Video Inference Parallelization with Tensorflow Serving

Group 7: Wen-Fu Lee, Steve Wang, Yahn-Chung Chen
University of Wisconsin-Madison

Abstract

Video inference in edge devices is challenging due to low latency requirement. Even with a inference-optimized model like NoScope, it is still hard to achieve real-time inference in one machine with limited resources, such as CPU, GPU, and memory. Thus, we propose Distributed NoScope, an extension of NoScope deployed in a distributed environment for more efficient video inference. With this approach, Distributed NoScope can achieve asynchronous processing for different components' tasks, and demonstrates up to 5.52 times speedups for video inference on real-world data over NoScope.

1 Introduction

Camera video data is exploding. For example, the UK alone has over 4 million CCTVs. Recent advances in deep learning enable automated analysis of this growing amount of video data, which sift through lifetimes of video that no human would ever want to watch. And it allows us to query for objects of interest, detect unusual and abnormal events so that we can create real-time alerts. However, these deep learning methods are extremely computationally expensive. It is still challenging for real deployment at scale according to the latency requirement, bandwidth requirement, accuracy requirement and resource limitations, especially on edge devices such as cameras.

Serving classifiers on edge devices or even a cloud is very expensive. The cost-effective solution is to train job-specific classifiers on the cloud and ship the trained models to the edge devices. Recently, there are some researches about training efficient models for job-specific tasks. For example, NoScope combines difference detection and specialized CNNs with fewer parameters so that the inference can be much faster compared to some state-of-the-art neural network such as Fast-R-CNN with little trade-off in accuracy.

However, when NoScope comes to resource limitation, it still falls short and cannot meet latency requirement. In this project, we separate different functional modules in NoScope and deploy them on different nodes with different numbers of replicates. Moreover, we also try to implement asynchronous version so that the latency can be further reduced.

In this project, we plan to experiment with different settings with the following design decisions and measure their latency, memory usage and CPU/GPU usage:

1. Decision for component placement: difference detectors, specialized CNNs and a reference NN can be executed on different nodes w/ or w/o GPU.

2. Decision for other parameters: number of served models, batch size, input image resolutions.

In summary, we make the following contributions in this work:

1. Distributed NoScope, an extension of NoScope in a distributed environment for more efficient video inference.

2. Docker-based deployment, which expedites server deployment by using containers with the minimal runtime requirements of the application, and also simplifies the maintenance.

3. An evaluation of Distributed NoScope on fixed-angle binary classification demonstrating up to 5.52 times speedups on real-world data.

2 Related Work

NoScope. In order to speed up inference over video at scale, NoScope, a system with inference-optimized model search (see Figure 1), provides a good reference for achieving this goal. First, NoScope exploits the scene-specific locality by training a specialized model for exchanging generality of standard NNs to achieve faster inference. With this specialized model, NoScope can run at over 15,000 frames per second compared to YOLOv2s 80 frames per second. Second, NoScope exploits the temporal locality by learning a low-cost difference detector to determine whether the contents have changed across frames instead of running a full NN. NoScope then combines these models in a cascade by performing efficient cost-based optimization to select both model architectures (e.g., the number

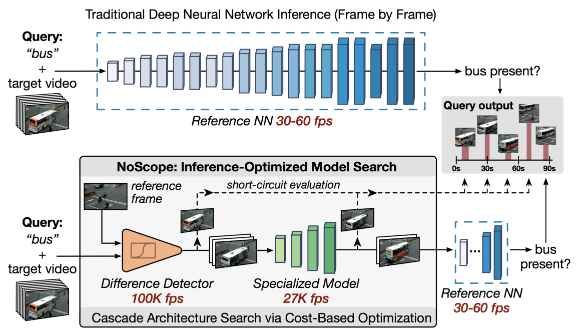

Figure 1: Given an input video, target object, and a reference neural network, NoScope automatically searches for and trains a cascade of models, including difference detectors and specialized networks that can reproduce the binarized outputs of the reference network with high accuracy, but up to three orders of magnitude faster.

of layers) and thresholds for each model, thus maximizing throughput subject to a specified accuracy target.

Tensorflow Serving. The current implementation of NoScope doesn't support execution in a distributed environment and doesn't have flexibility to scale up different components. Tensorflow Serving is introduced here to help us achieve these goals. TensorFlow Serving is a flexible, high-performance serving system for machine learning models, designed for production environments. TensorFlow Serving makes it easy to deploy new machine learning models with Docker containers, and one can request for the inference result of data with RESTful API or gRPC. In this project, we choose RESTful API as the interface between different components.

Asyncio. In our project, we use RESTful API as the interface for different functional components. It is important to reduce the idle time of our computing resources (e.g. the time between sending requests and receiving responses). Asyncio is a python standard library to write concurrent codes using the async/await syntax and is often a perfect fit for IO-bound and high-level structured network codes, just like the case of our project.

3 Design

Our work for distributed NoScope (see Figure 2) mainly consists of two parts: Server-Side Model Deployment and Client-Side Implementation.

Server-Side Model Deployment. Before model deployment, we need to attain models, the specialized NN and the reference NN, which can be used by Tensorflow serving images. The structure of our specialized NN model is a shallow CNN with 50x50x3 input resolution, and an output which indicates the confidence of whether the target object exists in a frame. We train the model based on the true label from YOLOv2. As for the reference NN model, we download the weights of YOLOv2 and the related configuration file, and convert them into a Tensorflow model.

After we attain all the Tensorflow models, we can use docker to deploy the model servers with Tensorflow serving image with some specification such as CPU use. To serve a model with GPU, we use Nvidia-docker image together with Tensorflow serving image. Client side can then use RESTful interfaces to send inference requests.

To scale up different components of distributed NoScope, we can deploy multiple model servers on different nodes, and the number of each component to be used can be decided at the client side according to the latency requirement and available computation resources.

Client-Side Implementation. For our application, the edge device in the client side is a camera at a fixed angle. First, before it sends video frames to the difference detector, those frames are first scaled down to the 50x50x3 resolution since the difference detector does not require high resolution for high detection accuracy. That is, scaling down the frames can save computation time while still keeping the detection accuracy. If not,

Second, the difference detector is used here to check whether the current frame is similar to a recent frame. If the Mean Square Error (MSE) between these two frames is smaller than T_{diff}, we can reuse the inference result of the recent frame without spending another video inference time since they are similar enough. Otherwise, the current frame is sent to a specialized NN for video inference via a RESTful interface. Specifically, all the inference requests will be put in a queue, and a load balancer (LB) will decide to which specialized NN each inference request will be sent. Here, LB is implemented to randomly select one of the specialized NNs in a way of uniform distribution.

Third, as stated previously, the specialized NN will return a confidence how likely the frame contains a target label or not. If the confidence is higher than c_{high}, it means that the frame contains the object of interest. If the confidence is lower than c_{low}, there is no any object of interest in the frame. If the confidence is between c_{high} and c_{low}, the inference result is not reliable so that it will be sent to a reference NN for further video inference. Still, this inference request is sent via a RESTful interface, and a LB will adopt the same mechanism to decide which reference NN to use.

Note that with Asyncio we can continuously send all the above video inference requests without waiting

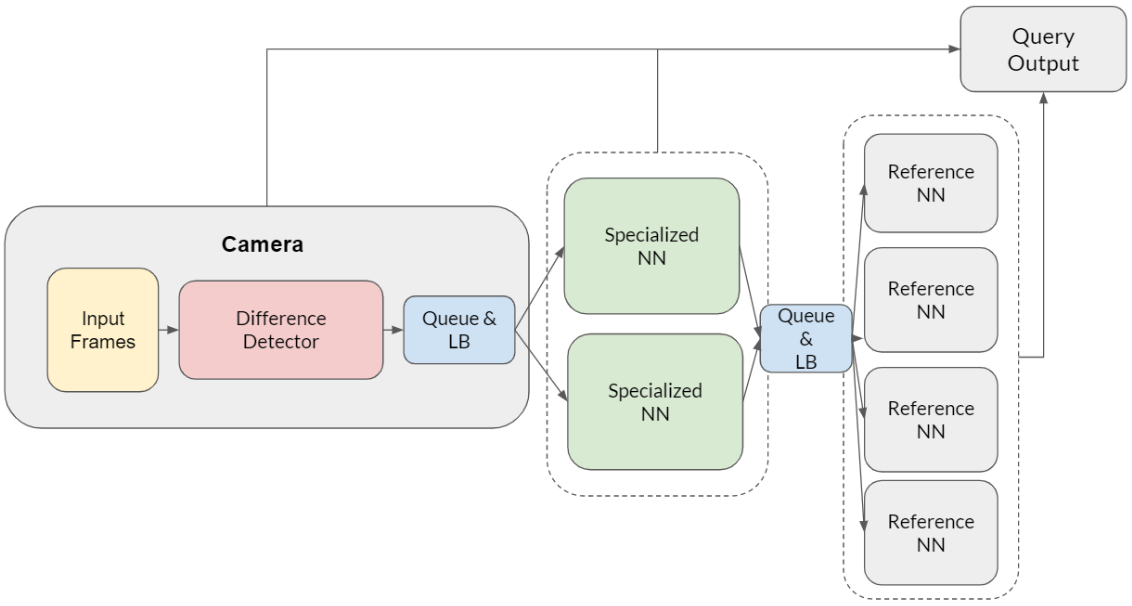

Figure 2: This figure shows the architecture of our design. Edge device such as camera consist of a client and a difference detector. And specialized NNs and Reference NNs are deployed on server sides.

those inference results back, which further saves inference time.

4 Evaluation

The main purpose of this project is to evaluate the performance of NoScope in distributed computing environment. We measured the performance of distributed NoScope based on different designs mentioned in Figure 2. The evaluation demonstrates the (i) latency for different settings and (ii) potential bottleneck of distributed NoScope, In our experiment, the throughput can be varied by the total number of frames loaded in the client.

4.1 Experiment Setup

We measured performance on a distributed NoScope of one client , one Specialized NN server, and four Reference NN servers. Each docker container is corresponding to a served component. All the docker containers are configured with maximal 4 CPU resource.

All the machines are configured with two 10-core CPUs at 2.20 GHz, 192GB Memory and one 1 TB 7200 RPM disks. Because the CPU resources on a single machine are much more than the allowable amount of CPU resources for a single container, we can increase the number of docker containers to achieve scaling for different settings.

4.2 Dataset

The dataset we obtained is downloaded from NoScope github repo, including the video jackson-town-square.mp4 (60-hour video length and 30 FPS) and the video labels jackson-town-square.csv. The video labels are obtained by the YOLOv2's prediction over all frames of the video, and the object of interest for our video inference is the ”car” label. Here, we use two kinds of video resolutions for inference in our design: 50x50 for the difference detector and the specialized NN and 608x608 for reference NN, respectively. The two former models can use lower video resolution while achieving the same accuracy; the latter model need a higher video resolution due to a highly complex neural network. Except the reference model, which is already pre-trained by [12], the difference detector and the specialized NN are trained on the first 5,000 frames (the training set) of the video, and the remaining frames (the evaluation set) are used for video inference instead.

4.3 Evaluation Metrics

In this project, latency is the main criteria for evaluating performance. The total latency can be formulated as the following equilibrium:

$$Latency = T_n + T_{c.q} + T_{s.q} + T_{s.c}$$

, where T_n is the transmission time for one request, $T_{c.q}$ is the total queuing delay in client, $T_{s.q}$ is the total queuing delay in server and $T_{s.c}$ is the computation time for a single request in server.

Let's assume we have multiple servers to consume the request and multiple process at client side. Latency can also be written as the following:

$$Latency = T_n + T_{c.q} + \frac{N_R - N_S}{N_S} * T_{s.c} + T_{s.c}$$

, where N_R is the number of requests, N_S is the number of server. By varying the number of frames loaded into the client, the throughput can be increased and then used to measure the performance of NoScope. In our experiment, we changed the throughput from 50 frames to 5000 frames, applied different numbers of specialized NN and reference NN, and then analyzed the data to find the relation between different design decisions and latency and further recognize the potential bottleneck in our design.

4.4 Reference NN Benchmark

4.4.1 Single Request Benchmark

In order to find the optimized design for distributed NoScope, the first step in this experiment would be identify the minimal latency for each component. Since NoScope can be divided to difference dectator, specialized NN and Reference NN, we can send one frame to each of them to measure the latency. The computation time for each component is found to be: 0.013 second for difference detector, 0.026 second for specialized NN and 4.26 for reference NN.

Other than the computation time, another critical factor is the network transmission time. To measure transmission time, we sent one request to server and one to the local for comparing the difference of response time in each setting. The transmission time is found to be 0.2 second in our environment.

The transmission time and computation time for each component provide us a yardstick for better evaluating distributed NoScope. The details are further described in next section.

4.4.2 Scaling Up Reference NN

As mentioned in 4.4.1, the computation time of difference detector and specialized NN are much faster than the one for reference NN. Therefore, we focus on modifying the number of reference NN (N_s) and set it as the experimental variable to measure the latency. The relations between latency and number of frames are shown in Figure 3.

In this experiment, we can first assume there is always queuing delay in server side, because the time for computing one request is longer than the transmission time.

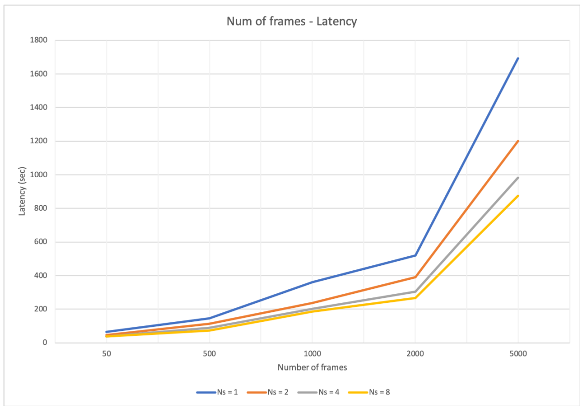

Figure 3: This figure shows the latency against different number of frame sent to RNN server. The available computing resource can be modified by changing the number of RNN Server (N_n).

It implies that once the request arrive in the server, it has to wait until any of the CPU is available. Based on this assumption, the latency equation showed in 4.3 can be used to measure performance.

According to Table 1, in the case of 500 frames, we can solve the equation and get $T_n + T_{c.q} = 64$ and $T_{s.q} + T_{s.c} = 83$, and for the case of 1000 frames, the latency would be $T_n + T_{c.q} = 160$ and $T_{s.q} + T_{s.c} = 200$, which are both around 2.4 times more than the corresponding ones in 500-frames case. Also, when it comes to the number of requests sent to reference NN servers, we found that the number is 2.4 times more in 1000-frames case than the one in 500-frames case. This pattern matched the equation in 4.3 which showed $T_{s.q}$ is proportional to N_s. On the top of that, another interesting property observed in this result is that $T_{c.q}$ is also proportional to N_s since T_n is a fixed value.

In accordance with the equation, $Latency = T_n + T_{c.q} + \frac{N_R - N_S}{N_S} * T_{s.c} + T_{s.c}$, we know that the latency can always be reduced by increasing the number of server no matter how many requests have to be processed. Therefore, $T_n + T_{c.q} + T_{s.c}$ would be the lowest possible latency while distributed NoScope have enough computing resource.

However, even though the latency from server side can be reduced significantly, we found the potential bottleneck would happen in client side. As we show above, the $T_{c.q}$ is proportional to the number requests sent from the client, and the possible way for reducing $T_{c.q}$ is to increase the CPU resource in client side, which is contradicted to our assumption that edge devices have limited computing resource. Therefore, the latency of $T_{c.q}$ would be a potential bottleneck for distributed NoScope while client has limited computing resource.

Table 1		Latency in diff. N_S			
N_F	N_R	1	2	4	8
50	22	66	47	38	38
500	49	147	113	89	74
1000	117	360	237	202	186
2000	170	521	390	305	267
5000	569	1692	1201	984	875

Table 1. where N_F is the number of frame , N_R is the number of request and N_S is the number of RNN server.

4.5 Distributed NoScope

In the introduction, it is mentioned that the computing resources in edge device are scarce and may not be enough for expensive computation like video inference. In this experiment, the amount of CPU in edge device is configured as two and used for either served non-distributed NoScope or client in distributed NoScope.

After processing 500 frames in each scenario, the latency for non-distributed NoScope is found to be 784 second , and the latency for distributed NoScope is found to be 142 second, which is 5.52 times faster than non-distributed NoScope. (See Figure 4)

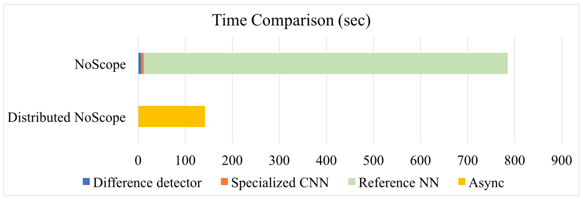

Figure 4: Time comparison between Distributed NoScope and NoScope.

5 Future work

Batching. In this project, we only send one frame per request, but some computation may benefit from batching multiple frames. In our project, we find that for 1 frame per request, it doesn't show significant difference in latency between using GPU and using CPU only. But what if we send multiple frame per request? GPU may be much better than CPU only. To implement batching, we will need to create a queue for sending to specialized NNs and another queue for sending to reference NNs. In our case, the frames sent to specialized NNs and reference NNs are not continuous, and the proportion of frames need to be sent to servers will also varies with time. So we might need to dynamic adjust the size of queue so that it will not lower down the latency too much.

Payload Compression. In the evaluation, we find that client queue time still increases with number of frames. And it may due to the bandwidth limit of sending/receiving to data to/from server. Compressing the payload may further help reduce queuing. If we do compression with batched frame, compression technique used for continuous frame can be further applied.

References

[1] G. Ananthanarayanan, P. Bahl, P. Bodk, K. Chintalapudi, M. Philipose, L. Ravindranath, and S. Sinha, ”Real-time video analytics: The killer app for edge computing,” *Computer*, vol. 50, no. 10, pp. 58-67, 2017.

[2] D. Kang, J. Emmons, F. Abuzaid, P. Bailis, and M. Zaharia, ”NoScope: optimizing neural network queries over video at scale,” *PVLDB 10*, no. 11, pp. 1586-1597, 2017.

[3] TensorFlow Serving for model deployment. Retrieved from https://www.tensorflow.org/serving/.

[4] Deep Learning on the Edge. Retrieved from https://towardsdatascience.com/deep-learning-on-the-edge-9181693f466c.

[5] Machine Learning Inference at the Edge. *AWS Summit*. Retrieved from https://www.slideshare.net /AmazonWebServices/machine-learning-inference-at-the-edge-94229494.

[6] asyncio - Asynchronous I/O in Python. Retrieved from https://docs.python.org/3/library/asyncio.html.

[7] aiohttp - Asynchronous HTTP Client/Server for asyncio and Python. Retrieved from https://aiohttp.readthedocs.io/en/stable/.

[8] Docker. Retrieved from https://docs.docker.com.

[9] Netdata. Retrieved from https://github.com/netdata.

[10] jackson-town-square video. Retrieved from https://storage.googleapis.com/noscope-data/videos /jackson-town-square.mp4.

[11] jackson-town-square csv. Retrieved from https://storage.googleapis.com/noscope-data/csvs-yolo/jackson-town-square.csv.

[12] YOLO: Real-Time Object Detection. Retrieved from https://pjreddie.com/darknet/yolo/.