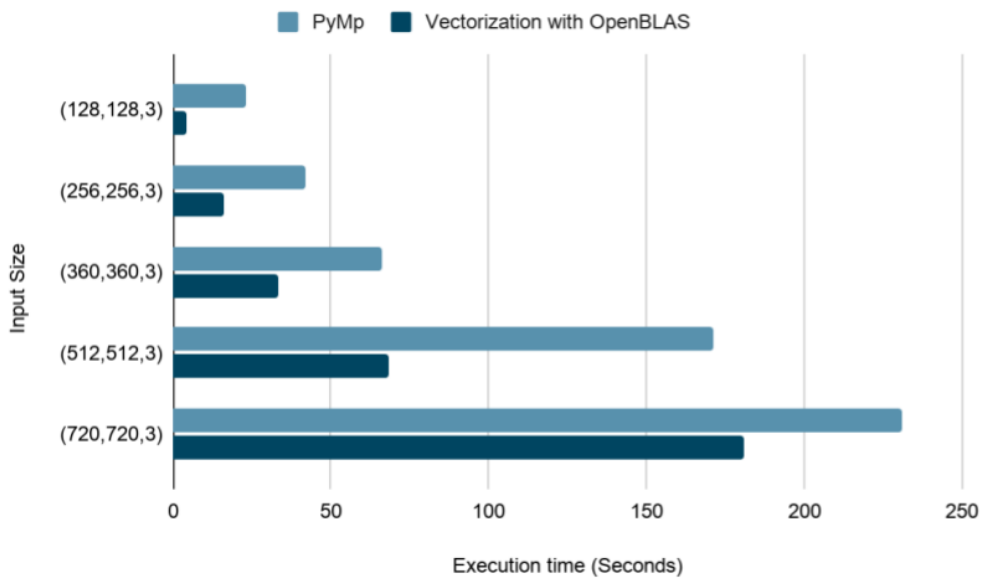

Fig. 5.2 Total execution time vs input size for Pymp multi threads and vectorization with OpenBLAS optimization .

From figure 5.2 ,with the increase of the input size, the total execution time for increases for both approaches. But for all the input sizes the vectorization optimized with OpenBLAS performs better than as it utilizes the resources efficiently.This approach further enables the real time computations.

We used the equation (3.3) to make offloading decisions in the raspberry pi. The computation time is calculated using the GFLOPs in the given device. For the amount of computation, the number of floating point operations in the given convolution is considered and then it divided by the GFLOPs of the device. Fig. 5.3 shows the results of the estimation with actual time.

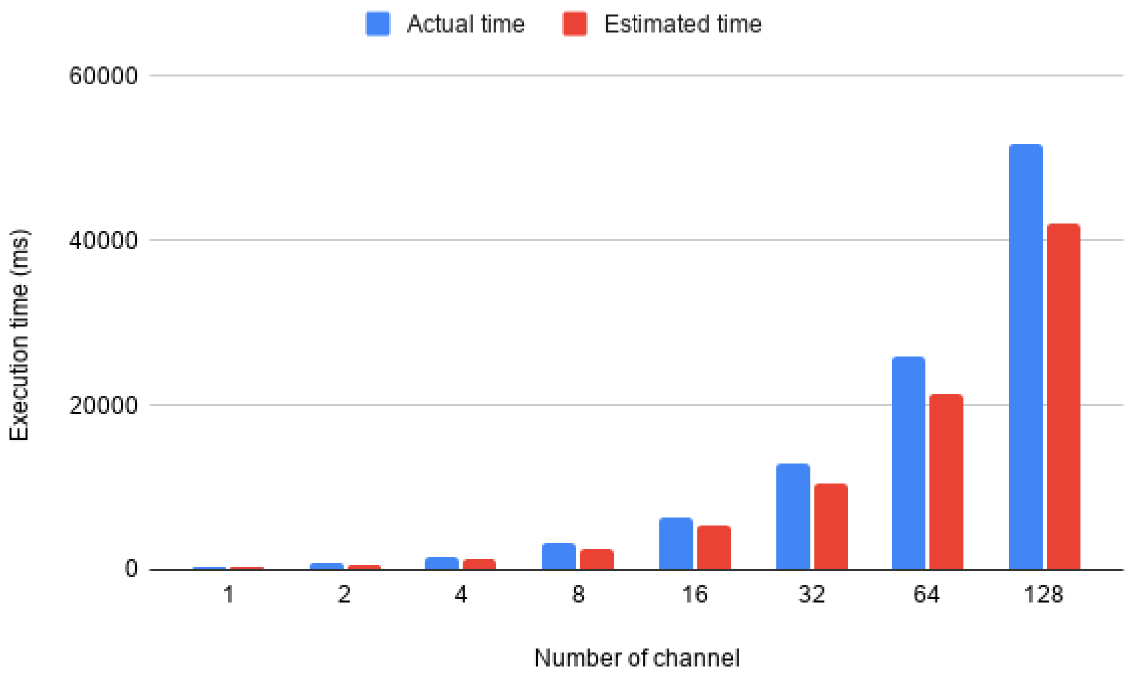

Fig. 5.3 Actual processing time vs Estimated processing time with the varying channel size.Input matrix (64, 64, channels) with kernel shape (9, 9,channels, 256).

5.2 Analysis

Based on the results it is evident that the object detection algorithm we choose, YOLO which has a complex Convolutional Neural Network cannot be run on a single raspberry pi 3 board due to resource constraints. Both Darknet's YOLO and Tiny YOLO cannot perform their computations on a single Raspberry Pi board. We have implemented a custom YOLO to run on a less-resourced environment. In a High resource enabled environment like Google Colab Cloud. Also, This custom YOLO implementation can be run on a single raspberry Pi board but the time for execution is comparatively high. The custom implementation was focused on the core CNN computation of YOLO. In the Cloud environment, the GPU and CPU which have capabilities up to 12 GB YOLO perform in separate efficiency. We ran our custom optimized YOLO algorithm in the High computationally capable CPU and GPU.

$$\frac{GPU\,Execution\,time(s)}{CPU\,Execution\,time(s)} = \frac{270.51}{29.43} = 9.19$$

Our YOLO Object detection algorithm performs nearly 9 times better than in GPU enabled cloud than Raspberry Pi. At the next instance, we implemented the optimized algorithm in distributed cluster of Raspberry Pi nodes. Here we distributed the computation to a cluster of two raspberry pi nodes. When the execution time in the Colab cloud is compared with execution time is any raspberry Pi implementation Colab cloud performs well. Because of the availability of high computational resources. But the goal is to perform the complex CNN operations at the edge. So we have successfully deployed our YOLO implementation which consists of the optimized CNN and performed the tasks with the very constrained and resource-limited environment of the Raspberry Pi.

$$\frac{Resource\ availability\ in\ Colab\ Cloud - GPU(RAM)}{Resource\ availability\ of\ single\ Raspberry\ Pi(RAM)} = \frac{12\ GB}{512\ MB} = 23.43$$

The reason YOLO performs better at the Colab Cloud is that Colab Loud provides around 24 times better Computational resources compared to a single edge Raspberry Pi. But our implementation of YOLO with the optimized CNN performed within around 270.51 seconds even in the very resource-constrained environment. Further, the optimized YOLO algorithm was distributed to two parallel Raspberry Pis.

$$\frac{Parallel\ implementation\ execution\ time(ms)}{Single\ Raspberry\ Pi\ execution\ time(ms)} = \frac{89}{270.51} = 0.329$$

With the parallel implementation, the Execution time is reduced nearly by a factor of two. This parallel implementation which distributes the computation among the Edge nodes performs better even with the very constrained and limited resources available at the edge devices. For further optimization, we used multi-threads to perform computation utilizing each raspberry pi cores. To find the optimal number of threads to use we tabulated the number of threads vs total execution time in Fig 5.1. The optimal number of threads per raspberry pi node was 10. With the optimal number of threads applied with the distributed computation using Map reduce we achieved the total execution time to 41.74 seconds.

$$\frac{\textit{Parallel implementation with multi threads execution time}(ms)}{\textit{Colab Cloud GPU enabled execution time}(ms)} = \frac{41.74}{29.43} = 1.41$$

We could achieve nearly the same performance at edge when we apply distributed computing together with optimization using multi threads. Furthermore we applied vectorization for CNN which also improves the performance. Then to combine both of these optimization techniques we used multi threads upon vectorized CNN. Here to enable multi core resource utilization on raspberry pi when using vectorization, we used the OpenBLAS, an optimized Basic Linear Algebra Subprograms library. From Fig Fig 5.2 we can see that for any given input size the execution time for vectorization with OpenBLAS is less than multi threading with PyMP implementation. Therefore we used vectorization with OpenBLAS for the core CNN computation of YOLO to enable real time computations.

Input channel size	Vectorized(ms)	Non vectorized(ms)
1	6.81	6760.97
2	8.38	6933.99
4	11.14	6825.21
8	15.51	6956.65
16	25.77	7228.15
32	45.71	7438.52
64	83.75	9521.63
128	168.72	14287.22

Table 5.1 Comparison of vectorized convolution with non vectorized implementation.

As previously explained we use the policy defined by the equation (3.3) to make the offloading decisions. For that we needed to calculate the execution time on the raspberry pis. We used GFLOPs based approach with the number of floating point operations to calculate the execution time. Figure 5.3 shows the execution time with different input matrices. Estimated times are shown in the red columns, when compare with the actual execution time our GFLOPs based method was able to estimate the execution time close to the actual execution time.

Table 5.1 shows the vectorized convolution execution time with the non vectorized implementation.For the input matrix (64,64, channels) with the kernel shape (3,3,channels,256). There is a high improvement with the vectorization. When consider the channel size of 128,

$$\frac{Non\ vectorized\ execution\ time(ms)}{Vectorized\ execution\ time(ms)} = \frac{14287.22}{168.72} = 84.68$$

Vectorization improved the convolution operation by a factor of 85.

Chapter 6

Conclusion

The aim of the project is to enable real time processing at the edge with limited resources.So with combining above mentioned techniques, real time processing at the edge is achievable. In this study we have developed a distributed CNN using map reduce which can be used to implement YOLO.Furthermore, we propose a novel approach where end devices consisting limited resources can train and generate real time decision in distributed manner, where their computations are distributed among other multiple nodes or offload the computation to the upper layer when the resources run out. We have distributed the CNN over multiple Raspberry Pis.The performance of the CNN has been measured under different conditions and platforms. Based on the results it can be concluded that by distributing CNN over multiple nodes the computation latency can be reduced.Then we optimized our implementation using vectorization and multi threads together so the execution time is reduced further enabling the real time computations using limited resources.

Offloading decisions were taken using a policy where the computation time and network transmissions delays were considered. Also we considered the memory usage for the computation which also affected the offloading decisions. We also apply federated learning to train a global model where each device is able to train a local model which addresses the privacy issues. These techniques provide the path to achieve better results using limited resource devices.

References

[1] J. S. D. R. G. A. F. Redmon, "(YOLO) You Only Look Once," *Cvpr*, 2016.

[2] "ultralytics/yolov3: YOLOv3 in PyTorch >ONNX >CoreML >TFLite."

[3] M. Merenda, C. Porcaro, and D. Iero, "Edge Machine Learning for AI-Enabled IoT Devices: A Review," *Sensors*, vol. 20, p. 2533, apr 2020.

[4] Y. Zhao, M. Li, L. Lai, N. Suda, D. Civin, and V. Chandra, "Federated Learning with Non-IID Data," 2018.

[5] W. Yu, F. Liang, X. He, W. G. Hatcher, C. Lu, J. Lin, and X. Yang, "A Survey on the Edge Computing for the Internet of Things," *IEEE Access*, vol. 6, pp. 6900–6919, 2017.

[6] G. I. Klas, "Fog Computing and Mobile Edge Cloud Gain Momentum Open Fog Consortium , ETSI MEC and Cloudlets," pp. 1–13, 2015.

[7] J. An, W. Li, F. L. Gall, E. Kovac, J. Kim, T. Taleb, and J. Song, "EiF: Toward an Elastic IoT Fog Framework for AI Services," *IEEE Communications Magazine*, vol. 57, no. 5, pp. 28–33, 2019.

[8] R. Mahmud, R. Kotagiri, and R. Buyya, "Fog Computing: A Taxonomy, Survey and Future Directions," *Internet of Things*, vol. 0, pp. 103–130, nov 2016.

[9] V. Mnih, A. Puigdomènech Badia, M. Mirza, T. Harley, T. P. Lillicrap, D. Silver, and K. Kavukcuoglu, "Asynchronous Methods for Deep Reinforcement Learning Volodymyr," *International Conference on Machine Learning*, vol. 48, 2013.

[10] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, "Deep reinforcement learning that matters," *32nd AAAI Conference on Artificial Intelligence, AAAI 2018*, pp. 3207–3214, 2018.

[11] T. Hester, T. Schaul, A. Sendonaris, M. Vecerik, B. Piot, I. Osband, O. Pietquin, D. Horgan, G. Dulac-Arnold, M. Lanctot, J. Quan, J. Agapiou, J. Z. Leibo, and A. Gruslys, “Deep q-learning from demonstrations,” *32nd AAAI Conference on Artificial Intelligence, AAAI 2018*, pp. 3223–3230, 2018.

[12] X. Wang, Y. Han, C. Wang, Q. Zhao, X. Chen, and M. Chen, “In-edge AI: Intelligentizing mobile edge computing, caching and communication by federated learning,” *IEEE Network*, vol. 33, pp. 156–165, sep 2019.

[13] Q. Yang, Y. Liu, T. Chen, and Y. Tong, “Federated machine learning: Concept and applications,” *ACM Transactions on Intelligent Systems and Technology*, vol. 10, no. 2, pp. 1–19, 2019.

[14] L. U. Khan, N. H. Tran, S. R. Pandey, W. Saad, Z. Han, M. N. H. Nguyen, and C. S. Hong, “Federated Learning for Edge Networks: Resource Optimization and Incentive Mechanism,” pp. 1–7, 2019.

[15] J. Konečný, H. B. McMahan, D. Ramage, and P. Richtárik, “Federated Optimization: Distributed Machine Learning for On-Device Intelligence,” pp. 1–38, 2016.

[16] M. Chen, H. V. Poor, W. Saad, and S. Cui, “Wireless Communications for Collaborative Federated Learning in the Internet of Things,” pp. 1–17, 2020.

[17] M. Villari, M. Fazio, S. Dustdar, O. Rana, and R. Ranjan, “Osmotic Computing: A New Paradigm for Edge/Cloud Integration,” *IEEE Cloud Computing*, vol. 3, pp. 76–83, nov 2016.

[18] L. Lovén, T. Leppänen, E. Peltonen, J. Partala, E. Harjula, P. Porambage, M. Ylianttila, and J. Riekki, “EdgeAI: A Vision for Distributed, Edge-native Artificial Intelligence in Future 6G Networks,” tech. rep.

[19] Y. Hao, Y. Miao, L. Hu, M. S. Hossain, G. Muhammad, and S. U. Amin, “Smart-Edge-CoCaCo: AI-Enabled Smart Edge with Joint Computation, Caching, and Communication in Heterogeneous IoT,” *IEEE Network*, vol. 33, pp. 58–64, mar 2019.

[20] P. Mach and Z. Becvar, “Mobile Edge Computing: A Survey on Architecture and Computation Offloading,” *IEEE Communications Surveys and Tutorials*, vol. 19, no. 3, pp. 1628–1656, 2017.

[21] Y. Liu, J. Yang, Y. Huang, L. Xu, S. Li, and M. Qi, "MapReduce Based Parallel Neural Networks in Enabling Large Scale Machine Learning," *Computational Intelligence and Neuroscience*, vol. 2015, 2015.

[22] M. Chen, W. Wang, S. Dong, and X. Zhou, "Video vehicle detection and recognition based on mapreduce and convolutional neural network," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10942 LNCS, pp. 552–562, Springer Verlag, 2018.

[23] S. B. Calo, M. Touna, D. C. Verma, and A. Cullen, "Edge Computing Architecture for applying AI to IoT," tech. rep.

[24] Z. Zou, Y. Jin, P. Nevalainen, Y. Huan, J. Heikkonen, and T. Westerlund, "Edge and Fog Computing Enabled AI for IoT-An Overview," tech. rep.

[25] H. Brendan McMahan, E. Moore, D. Ramage, S. Hampson, and B. Agüera y Arcas, "Communication-efficient learning of deep networks from decentralized data," *Proceedings of the 20th International Conference on Artificial Intelligence and Statistics, AISTATS 2017*, vol. 54, 2017.

[26] A. Frickenstein, M. Rohit Vemparala, C. Unger, F. Ayar, and W. Stechele, "Dsc: Dense-sparse convolution for vectorized inference of convolutional neural networks," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, pp. 0–0, 2019.

[27] A. Lavin and S. Gray, "Fast algorithms for convolutional neural networks," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4013–4021, 2016.

[28] X. Liu, J. Pool, S. Han, and W. J. Dally, "Efficient sparse-winograd convolutional neural networks," *arXiv preprint arXiv:1802.06367*, 2018.

[29] J. Ren and L. Xu, "On vectorization of deep convolutional neural networks for vision tasks," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 29, 2015.

[30] J. Kang and D.-S. Eom, "Offloading and transmission strategies for iot edge devices and networks," *Sensors*, vol. 19, no. 4, p. 835, 2019.

[31] A. E. Eshratifar and M. Pedram, "Energy and performance efficient computation offloading for deep neural networks in a mobile cloud computing environment," in *Proceedings of the 2018 on Great Lakes Symposium on VLSI*, pp. 111–116, 2018.

[32] M. Min, L. Xiao, Y. Chen, P. Cheng, D. Wu, and W. Zhuang, "Learning-based computation offloading for iot devices with energy harvesting," *IEEE Transactions on Vehicular Technology*, vol. 68, no. 2, pp. 1930–1941, 2019.

[33] Y. Nimmagadda, K. Kumar, Y.-H. Lu, and C. G. Lee, "Real-time moving object recognition and tracking using computation offloading," in *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2449–2455, IEEE, 2010.

[34] K. Kumar, J. Liu, Y.-H. Lu, and B. Bhargava, "A survey of computation offloading for mobile systems," *Mobile networks and Applications*, vol. 18, no. 1, pp. 129–140, 2013.

[35] H. Qi, E. R. Sparks, and A. Talwalkar, "Paleo: A performance model for deep neural networks.," in *ICLR (Poster)*, 2017.

[36] X. Masip-Bruin, E. Marin-Tordera, A. Jukan, and G. J. Ren, "Managing resources continuity from the edge to the cloud: Architecture and performance," *Future Generation Computer Systems*, vol. 79, pp. 777–785, feb 2018.

[37] K. Bilal, O. Khalid, A. Erbad, and S. U. Khan, "Potentials, trends, and prospects in edge technologies: Fog, cloudlet, mobile edge, and micro data centers," *Computer Networks*, vol. 130, pp. 94–120, jan 2018.

[38] S. Taherizadeh, V. Stankovski, and M. Grobelnik, "A capillary computing architecture for dynamic internet of things: Orchestration of microservices from edge devices to fog and cloud providers," *Sensors (Switzerland)*, vol. 18, sep 2018.

[39] V. Mazzia, A. Khaliq, F. Salvetti, and M. Chiaberge, "Real-time apple detection system using embedded systems with hardware accelerators: An edge AI application," *IEEE Access*, vol. 8, pp. 9102–9114, 2020.

[40] K. Jo, J. Im, J. Kim, and D. S. Kim, "A real-Time multi-class multi-object tracker using YOLOv2," *Proceedings of the 2017 IEEE International Conference on Signal and Image Processing Applications, ICSIPA 2017*, pp. 507–511, 2017.

[41] A. Krizhevsky, I. Sutskever, and G. E. Hinton, “ImageNet Classification with Deep Convolutional Neural Networks,” tech. rep.

[42] R. Huang, J. Pedoeem, and C. Chen, “YOLO-LITE: A Real-Time Object Detection Algorithm Optimized for Non-GPU Computers,” *Proceedings - 2018 IEEE International Conference on Big Data, Big Data 2018*, pp. 2503–2510, 2019.

[43] J. Dean and S. Ghemawat, “MapReduce: Simplified data processing on large clusters,” 2004.

[44] N. Kalchbrenner, E. Grefenstette, and P. Blunsom, “A convolutional neural network for modelling sentences,” *52nd Annual Meeting of the Association for Computational Linguistics, ACL 2014 - Proceedings of the Conference*, vol. 1, pp. 655–665, 2014.

[45] M. E. Paoletti, J. M. Haut, J. Plaza, and A. Plaza, “A new deep convolutional neural network for fast hyperspectral image classification,” *ISPRS Journal of Photogrammetry and Remote Sensing*, vol. 145, pp. 120–147, 2018.