Soft Computing (2021) 25:10103–10115
https://doi.org/10.1007/s00500-021-05891-2

DATA ANALYTICS AND MACHINE LEARNING

Deploying a smart queuing system on edge with Intel OpenVINO toolkit

Rishit Dagli[1] · Süleyman Eken[2]

Accepted: 17 May 2021 / Published online: 28 June 2021

Abstract
Recent increases in computational power and the development of specialized architecture led to the possibility to perform machine learning, especially inference, on the edge. OpenVINO is a toolkit based on convolutional neural networks that facilitates fast-track development of computer vision algorithms and deep learning neural networks into vision applications, and enables their easy heterogeneous execution across hardware platforms. A smart queue management can be the key to the success of any sector. In this paper, we focus on edge deployments to make the smart queuing system (SQS) accessible by all also providing ability to run it on cheap devices. This gives it the ability to run the queuing system deep learning algorithms on pre-existing computers which a retail store, public transportation facility or a factory may already possess, thus considerably reducing the cost of deployment of such a system. SQS demonstrates how to create a video AI solution on the edge. We validate our results by testing it on multiple edge devices, namely CPU, integrated edge graphic processing unit (iGPU), vision processing unit (VPU) and field-programmable gate arrays (FPGAs). Experimental results show that deploying a SQS on edge is very promising.

Keywords Smart queuing system · Edge computing · Edge AI · Soft computing · Optimization · Intel OpenVINO

1 Introduction

We can define a queue as an arrangement of people or vehicles waiting in line for their turn to get a service or move forward in an activity, while queuing is the act of taking place in such an arrangement (Lee 2019). In daily activities, queuing always occurred whenever people have interest in mutual service at a same particular period and there are limited capabilities to satisfy or provide service to all the interested individual at once, for example a queue of people at ticket windows or vehicles waiting in line at the toll. The reason for queuing is to accomplish or get the service intended in a fair and organized manner. Queue management systems (QMSs) are the go-to solution for handling queues, allowing for easy management and a streamlined experience while reducing wait times and increasing efficiency.

Often in retail stores or transportation queues, people end up wasting a lot of time, due to improper or manual queue management. People can lose time in transportation because of unwanted waiting or waiting long behind a billing counter. Retail stores often having a lot of customers make it further difficult to efficiently manage queues in these scenarios increasing the wait time drastically for a customer creating a bad experience. The queue management ideas could also be helpful in transportation scenarios and manage queues. This would help reduce waiting times in public transportation too.

Apart from the above-mentioned scenarios, the ideas of using queue management could particularly be also helpful in allotting workers efficiently in a factory environment. This could help to ensure more efficient manufacturing in factories. Queue management strategies would further be helpful in many such scenarios (see Appendix A for use case requirements).

The edge system we propose also reduces the power consumption by a great amount as it does not have to send the video streaming data to a cloud-based server greatly saving on the power costs. Since the proposed system no longer

✉ Süleyman Eken
suleyman.eken@kocaeli.edu.tr

Rishit Dagli
rishit.dagli@gmail.com

1 Thakur International School Mumbai, Mumbai, India

2 Department of Information Systems Engineering, Kocaeli University, Izmit, Turkey

requires to send or receive data from a server a cloud-based system, we further make this system easily accessible and implementable as all the processing is done on the device itself. This also saves costs of maintaining a server and ensuring up-time. The proposed system performing the computation on the device itself also provides a lower latency as the algorithms are now being run on the device itself (Zhang 2017). This is particularly helpful on a retail or transportation scenario where low latency in the application would be required.

Thus, we propose this system which is easily accessible and implementable, requires very low deployment costs, consumes lower power, does not have a huge impact of network and also gives a lower latency. Further, this SQS can also run on existing hardware with little or no changes. However, to make the model updates to facilitate better inferences from real-time data, the system does use network. We do this not by sending the complete video stream or data but instead uploading a locally trained small piece of the model if and when network becomes available. We do so to ensure privacy by not sending the video stream itself but by sending a transformed version of it, i.e., a small sized locally trained model. This is then sent to a managed server which uses this to improve on or update the model making it better at performing inferences. This considerably reduces the impact on network too. This updated model is then sent back to the edge devices if network is available and the latest version of model is deployed (Yang 2019).

The remainder of this article is organized as follows: In Sect. 2, related works are given. In Sect. 3, proposed SQS on edge is presented. In Sect. 4, we compare deep networks for smart queuing of different videos and provide performance analysis of various inference implementations. The last section concludes the article and gives future works.

2 Related works

Nowadays, companies rely a lot on computer vision, especially for established use cases such as detection, classification, and recognition with video surveillance being one of the applications. Companies working on developing these systems look for easy-to-use and deploy frameworks to facilitate the transition to a final product they can offer their clients. One such solution is the Intel OpenVINO Toolkit (https://software.intel.com/en-us/OpenVINO-toolkit) which leverages image processing, machine learning and deep learning to solve computer vision problems while allowing cross-platform deployment. It comes with a variety of built-in and optimized algorithms for the wide range of Intel hardware (CPUs, FPGAs, Movidius Neural Compute Stick, etc.). One of its key strengths is allowing developers to use the same application programming interface and execute their code heterogeneously on Intel accelerators. Deploying SQS on edge with Intel OpenVINO Toolkit has important use cases requiring fast deep learning model inference. So, we firstly give queue and related terms and technologies and then present real-time object detector for deployment on edge in next paragraphs.

There are two main parameters of queue management, which are the number of channels (or servers) and the number of phases of service. Each parameter can take two values: single (one) or multi (several). Different combinations of channels and phases give four distinct types of queue management: (1) single-channel, single-phase (e.g., an automated car wash), (2) single-channel, multi-phase (e.g., retail banking), (3) multi-channel, single-phase (e.g., airline ticket counter), and (4) multi-channel, multi-phase (e.g., a laundromat with several washers and dryers) (Adams 2012). There are also different types of queues, ranging from structured queues with fixed and predetermined locations, such as at supermarket checkouts, to unstructured ones with no specific rules for position arrangement. Kiosk-based queuing systems where clients use a terminal to select a service upon arrival usually appear in environments where medical and banking services are offered, as well as in telecommunication companies and government institutions. To eliminate or at least reduce waiting time and efficiency, modern companies also implement mobile-based queuing systems, and customers can use their devices to check the current status of the queue and join it at their convenience only showing up when it is their turn (Titarmare and Yerlekar 2018). Each has its own advantages and drawbacks. Moreover, to manage a queue, we need to understand the queue discipline (the order of servicing: first in–first out (FIFO), last in–first out (LIFO), service in random order (SIRO), and priority selection (Dada and Thiesse 2008).

QMSs are critical components in any sector of business. It is important for the service provider to provide efficient queuing system to maintain high customer satisfaction. Besides, QMS can help record, predict, and calculate the statistic of the queuing pattern such as customers' arrival rate, queuing behavioral over time, and average waiting time which can help in decision making. Queuing systems, even smart ones, whereby a client gets a digital ticket from a machine or online and waits for a turn, face many limitations in terms of creating an improved user experience (Ghazal et al. 2015). There are also many reasons to use a QMS. A proper queue management strategy shortens on-site wait times and reduces walkaways. By allowing customers to use their time in a more efficient manner, a QMS builds up customer experience. By improving customer engagement, a QMS turns visitors into customers and customers into promoters. The QMS is a set of tools and sub-systems for multiple industries including banking/insurance (Ibrahim et al. 2006), healthcare (Rezaee et al. 2014), retails (Berman and Larson 2004), transportation (Schwarz and Martin 2016), manufacturing (Rüttimann and Stöckli 2020), government, and telecommunication (Giambene 2005). Ibrahim et al. (2006) employed factor analysis procedure to identify the underlying structure among the explored electronic service quality attributes. Good queue management is one of the composite dimensions of electronic service quality that are the UK customers' perceptions of their bank actual performance. Rezaee et al. (2014) presented a congestion management protocol for healthcare wireless sensor networks (HWSNs). Their protocol includes active queue management scheme to separate virtual queues on a single physical queue to store the input packets from each child node. Berman and Larson (2004) modeled facility having workers who are cross-trained to do front room and back room operations in retail sector. Switching from back room to front room is done using queues. Schwarz and Martin (2016) proposed a new discrete-time approach for the steady-state analysis of circulating vertical conveyor systems bulk service queues. Rüttimann and Stöckli (2020) distinguished the classic traditional batch and queue manufacturing for optimal scheduling sequences. In the light of these studies, it is possible to say that having good queue management procedures in place can make a real difference to any customer facing environment. COVID-19 has really highlighted the need for effective queue management.

Many computer vision applications rely heavily on object detection (OD) and object recognition (OR), which creates a need for well-optimized algorithms. However, it is common for object detection CNNs to face some hurdles while moving from development to deployment unless these difficulties are accounted for and addressed as pointed out by Kozlov and Osokin (2019). In their work, they follow up by using a Single-Shot MultiBox Object Detection (SSD) to detect vehicles and pedestrians with Intel OpenVINO Toolkit. Osokin (2018) also worked on multi-person pose estimation architecture on edge devices and optimized the popular method OpenPose (Cao et al. 2018). A new improved framework for neural networks was proposed in the work by Kozlov et al. (2020). Its success is thanks to the use of modern network compression methods, such as sparsity, quantization, and binarization. Using these techniques results in lighter models that can be run on cheap general-purpose computers. Currently, a PyTorch version of their framework neural network compression framework (NNCF) is available as a part of OpenVINO Training Extensions [1]. Targeting OpenVINO as well, Kustikova et al. implement a solution for semantic segmentation of on-road images (Kustikova et al. 2019a) and another to benchmark inference performance of deep learning models on various types of hardware (Kustikova et al. 2019b) (CPUs, integrated graphics and embedded devices) [2,3]. Similarly, (Kljucaric and George 2019) used Chinese character recognition to evaluate the performance of current CNN architectures and frameworks on different hardware types (FPGAs, GPUs, and CPUs). Mathew et al. (2019) as a continuation of the work on DetectNet Deep Learning Model for lung nodule detection trained on the LIDC dataset, ported it from NVIDIA to Intel OpenVINO. Castro-Zunti et al. (2020) modified a single-shot detector (SSD) and used it for license plate segmentation and recognition on an embedded system while providing plate processing time comparisons between a variety of hardware types. Regarding performance, (Gorbachev et al. 2019) suggest a new software solution for holistically analyzing model performance on various hardware. Mittal and Bhushan (2020) propose an efficient workflow for OpenVINOs, while (Jin and Finkel 2020) evaluate the performance of 14 deep learning inference models using OpenVINO. When these studies in the literature are examined, it is important to implement an image-based smart queue system. So, we propose an OpenVINO-based SQS, which accelerates the processing speed of the model and meets the real-time applications.

Contributions to the literature with the paper can be listed as follows:

- We propose an efficient and smart way to perform queue management for different use-cases. We majorly focus on deploying the SQS on an edge device.
- We use Power Proxying that can proxy a variety of network protocols, increasing the standby time of the host without compromising its active connections.
- We also propose a way to update the model making the model better as we get more and more real-life data without leveraging security or network. We provide ways with which apart of queuing, complex analysis and finding out trends for consumer patterns in a retail store could also be implemented. This would enable owners to derive insights based on consumer patterns such as where should an advertisement or the billing counter be placed for efficient management of crowd.
- We validate the above-mentioned ideas and claims by testing them on multiple edge devices with Intel's OpenVINO Toolkit to perform optimization on the model for it to be able to run on edge and deploying it to multiple hardware and edge machines easily. Further, we also profile our application and find hot-spots in it with OpenVINO Toolkit.

[1] https://github.com/opencv/openvino_training_extensions/tree/develop/pytorch_Toolkit/nncf

[2] https://github.com/itlab-vision/openvino-dl-benchmark

[3] http://hpc-education.unn.ru/dli

3 Proposed architecture

The proposed SQS demonstrates how to create a video AI solution on the edge. The system detects people in a specified area and accordingly detects the number of people in a queue. It would then also notify whether a person would need to change the queue to reduce congestion. If there are more than a certain number of people in the watched area, a warning is given as the queue is full and move to the next queue. In this paper, we use various SSD models (Liu et al. 2016) to detect people in an image using a single deep neural network. This eliminates proposal generation and subsequent pixel or feature re-sampling stage and encapsulates all computation in a single network. SSD has comparable accuracy to methods that utilize an additional object proposal step and is much faster while providing a unified framework for both training and inference.

We choose multiple models from Intel's OpenVINO Model Zoo (https://github.com/OpenVINOtoolkit/open_model_zoo) and TensorFlow Model Garden (https://github.com/tensorflow/models). We test these pre-trained models for use-cases on basis of model load time, inference time on one frame of the video and FPS (frames per second) of the inference stream. We also perform a preliminary comparison of the models based on theoretical parameters specifically billions floating-point operations (GFLOPs) and amount of memory consumption before performing analysis on practical parameters mentioned above. We majorly perform the theoretical analysis to make sure that the edge device would not run out of memory and the model should occupy disk space such that it can be used with a particular edge device.

We perform various optimizations over the source model from Intel's OpenVINO Model Zoo, TensorFlow Model Garden or transfer learned models. We do these optimizations to be able to run these models on edge devices which have limitations on processing, memory, power-consumption, network usage, and model storage space. While performing the optimizations due to the limitations of edge devices on which the SQS would majorly be deployed, we majorly care about model load time, per frame inference time and model size while performing the optimizations. We describe the process we use to perform the optimization in greater detail in later parts of this paper.

We then make further comparisons between these models and make experiments for various model to find a suitable configuration with the OpenVINO DL Workbench to find a model with lower size and acceptable throughput. We also make use of VTune profiler to help us find and solve any hotspots in the application that may be faced during run-time. This helps us improve the overall performance and throughput of the application.

To be able to deploy these models on multiple devices, we run them on the inference engine which provides a common API to deliver inference solutions on the platform of our choice: CPUs, iGPUs, FPGAs or VPUs. To use these, we first convert these models into an acceptable IR (intermediate representation) format. It also enables us to use hardware-specific APIs to further improve the performance of the models.

We then use Intel's DevCloud to make and test real-time deployments for multiple hardware. We also test for the primary use-cases and scenarios where the SQS may be used, namely in retail stores, manufacturing factories and in public transports. We do so on multiple hardware, CPUs, GPUs (graphic processing unit), FPGAs (field-programmable gate arrays) and VPUs (vision processing unit) to find out the best possible hardware and compare the different hardware for each of these use cases. With the Intel DevCloud, we can make these deployments without using a physical device by submitting a job.

Figure 1 shows the main steps of the SQS. Figure 2 illustrates the user workflow for code development, job submission and viewing results. It mainly includes three sub-steps: (1) person detection, (2) submission of the job to multiple devices on the DevCloud, and (3) inferencing video output.

To demonstrate the following ideas, we perform inference of some open-source video clips pertaining to the primary use-case mentioned earlier in this paper, and we also run these on multiple devices. The results of these are in the next section.

4 Experimental results

4.1 Experimental setup

We first optimize the model so it can be run on edge devices. This is necessary to reduce cost, latency, especially for deployment on hardware with limited capabilities, be it processing, memory, power consumption, networking, or storage space. The optimization also takes into account more powerful and specialized hardware with accelerators (https://www.tensorflow.org/model_optimization). Hardware and software requirements are as follows:

– *Hardware:* This project makes the use of Intel DevCloud to test on CPU, GPU, FPGA and VPU. So, no specific hardware is required.
– *Software:* Intel® Distribution of OpenVINO^{TM} Toolkit 2019 R3 release.
  Python > 3.5

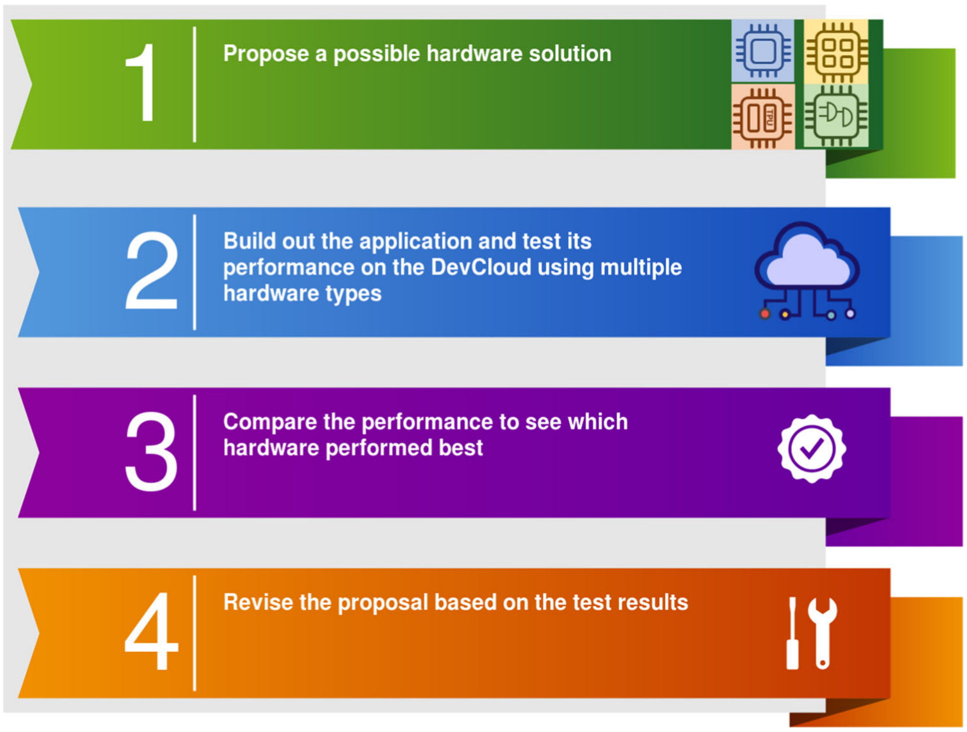

1
Propose a possible hardware solution
2
Build out the application and test its performance on the DevCloud using multiple hardware types
3
Compare the performance to see which hardware performed best
4
Revise the proposal based on the test results

Fig. 1 Main steps of the SQS

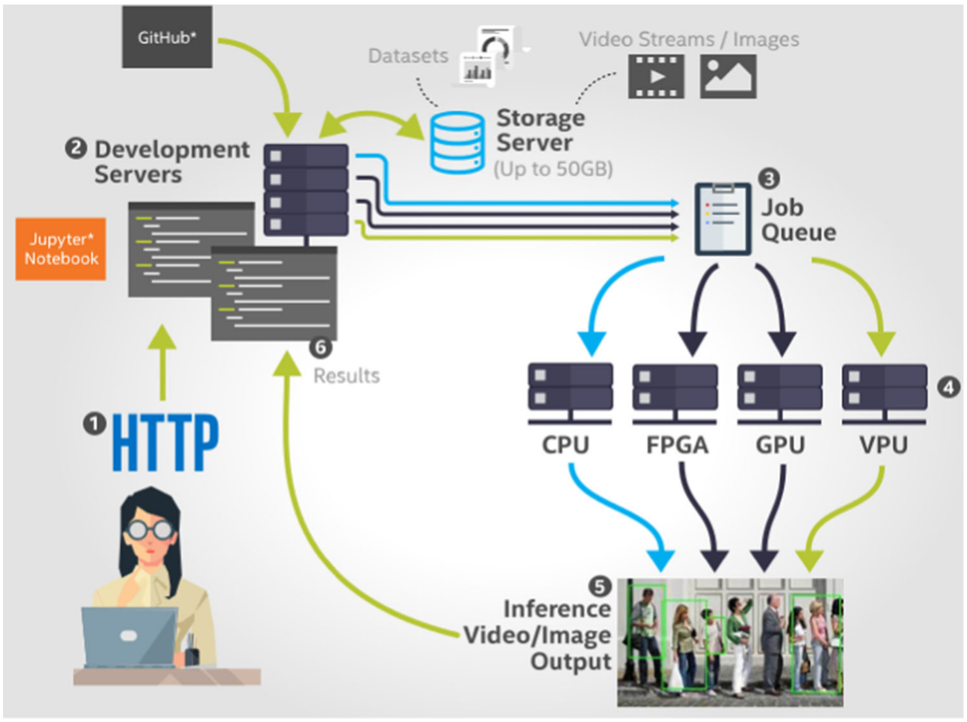


Fig. 2 SQS run life-cycle

4.1.1 Freezing models

We particularly need to employ model freezing strategies while using TensorFlow pre-trained models or on which we perform transfer learning. While using TensorFlow models, all trainable parameters are represented as variables in the graph. To get the best possible model, the workflow demands the freezing of these variables between each layer, whose values usually change after each step of the training process. This is achievable through the use of TensorFlow's functions when using transfer learning from a pre-trained model. We start by defining some necessary classes:

- *NodeDef* to hold one single value or operation from the network.
- *Checkpoint* to take a periodic snapshot of the variables as the model trains.
- *GraphDef* to define the whole executable graph including all the NodeDefs within it. During execution, the values of any node variables are accessible from the checkpoint files (Takawale and Thakur 2018).
- *Saver* to save the variables in a checkpoint file. They are then modified to be constants using the 'freeze_graph' class, which gives us a new GraphDef with the constants in return.

4.1.2 Optimizing models

As mentioned earlier, we first optimize the model so that it can be deployed on edge devices. In case of TensorFlow models, we first freeze the models too. We use multiple techniques to do so, and the majorly used ones are:

- *Operation Fusing:* Many convolutional neural networks (CNNs) include layers that can be represented as a sequence of linear operations such as addition, multiplication or matrix multiplications. These layers can be fused into previous convolution or fully connected dense layers. However, this technique is not applicable when a convolutional layer comes after a add operation due to paddings.
- *Quantized Distillation:* By improving the training process using distilling (Hinton et al. 2015). This is achievable in the form of a distillation loss from a teacher network and inserted into a smaller student network, and we quantize the weights of the latter to a limited set of levels (Polino et al. 2018). The result is a significantly smaller model size.
- *Mixed precision Quantizing:* Here, we quantize different layers with different bit-widths. For mixed precision quantization, we assume that we have the flexibility to choose different precisions for different layers of a network. Mixed precision computation is widely supported by hardware platforms such as CPUs, FPGAs, and dedicated accelerators. Thus, we choose precision for each layer; for most of the models tested, we use differentiable neural architecture search (Wu et al. 2018). This allows us to reduce the model size and also receive lower latency predictions.

4.1.3 Converting models to an intermediate representation

To use these models with Intel's OpenVINO Toolkit, they first need to be converted to an intermediate representation (IR) which is OpenVINO toolkit's format of graph representation and its own operation set. A graph here is represented with two files: an XML file and a binary file (https://docs.openvinotoolkit.org/latest/openvino_docs_MO_DG_IR_and_opsets.html#intermediate_representation_used_in_openvino). The XML file describes a network topology using <layer> tag for an operation node and <edge> tag for a data-flow connection. Each operation can also have a fixed number of attributes. Constant values like convolutional weights or biases are stored in the accompanying binary file. This intermediate representation can be used to run the model on edge with OpenVINO's Deep Learning Inference Engine (https://docs.openvinotoolkit.org/2020.4/openvino_docs_IE_DG_Deep_Learning_Inference_Engine_DevGuide.html) which is a unified API to allow high-performance inference on many hardware types (CPUs, GPUs, FPGAs, Neural Compute Stick, Raspberry Pi etc.) making thee deployment procedure a lot easier. We can also make modifications or changes in the workflow for a specific device type with the Deep Learning Inference Engine.

After configuring the OpenVINO Model Optimizer for a particular framework, it provides with some easy-to-use framework-specific Python scripts which one can directly use to convert a model to intermediate representation (IR). The OpenVINO model optimizer performs some basic model optimization and converts it to intermediate representation. We also make modifications and extend the model optimizer with custom primitives to support more customization as mentioned earlier.

We also tested for these use-cases with some pre-trained models from the Intel OpenVINO Model Zoo. The Intel OpenVINO Model Zoo also serves as a source of finding open-source pre-trained models converted in the IR format. One can directly use models from the Model Zoo and use them with Intel OpenVINO.

4.1.4 Post training optimization

We then use post-training quantization to convert the models into a more hardware-friendly representation by applying specific methods that do not require re-training of the model. Before quantization, we also align ranges of output activation of convolutional layers in order to reduce the quantization error. To perform post-training quantization, we use the following methods.

- *8-bit quantization:* We use this vanilla quantization method that automatically inserts FakeQuantize operations into the model graph based on the specified target hardware and initializes them from the calibration dataset. We opportunistically replace 32-bit floating point (FP32) computations with 8-bit integers (INT8) and transform the FP32 computational graph (Bhandare et al. 2019). These seem to greatly improve the performance of the model. We then also adjust biases of convolutional and fully connected layers based on the quantization error of the layer.
- *Parzen Estimator:* We then use a Tree-structured Parzen Estimator (TPE) Approach, a sequential model-based optimization method to perform global optimization of the post training quantization parameters. We do this by first defining a hyper-parameter search space and create an objective function which takes in hyper-parameters and outputs a score that it needs to maximize; in this case the score we used was the appropriate queue detection. We then get some observations which are randomly selected from the set of hyper-parameters. We then make two quantiles x_1 and x_2 where we divide all the good scores into one quantile. We then model these using the Parzen estimator which is the average of the kernels centered on this data. We then draw hyperparameters and evaluate them in terms of both quantile densities. These hyperparameters are then evaluated back on the objective function (Bergstra et al. 2011).
- *Per-channel quantization:* We also make use of per-channel quantization for convolutional and fully connected layers. Our experiments show that per-channel quantization is needed to compensate for the accuracy drop resulting from quantization, asymmetric per-layer quantization seems to work best for this, while at the same time, it works as a good baseline for post-training quantization of weights and activations. This results in an accuracy that is very close to what floating-point offers for all networks (Krishnamoorthi 2018). We perform asymmetric quantization according to the following formula:
  $input\ low' = min(input\ low, 0)$
  $input\ high' = max(input\ high, 0)$
  So,
  $input\ high'' = \frac{ZP-levels+1}{ZP} \times input\ low'$
  $input\ low'' = \frac{ZP}{ZP-levels+1} \times input\ high'$

Thus, we use the following process for post-training quantization to speed up the model inference times drastically and reduce the model sizes by a great amount. This particularly helps in ensuring that retail stores or factories can use this with the devices already in use by them. Furthermore, as we will mention we have also tried these algorithms on specialized embedded systems too.

4.1.5 Comparing models with deep learning workbench

The OpenVINO Deep Learning Workbench provides interactive user interface (see Fig. 3), simplified process of 8-bit quantization, speeding up convolutional operations using the Winograd's minimal filtering algorithms, measuring accuracy of the resulting model (Gorbachev et al. 2019; Lavin and Gray 2016). We first import the OpenVINO IR model, select the target hardware available for inference. This then allows us to analyze the theoretical information about the model like GFLOPs, amount of memory consumption and the inference times.

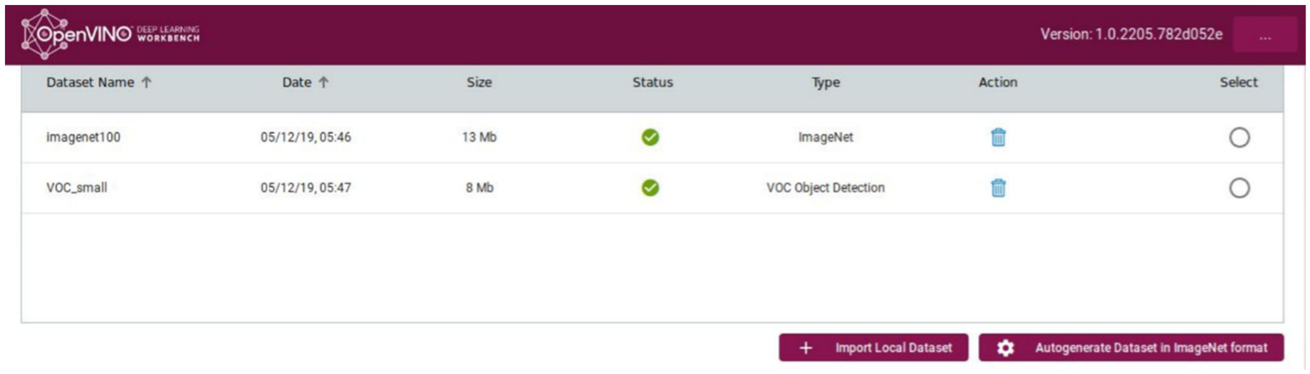
We perform various such experiments to find a suitable configuration that can provide the highest throughput with minimal latency. As mentioned earlier, we also perform some post-training quantization; the Deep Learning Workbench allows us to compare the performance of these quantized models with the original model. We compare the original and quantized model on multiple hardware, CPU, GPU, Intel Neural Compute Stick (Myriad VPU) and FPGAs too. We then select the best configuration for these models and implement them.

Thus, we compare some models: Transfer learned from existing state of the art models (specifically from MobileNet Howard et al. 2017; Sandler et al. 2018, Inception Szegedy et al. 2016 and ResNet He et al. 2016), models from the OpenVINO Model Zoo and TensorFlow Model Garden in this manner based on various theoretical parameters such as GFLOPS or memory consumption and different hardware (see Fig. 4). We also perform the comparisons between the model on practical factors like inference time, FPS and model loading time as we mention later in this paper.

4.2 Identifying hotspots in the application

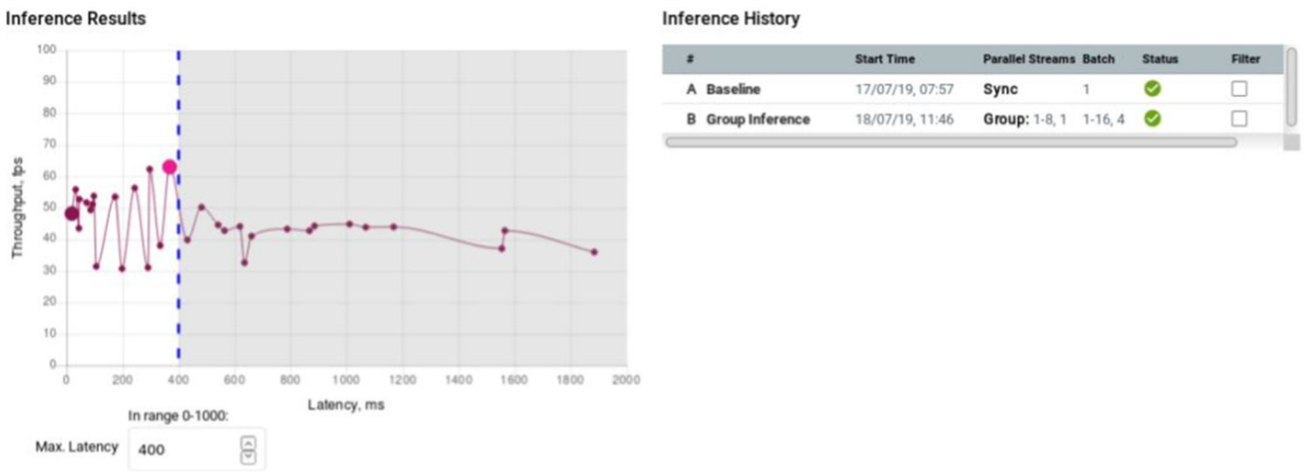
Analyzing the end application and finding areas in the application that particularly have a negative impact toward the final application; increasing latency or consuming a lot of power is mandatory to put forth a better application to the users. We also want to optimize the code as much as possible to run on edge infrastructure considering the power, memory, network and inference time limitations. We use Intel VTune Amplifier to perform these tasks[4]. Considering the

[4] https://software.intel.com/sites/default/files/1d/13/8486


OpenVINO DEEP LEARNING WORKBENCH
Version: 1.0.2205.782d052e
...
Import Local Dataset
Autogenerate Dataset in ImageNet format

Dataset Name ↑	Date ↑	Size	Status	Type	Action	Select
imagenet100	05/12/19, 05:46	13 Mb	✓	ImageNet	🗑	○
VOC_small	05/12/19, 05:47	8 Mb	✓	VOC Object Detection	🗑	○

Fig. 3 Testing multiple models with DL workbench



#		Start Time	Parallel Streams	Batch	Status	Filter
A	Baseline	17/07/19, 07:57	Sync	1	✓	☐
B	Group Inference	18/07/19, 11:46	Group: 1-8, 1	1-16, 4	✓	☐

Fig. 4 Testing models with DL workbench

full end-to-end solution, it is an important aspect to optimize the hotspots and instead move toward a holistic solution. The Intel VTune amplifier also makes it simple to do so and allows us to explore the parts of our end-to-end application and find the hot spots in it. Finding these hot spots in the application proves to be useful in optimizing the code.

It is also another important aspect to build a solution considering that a problem may not be due to the software implementation but inappropriate allocation of hardware resources or understanding how the underlying hardware works thus building an overall holistic solution giving better performance. For example, if the full use scenario is transcode an efficient Gen GPU implementation would set up concurrent encodes and decodes. On Intel processor graphics hardware, transcode is faster than running encode or decode separately. With transcode being the overhead of additional synchronization, moving raw frames to/from the CPU can be avoided (as demonstrated in the Intel VTune Whitepaper (https://software.intel.com/content/www/us/en/develop/download/white-paper-using-vtune-to-optimize-media-video-applications.html)). Using the VTune amplifier and analyzing the application from it allow us to derive insights from it. We get performance information from all compute engines in one dashboard interface and aligned on a common timeline. This allows us to see all the components working together and allow for easier use.

4.3 Testing the application for production

We need to test the application and the models for performance in various use-cases and hardware. We use a few open-sourced video clips pertaining to the major use cases mentioned for demonstration purposes. We also make use of Intel DevCloud which is a cloud service designed to help developers' prototype and experiment with computer vision applications using the Intel Distribution of OpenVINO Toolkit. It also allows us to test on latest hardware and software too and easily explore real-world workloads (https://software.intel.com/content/www/us/en/develop/tools/devcloud/edge.html).

The Intel DevCloud allows us to test on CPUs, GPUs, FPGAs and VPUs; we test on these hardware for the mentioned use cases. While testing out the models, we submit it as a job to thee Intel DevCloud which then allows us to

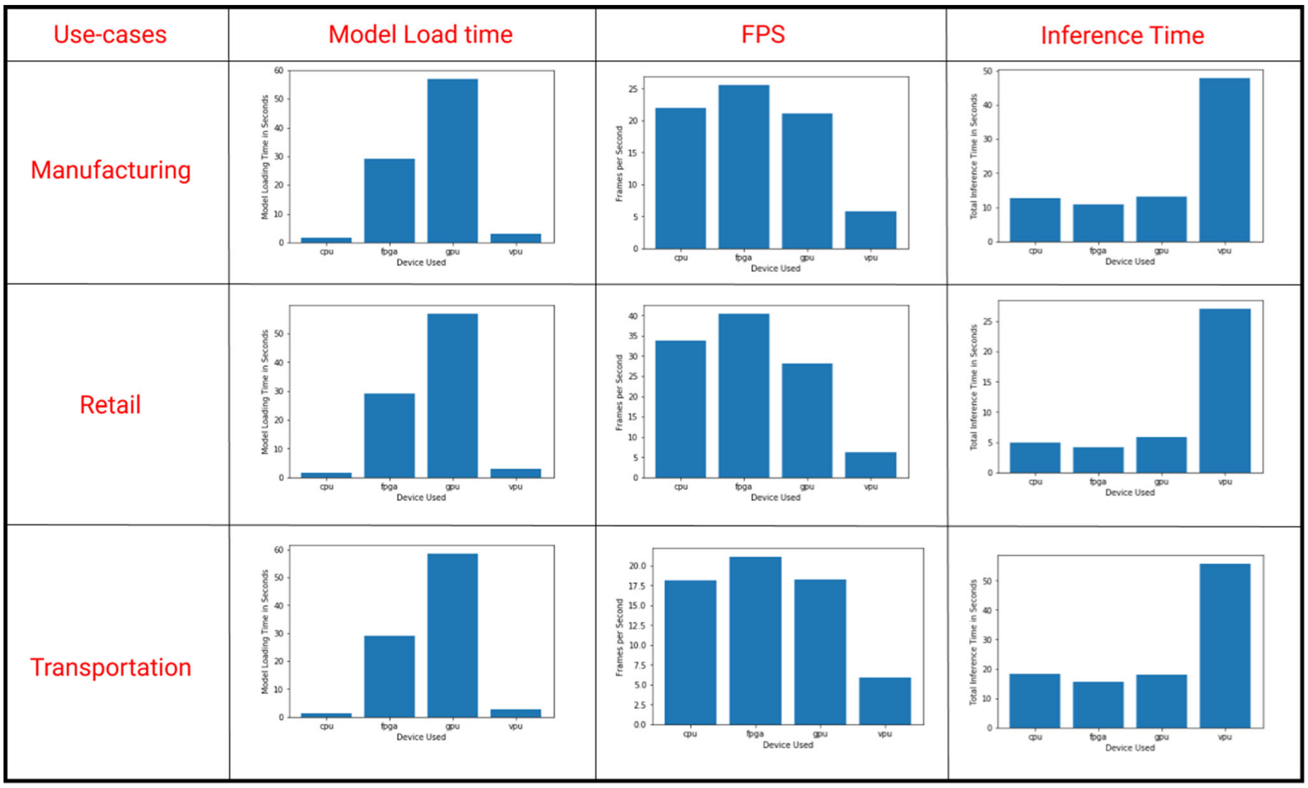


Fig. 5 Comparing models on different hardware

use the hardware and test out the models. For demonstration purposes as we show here, we test the models on:

- CPU—IEI Tank 870-Q170 edge node with an Intel Core^{TM} i5-6500TE processor
- CPU and integrated GPU (iGPU)—IEI Tank 870-Q170 edge node with an Intel Core^{TM} i5-6500TE processor with HD Graphics 530 integrated GPU
- VPU—IEI Tank 870-Q170 edge node with an Intel Core^{TM} i5-6500TE processor with Intel Neural Compute Stick 2
- FPGA—IEI Tank 870-Q170 edge node with an Intel Core^{TM} i5-6500TE processor with IEI Mustang-F100-A10 FPGA card

Figure 5 shows the comparisons among hardware with Intel DevCloud for the three use-cases mentioned earlier for demonstration purposes.

As shown in Fig. 5, the inference time graph FPGAs take the least amount of time to perform inferences. The client needs inference to be performed fast so this FPGA can do a commendable job. We can see that FPGA also gives the highest FPS. The client requires 25–30 FPS which can be addressed by an FPGA. Further they are also field programmable and have a high life which is requested by the client. Thus, FPGA would be a good choice for manufacturing use case. Figure 6 shows smart queuing output on manufacturing video.

CPU has a comparatively lower inference time than a GPU and VPU which is crucial for retail scenario. A CPU can also make inferences on a good enough FPS for a retail store. It would also help save costs and electricity bill as requested by the client. Thus, CPU would be a good choice for retail use case. Figure 7 shows smart queuing output on retail video.

VPU provides a very high inference time and inference on very low FPS. A CPU or GPU would ideally be good for this if the budget and power scenario would be ignored. Since the client requires these aspects, VPU would be a good choice for transportation use-case. Figure 8 shows smart queuing output on transportation video.

5 Conclusion

With the proposed system, smart queuing has been developed by inferencing deep models on various devices (CPUs, integrated GPUs, embedded devices). An open-source implementation of OpenVINO Toolkit-based SQS is available on GitHub. To demonstrate creating a video AI solution on the edge, we perform inference some open-source video clips pertaining to the use-cases such as retail, manufacturing, and public transportation. Through the use-case and rele-

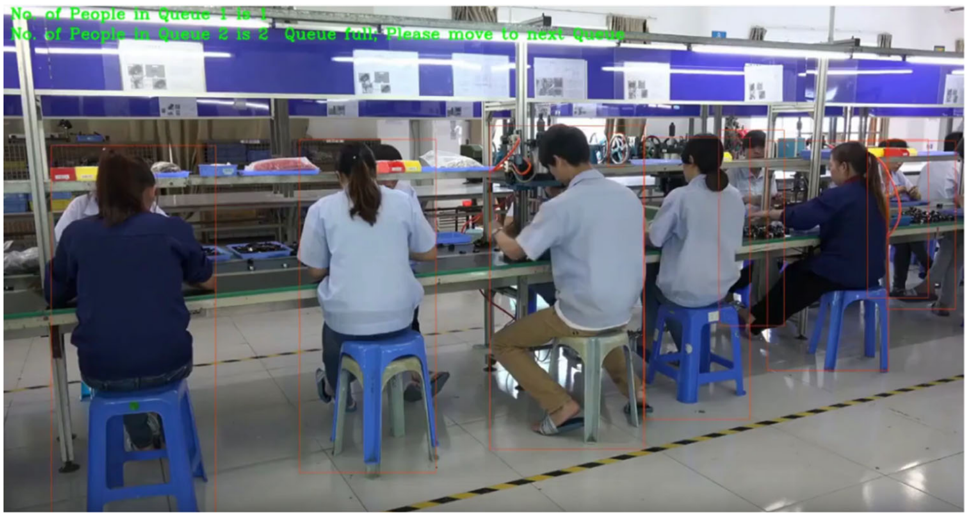

No. of People in Queue 1 is 1
No. of People in Queue 2 is 2 Queue full; Please move to next Queue

Fig. 6 Output for manufacturing use case

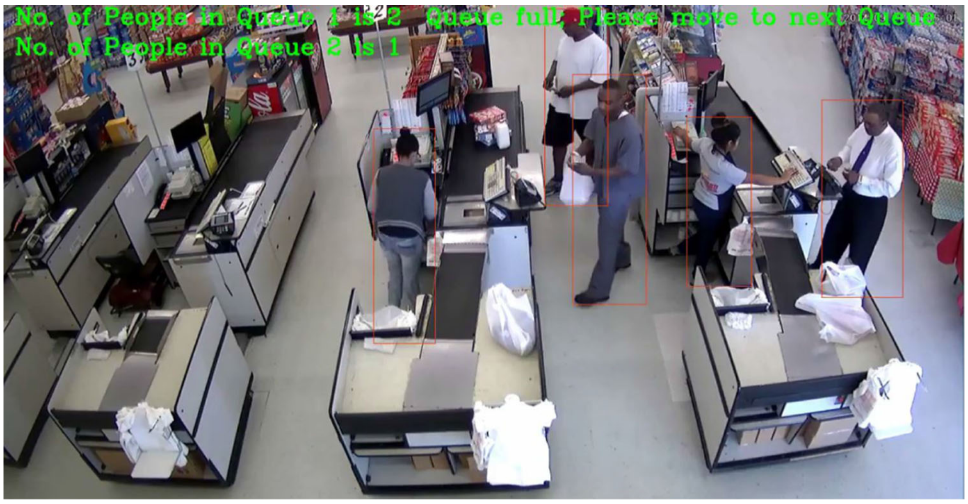

No. of People in Queue 1 is 2 Queue full; Please move to next Queue
No. of People in Queue 2 is 1

Fig. 7 Output for retail use case

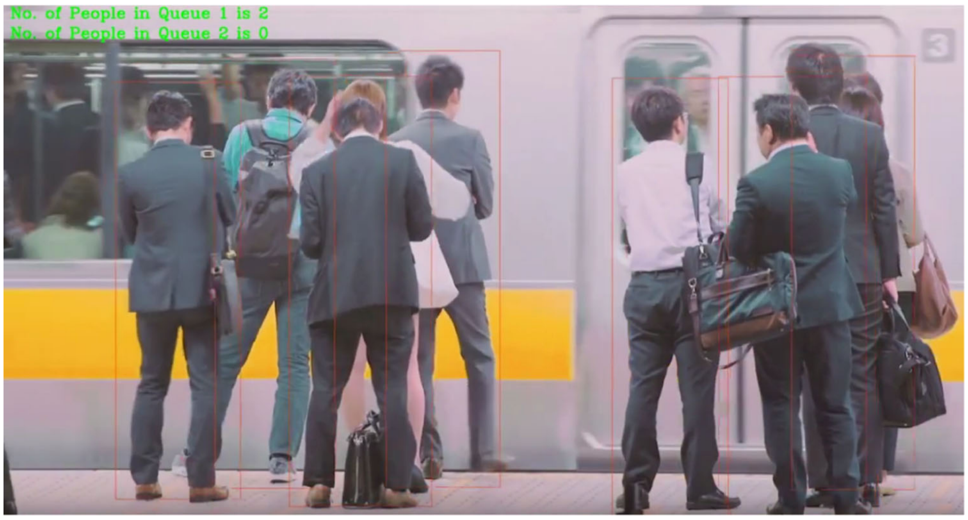


Fig. 8 Output for transportation use case

vant client requirements, we suggest a potential hardware type and explain how this hardware would satisfy each of the requirements.

In the future, we plan to increase the number of software and hardware platforms. We will analyze performance for the inference implementations based on the well-known deep learning frameworks. Further, we will compare inference performance using various deep learning tools.

Author Contributions RD conceived the presented idea and performed the computations. SE encouraged RD to investigate smart queue on edge and supervised the findings of this work. Both of authors discussed the results and contributed to the final manuscript.

Funding The author received no financial support for the research, authorship, and/or publication of this article.

Conflict of interest The author declares that he has no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Availability of data and material The codes and input videos are available at: https://github.com/Rishit-dagli/Smart-Queuing-System-On-Edge A video of the demo is available at:https://www.youtube.com/watch?v=W-DWBYhrwj0

Appendix A: Client requirements for use cases

See Tables 1, 2 and 3.

Table 1 Use case 1: manufacturing

Requirement observed	How does the chosen hardware meet this requirement?
The factory has a vision camera installed at every belt. Each camera records video at 30–35 FPS (frames per second). The client would like the image processing task to be completed five times per second	This aspect shows that latency is a major concern in this case. When an FPGA is programmed with the bit-stream required for this application, it can run the model with very high performance and give very low latency. An FPGA can run many sections of the chip in parallel, and the ability of FPGA to not go off-chip for performing inference from the model would particularly be very useful for this kind of scenario. An FPGA also does not send the output back to the CPU using PCIe bus making the inference a lot faster
The second issue the client has encountered is that a significant percentage of the semiconductor chips being packaged for shipping have flaws. These are not detected until the chips are used by clients. If these flaws could be detected prior to packaging, this would save money and improve the company's reputation	To solve this issue, the edge system would require to be able to deliver high performance. FPGAs would be a perfect fit for doing so as they provide a very high performance. Another aspect of FPGA which might help in this scenario would be its ability to be reprogrammed on the field, which can help improve inferences
To be able to detect chip flaws without slowing down the packaging process, the system would need to be able to run inference on the video stream very quickly	FPGAs can also be used as hardware accelerators speeding up the inference. The parallel processing and no need to go off chip further provide an added advantage in performing the inference faster
Additionally, because there are multiple chip designs and new designs are created regularly, the system would also need to be flexible so that it can be reprogrammed and optimized to quickly detect flaws in different chip designs.	This need of the client makes FPGA align to the required hardware. FPGAs are highly flexible; they are field programmable and can be reprogrammed as needed
The client would ideally like it to last for at least 5-10 years.	FPGAs have a very long lifespan generally 10 years and thus could be used by the client
Queue Monitoring Requirements	
Maximum number of people in the queue	2
Model precision chosen (FP32, FP16, or Int8)	FP16

Table 2 Use case 2: retail

Requirement observed	How does the chosen hardware meet this requirement?
Most of the store's checkout counters already have a modern computer, each of which has an Intel i7 core processor. Currently these processors are only used to carry out some minimal tasks that are not computationally expensive	However, the client already has a lot of CPUs which are not able to carry some computationally expensive tasks. These CPUs with some new ones for the expensive tasks could be used
The client does not have much money to invest in additional hardware	Existing CPUs with some new ones can be used by the client can be made use of to reduce the costs

Table 2 continued

Requirement observed	How does the chosen hardware meet this requirement?
The client would like to save as much as possible on his electric bill	CPUs could meet the hardware requirements and also help save on the client's electric bill
Queue Monitoring Requirements	
Maximum number of people in the queue	2–5
Model precision chosen (FP32, FP16, or Int8)	FP32

Table 3 Use case 3: transportation

Requirement observed	How does the chosen hardware meet this requirement?
The CPUs in these machines are currently being used to process and view CCTV footage for security purposes, and no significant additional processing power is available to run inference	A VPU or NCS 2 can be plugged in a USB port and has a convenient plug and play kind of performance. This particularly is favorable as no more additional processing power is available to run inference
The client's budget allows for a maximum of $300 per machine.	A VPU or a NCS 2 stick costs almost $100 opposed to the comparatively higher costs of FPGA, CPU or GPU
The client would like to save as much as possible both on hardware and future power requirements.	A VPU or NCS 2 stick is designed to run on very low power. The NCS 2 can run on just 1 W of power adhering to the clients' needs
Queue Monitoring Requirements	
Maximum number of people in the queue	7–15
Model precision chosen (FP32, FP16, or Int8)	FP16

References

Adams R (2012) Active queue management: a survey. IEEE Commun Surv Tutorials 15(3):1425–1476

Bergstra JS et al (2011) Algorithms for hyper-parameter optimization. In: Advances in neural information processing systems, pp 2546–2554

Berman O, Larson RC (2004) A queueing control model for retail services having back room operations and cross-trained workers. Comput Oper Res 31(2): 201–222

Bhandare A et al (2019) Efficient 8-bit quantization of transformer neural machine language translation model. ArXiv Preprint arXiv:1906.00532

Cao Z et al. (2018) OpenPose: realtime multi-person 2D pose estimation using Part Affinity Fields. ArXiv Preprint arXiv:1812.08008

Castro-Zunti Riel D, Yèpez J, Ko S-B(2020) License plate segmentation and recognition system using deep learning and OpenVINO. IET Intell Trans Syst 14(2): 119–126

Dada A, Thiesse F (2008) Sensor applications in the supply chain: the example of quality-based issuing of perishables. In: The internet of things. Springer, pp 140–154

Deep Learning Inference Engine Developer Guide. https://docs.openvinotoolkit.org/2020.4/openvino_docs_IE_DG_Deep_Learning_Inference_Engine_DevGuide.html

Ghazal M, Hamouda R, Ali S (2015) An iot smart queue management system with real-time queue tracking. In: 2015 Fifth international conference on e-learning (econf). IEEE, pp 257–262

Giambene G (2005) Queuing theory and telecommunications. Springer

Gorbachev Y et al. (2019) OpenVINO deep learning workbench: comprehensive analysis and tuning of neural networks inference. In: Proceedings of the IEEE international conference on computer vision workshops

He K et al (2016) Deep residual learning for image recognition. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp 770–778

Hinton G, Vinyals O, Dean J (2015) Distilling the knowledge in a neural network. ArXiv Preprint arXiv:1503.02531

Howard AG et al. (2017) Mobilenets: efficient convolutional neural networks for mobile vision applications. ArXiv Preprint arXiv:1704.04861

Ibrahim Essam E, Joseph M, Ibeh Kevin IN (2006) Customers' perception of electronic service delivery in the UK retail banking sector. Int J Bank Market

Intel DevCloud. https://software.intel.com/content/www/us/en/develop/tools/devcloud/edge.html

Intel Distribution of OpenVINO toolkit. https://software.intel.com/en-us/OpenVINO-toolkit

Intermediate Representation Used in OpenVINO. https://docs.openvinotoolkit.org/latest/openvino_docs_MO_DG_IR_and_opsets.html#intermediate_representation_used_in_openvino

Jin Z, Finkel H (2020) Analyzing deep learning model inferences for image classification using OpenVINO. In: 2020 IEEE international parallel and distributed processing symposium workshops (IPDPSW). IEEE, pp 908–911

Kljucaric L, George Alan D (2019) Deep-learning inferencing with high-performance hardware accelerators. In: 2019 IEEE high performance extreme computing conference (HPEC). IEEE, pp 1–7

Kozlov A et al. (2020) Neural network compression framework for fast model inference. ArXiv Preprint arXiv:2002.08679

Kozlov A, Osokin D (2019) Development of real-time ADAS object detector for deployment on CPU. In: Proceedings of SAI intelligent systems conference. Springer, pp 740–750

Krishnamoorthi R (2018) Quantizing deep convolutional networks for efficient inference: a whitepaper. ArXiv Preprint arXiv:1806.08342

Kustikova V et al (2019) DLI: deep learning inference benchmark. In: Russian supercomputing days. Springer, pp 542–553

Kustikova V et al (2019) Intel distribution of OpenVINO toolkit: a case study of semantic segmentation. In: International conference on analysis of images, social networks and texts. Springer, pp 11–23

Lavin A, Gray S (2016) Fast algorithms for convolutional neural networks. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp 4013–4021

Lee H (2019) The definitive guide to queue management systems. https://www.qminder.com/what-is-queue-management-system

Liu W et al. (2016) Ssd: single shot multibox detector. In: European conference on computer vision. Springer, pp 21–37

Mathew G, Sindhu RS, Suchithra VS (2019) Lung nodule detection from low dose CT scan using optimization on Intel Xeon18 and Core processors with Intel Distribution of OpenVINO Toolkit. In: TENCON 2019-2019 IEEE region 10 conference (TENCON). IEEE, pp 1783–1788

Mittal V, Bhushan B (2020) Accelerated computer vision inference with AI on the edge. In: 2020 IEEE 9th international conference on communication systems and network technologies (CSNT). IEEE, pp 55–60

OpenVINO Model Zoo. https://github.com/OpenVINOtoolkit/open_model_zoo

Optimize machine learning models. https://www.tensorflow.org/model_optimization

Osokin D (2018) Real-time 2d multi-person pose estimation on CPU: lightweight OpenPose. ArXiv Preprint arXiv:1811.12004

Polino A, Pascanu R, Alistarh D (2018) Model compression via distillation and quantization. ArXiv Preprint arXiv:1802.05668

Rezaee AA, Yaghmaee MH, Rahmani AM (2014) Optimized congestion management protocol for healthcare wireless sensor networks. Wirel Pers Commun 75(1): 11–34

Rüttimann BG, Stöckli MT (2020) From batch & queue to industry 4.0-type manufacturing systems: a taxonomy of alternative production models. J Serv Sci Manage 13(2): 299–316

Sandler M et al (2018) Mobilenetv2: inverted residuals and linear bottlenecks. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp 4510–4520

Schwarz JA, Martin E (2016) Performance evaluation of a transportation-type bulk queue with generally distributed inter-arrival times. Int J Prod Res 54(20):6251–6264

Szegedy C et al (2016) Rethinking the inception architecture for computer vision. In: Proceedings of the IEEE conference on computer vision and pattern recognition, pp 2818–2826

Takawale HC, Thakur A (2018) Talos app: on-device machine learning using tensorflow to detect android malware. In: 2018 Fifth international conference on internet of things: systems, management and security. IEEE, pp 250–255

TensorFlow Model Garden. https://github.com/tensorflow/models

Titarmare N, Yerlekar A (2018) A survey on patient queue management system. Int J Adv Eng Manage Sci 4(4): 239985

Using Intel R VTuneTM Amplifier to Optimize Media Video Applications. https://software.intel.com/content/www/us/en/develop/download/white-paper-using-vtune-to-optimize-media-video-applications.html

Wu B et al (2018) Mixed precision quantization of convnets via differentiable neural architecture search. ArXiv Preprint arXiv:1812.00090

Yang Q et al (2019) Federated machine learning: Concept and applications. ACM Trans Intell Syst Technol 10(2):1–19

Zhang J et al (2017) Energy-latency tradeoff for energy-aware offloading in mobile edge computing networks. IEEE Internet Things J 5(4):2633–2645