Pragmatic Approach to CNN Deployment using Modern Tooling

Abstract

Rapid prototyping with convolutional neural networks (CNNs) for product developers is a difficult operational procedure for many software engineers. Significant considerations are required in terms of computational resources and surrounding infrastructure for value oriented products to be developed and there are limited open-source turn-key solutions regarding neural network product development. While jupyter notebook runtime is a robust solution for architecture evaluation, it is not robust to integration within the software development context, as considerations have to be made regarding deployment environment of the specific team's product. Prior work on developing turn-key solutions are heavily monetized and hidden away behind cloud service provider regions. The target contribution is a self-contained Helm deployment consisting of a collection of object detection models, (Sparse R-CNN, DETR, Faster R-CNN in this instance) executing inference task and calculating evaluation metrics given specified training weights and making comparisons across the model stack. This results in a capability of running experiments in a target deployment environment instead of a controlled environment of a jupyter notebook runtime.

1. Introduction:

There have been considerable advances and breakthroughs regarding various implementation of CNN architectures where the model architecture itself have been iterated over, and their performance evaluated. In order to measure performance on both the training and inference workloads negate any external variables that can affect model performance, the experimental environment is generally a jupyter notebook runtime. To understand evaluation and performance of CNNs in particular, a literature review surrounding improvements to CNN architectures and training approach with respect to model performance is considered. The idea is to utilize the learnings from the respective papers in terms of model architecture optimizations or training algorithm optimization, and draw conclusions on where on a deployed environment, these factors can be integrated, and see if there is performance metric parity between a controlled environment and a typical deployment environment. For example, the Sparse R-CNN architecture focus on reduction of FLOPS from a training standpoint by utilizing sparse computations throughout its training pipeline (Peize Sun, 2021). Although, this paper will not be exploring the evaluation of the architecture itself, its inference results will be compared across a similar family of CNN models (DETR, and Faster R-CNN). Need for computational efficiency go hand-in-hand with an effort to parallelizing workflows. As a result, the paper also draws inspiration from studies on federated learning, where devices located on a mesh network can communicate across defined protocols can pool computational resources to train large models, while each device on the mesh network is constrained to IoT caliber performance envelope (Chaoyang He, 2020). This also raises the question, are there any hardware caveats that need to be considered as part of either sparse computation or overhead that come with sharing information across computing nodes? While implementing the DeltaCNN architecture, the authors of the respective study observed there are real-world performance caveats related to sparse computation. The GPU and its behavior surrounding allocation and management of VRAM cause sparse calculation to experience runtime latencies. The scope of this CNN inference implementation does not consider findings encountered during the implementation of DeltaCNN. However, extension to the proposed deployment will be offered as future benchmarking considerations (Mathias Parger, 2022).

Given the above learnings, the proposed implementation considered the following requirements:

- Deployment must have minimal pre-requisite dependencies
- Model deployment must be user hardware and operating system agnostic
- Model deployment must be fault tolerant to failures
- Model deployment is atomically scalable
- Application support for hot-reloading during development

The reimplemention with these factors in mind resulted in a full-stack application capable of providing baseline level evaluation metrics, for which the application can be extended upon. The pragmatic approach to the infrastructure also results in turn-key integration in any production environment.

2. Background:

When developing products or iterating on prototypes, one of the largest pain point is dependency and infrastructure management. Often times, this is an issue where experiments are executed on a target environment, that may include complex dependencies specific to a product target environment, and either scaling the application, or developer collaboration is significantly effected. This is where the concept of infrastructure-as-code (IaC) is embraced, where the appropriation of compute resources and environment management is automated through either imperative commands that are executed in sequence, or declarative statements indicating the desired state of your infrastructure (IaC, 2022). No one piece of software or tooling is defined as IaC, rather a collection of software, tooling, and protocol is considered an instance of IaC implementation. While there are standard approaches to the implementation of IaC, each application or product has an unique set of resource challenges that need to be met. For AI/ML applications and products, these complexities exist in both I/O bound operations and allocation of discrete compute resources. Significant assumptions are made to simplify the implementation with respect to this study with considerations made for future improvements and extensibility.

3. Literature Review: Sparse R-CNN

Paper: *Sparse R-CNN: End-to-End Object Detection with Learnable Proposals [CVPR 2021]: https://arxiv.org/abs/2011.12450* (Peize Sun, 2021)

3.1. Storyline

High-Level Motivation: In the object detection space, the most mature algorithm in use and reviewed is comprised of dense computations related to non-learnable objects in the inference and training process. When considering classical computer vision methods, the sliding window paradigm applies classifiers to dense image grids. A modern implementation of this technique involves a one-stage detector pipeline involving detection based on dense feature map anchor boxes, or reference points, and how close the predictions are to the respective anchor boxes or reference points. Such an approach is computationally expensive. For example, with respect to the anchor box approach of defining object candidates, there would exist redundant or near-duplicate selections of object candidacy. This effect requires post-processing to eliminate the redundancy. Related work exists to mitigate some of the issues, where some combination of dense and sparse pipelines are offered within both the image frame localization and image feature extraction. However, prior to this work, a pure, well-developed sparse method of object detection did not exist, when considering pipelines for both determining candidacy of a region as an object and feature computations around a determined region. A fully sparse method for object detection would allow for faster convergence of the training loop for a given feature extractor architecture (such as a Feature Pyramid Networks (FPN)), while maintaining the robust performance provided using dense object detection pipelines.

Prior Work: Two approaches can be considered when looking at related works regarding the issue. A dense-to-sparse detector pipeline exists where the algorithm interacts with dense image features to obtain a collection of sparse sets made up of proposal boxes. The approach has been defined as DEtection TRansformer (DETR). Previous short-comings are still reflected in this approach. The Non-Maximum Suppression approach to redundancy elimination is still required as the proposal boxes may contain redundancies. Furthermore, the sparse object boundary proposals are being derived from dense feature maps, maintaining dependency on dense computations.

A sparse approach to region proposal exists and is implemented as part of G-CNN architecture. Proposals are derived from an iterative approach starting from a multi-scaled grid of fixed boxes, regardless of input image subject. The following approach to object region detection does not perform as well as the dense approach or the semi-dense approach presented by DETR. In addition, the initialization of the multi-scale grid is treated as a hyper parameter, as the number of grids is ”handcrafted”.

Research Gap: The research gap that exists consists of formulating a fully-sparse detection pipeline, which can have comparable performance to dense object detection. A fully-sparse implementation would allow for higher computational efficiency, longer training duration, larger dataset training, and the unlocking of capabilities for edge deployments for object detection. If the feasibility of implementation into engineering problems were to improve, software framework development would follow - further expanding on available use case.

Contributions: The main contribution of this work is providing a fully-sparse pipeline for object detection. This was achieved by implementing a learnable proposal box approach for denoting object localization, and a learnable encoding of the proposal box, dubbed the learnable proposal feature, maintaining information resolution of the learned proposal boxes. An additional workflow is introduced, called **Dynamic Instance Interactive Head**, responsible for introducing non-linearity to the learnable feature, before it is passed to a classification MLP module. The whole pipeline is executed without interacting with global dense properties of the image or dense feature maps.

3.2. Proposed Solution

Learnable Proposal Box: A fixed small set of learnable boxes are initialized as the localization proposal instead of acquiring the proposal from an RPN (Region Proposal Network). These proposal would have the shape $(N \times 4)$, where its is N initial proposal boxes of 4 dimensions. The dimensions comprise of index, normalized center coordinates, height, and width. The following parameters are updated through back-propagation. As these parameters are learnable, initialization state is flexible.

Learnable Proposal Feature: As the learnable proposal boxes are coarse representation of object localization, the features are a high dimensional vector set of size $(N \times d)$ where d is vector size of each sample. The purpose of this latent vector is to encode additional information from the area within the proposal box.

Dynamic Instance Interactive Head: The following module is used object location prediction and classification (includes non-linearity operations mentioned previously. This module takes the proposal boxes, and features, derives another set of feature maps using RoIAlign operations. The RoI features and proposal features interact to create a final object feature. The final object feature is then passed through a nonlinear module containing convolutions and ReLU functions before sending it to an MLP module for regression.

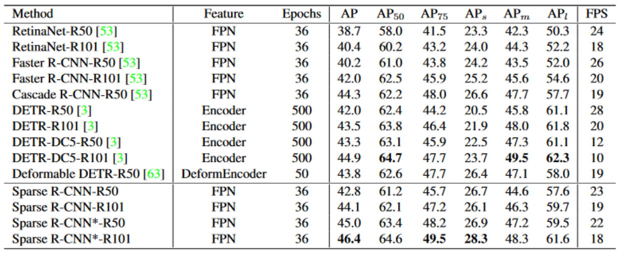
3.3. Claims-Evidence

Claim 1: Sparse R-CNN Outperforms Well-Established Mainstream Detectors (RetinaNet/Faster R-CNN on Resnet-50)

Evidence 1: In the following table *[Paper: Main Results, Table 1]*, you can see that the average precision is higher for Sparse R-CNN using Resnet-50 as the backbone when compared between RetinaNet and Faster R-CNN using the same backbone.

Claim 2: Sparse R-CNN Converges Much Faster Than Mainstream Detectors

Evidence 2: The following figure *[Paper: Main Results, Figure 2]* shows Sparse R-CNN reaching higher AP within the same training schedule as RetinaNet and Faster R-CNN. DETR requires longer training schedule but is also shown in an effort to be comprehensive.



Method	Feature	Epochs	AP	AP_{50}	AP_{75}	AP_s	AP_m	AP_l	FPS
RetinaNet-R50 [53]	FPN	36	38.7	58.0	41.5	23.3	42.3	50.3	24
RetinaNet-R101 [53]	FPN	36	40.4	60.2	43.2	24.0	44.3	52.2	18
Faster R-CNN-R50 [53]	FPN	36	40.2	61.0	43.8	24.2	43.5	52.0	26
Faster R-CNN-R101 [53]	FPN	36	42.0	62.5	45.9	25.2	45.6	54.6	20
Cascade R-CNN-R50 [53]	FPN	36	44.3	62.2	48.0	26.6	47.7	57.7	19
DETR-R50 [3]	Encoder	500	42.0	62.4	44.2	20.5	45.8	61.1	28
DETR-R101 [3]	Encoder	500	43.5	63.8	46.4	21.9	48.0	61.8	20
DETR-DC5-R50 [3]	Encoder	500	43.3	63.1	45.9	22.5	47.3	61.1	12
DETR-DC5-R101 [3]	Encoder	500	44.9	**64.7**	47.7	23.7	**49.5**	**62.3**	10
Deformable DETR-R50 [63]	DeformEncoder	50	43.8	62.6	47.7	26.4	47.1	58.0	19
Sparse R-CNN-R50	FPN	36	42.8	61.2	45.7	26.7	44.6	57.6	23
Sparse R-CNN-R101	FPN	36	44.1	62.1	47.2	26.1	46.3	59.7	19
Sparse R-CNN*-R50	FPN	36	45.0	63.4	48.2	26.9	47.2	59.5	22
Sparse R-CNN*-R101	FPN	36	**46.4**	64.6	**49.5**	**28.3**	48.3	61.6	18

Figure 1. Accepted Precision: Model Comparison

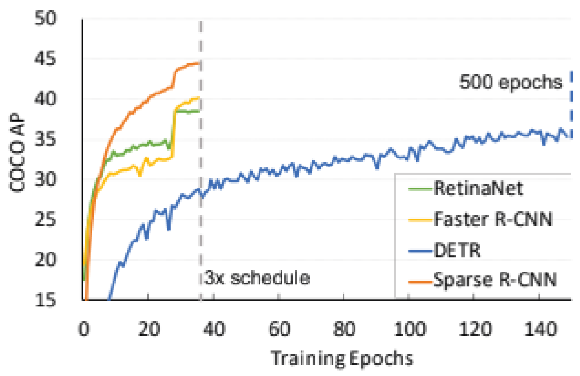
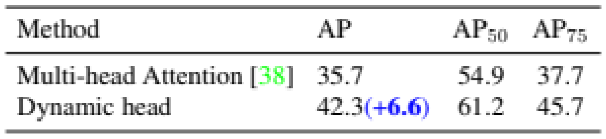


Figure 2. Model Training Schedule Comparison

Claim 3: Dynamic Head method offered as part of the contribution to this paper performs better than the use of multi-head attention method for object recognition.

Evidence 3: According to the following figure *[Paper: Main Results, Table 9]* AP metrics for Dynamic head is significantly better than multi-head attention



Method	AP	AP_{50}	AP_{75}
Multi-head Attention [38]	35.7	54.9	37.7
Dynamic head	42.3(+6.6)	61.2	45.7

Figure 3. Attention Methods: Dynamic vs. Multi-Head

3.4. Critique and Discussion:

The most interesting part of the paper was its approach towards implementation evaluation. I appreciated the standardized experimental setup of using a variation of architectures in its implementation, such as setting the feature extraction approach static, using a variety of backbone architectures, and interchange of self-attention and interactive head modules. It provides a comprehensive exposition about its performance.

However, I do believe a structured ablation study with defined objective would have been helpful in solidifying the claims of the superior performative nature of this architecture. An example would be evaluation of the target architecture and its competitors over a permutation of backbone architectures, feature extraction methods, and different proposal defining approaches. Evaluation of the architectures over datasets other than COCO could also be considered.

4. Literature Review: Federated Learning

Paper: *Group Knowledge Transfer: Federated Learning of Large CNNs at the Edge [NeuralIPS 2020] https://arxiv.org/abs/2007.14513* (Chaoyang He, 2020)

4.1. Storyline

High-Level Motivation: CNNs architectures utilized for object detection and image classification have been known to be computationally intensive. Various approaches have been considered in the field to optimize performance including, but not limited to: improved feature extractors, conversion from dense to sparse calculations, etc. The following paper takes a different approach with a specific use case in mind: Training models on edge compute devices, where computational resources are constrained using a federated learning paradigm. Having larger model architectures is an approach to improving inferencing accuracy for CNNs. However, edge devices like IoT devices do not have the GPU hardware or the required capable compute resources to train a model like ResNet101. As a result, an approach was considered where edge devices would train on domain specific dataset on a model that is appropriate for the edge device, and send the hidden states and feature map to a central server with appropriate hardware that can collect edge device hidden state, to perform its own training on a larger CNN architecture. The model states post-training are propagated back to the edge devices in an approach where it is compatible with models within the edge devices. Prior work with federated learning involves batch training small batch sizes on the same edge devices on the same dataset which are centralized on a powerful server. An improvement is considered by the paper where edge devices can have any supported model training on non-homogeneous datasets to train the central server. Bi-directional hidden state transfer is also considered for stronger inference performance.

Prior Work: The paper reference an approach to Federated Learning (FL) where a learning aggregation architecture is used. Their own platform FedAVG, performs localized SGD on edge device compatible neural networks, and these trained models are averaged out on a single central server to accommodate learning achieved by all edge devices. The central server then communicates the trained model back to the edge devices. The issue with this approach is the fact that the central server will be limited to aggregating results from a small neural network that is compatible with edge devices. Another approach to federated learning is called Split Learning [SL]. This approach divides a large model into partitions and offloads the computation to the edge device for each partition. the major issue with this approach is I/O bound networking costs, causing the training loop to perform less efficiently from a training speed perspective.

The FedGKT implementation introduced in this paper takes advantage of specific techniques, such as: alternated learning (passing training state between server and client), alternated loss function optimization, and reformulation of resource partition through non-convex optimization.

Research Gap: Within FL, the targeted research gap for the contribution is related to possibilities of using non-homogeneous datasets with respect to each edge device and sharing the training states of each edge device. As edge device datasets could be limited based on deployment circumstances, an approach to aggregate the datasets and learn from the aggregate has model performance implication of deployed systems. This reduces the chance of model on one specific edge device over fitting based on the dataset it could collect and train on. This approach has the chance to reduce the overall floating point operations (FLOPS) required to complete a training loop, as each edge device would receive the dataset recovered within the device network and each edge device would train over the entire dataset, creating redundancies.

Contributions: The main contributions of the paper were algorithms related to group knowledge transfer of edge device training loop, resource-bound computation allocation optimization and loss-function pertaining to edge device performance boost. The collective goal of implementing the items was to reduce overall computational overhead, and possibly reduce the time required to train a series of deployed edge devices. The authors verify its implementation using ResNet and its model variants as the benchmark for evaluation.

4.2. Proposed Solution:

Some of the system assumptions are listed below:

- Supervised learning with C categories in the dataset D
- K clients (edge devices) with own dataset D
- $D^k := \{(X_i^k, y_i)\}_{i=1}^{N^{(k)}}$; where X_i is the ith training

sample, y_i is the label, $N^{(k)}$ is the sample number with respect to edge device

Resource-Bound Optimization: The CNN is divided into two models, the feature extractor model and the global server side model (W split into W_e and W_s respectively). A classifier model is created to for the small feature extractor to create a fully-trainable model on the edge (W_c is the classifier working with W_e on the edge devices). A non-convex optimization problem is solved to train F_c and F_s (full edge and server models).

$$\underset{\boldsymbol{W}_s}{\operatorname{argmin}} F_s(\boldsymbol{W}_s, \boldsymbol{W}_e^*) = \underset{\boldsymbol{W}_s}{\operatorname{argmin}} \sum_{k=1}^{K} \sum_{i=1}^{N^{(k)}} \ell_s\left(f_s(\boldsymbol{W}_s; \boldsymbol{H}_i^{(k)}), y_i^{(k)}\right) \tag{2}$$

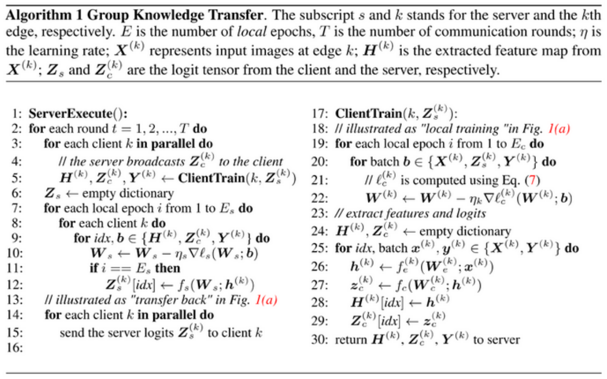
$$\textit{subject to: } \boldsymbol{H}_i^{(k)} = f_e^{(k)}(\boldsymbol{W}_e^{(k)}; \boldsymbol{X}_i^{(k)}) \tag{3}$$

$$\underset{(\boldsymbol{W}_e^{(k)}, \boldsymbol{W}_c^{(k)})}{\operatorname{argmin}} F_c(\boldsymbol{W}_e^{(k)}, \boldsymbol{W}_c^{(k)}) = \underset{(\boldsymbol{W}_e^{(k)}, \boldsymbol{W}_c^{(k)})}{\operatorname{argmin}} \sum_{i=1}^{N^{(k)}} \ell_c\left(f^{(k)}((\boldsymbol{W}_e^{(k)}, \boldsymbol{W}_c^{(k)}); \boldsymbol{X}_i^{(k)}), y_i^{(k)}\right) \tag{4}$$

$$= \underset{(\boldsymbol{W}_e^{(k)}, \boldsymbol{W}_c^{(k)})}{\operatorname{argmin}} \sum_{i=1}^{N^{(k)}} \ell_c(f_c^{(k)}(\boldsymbol{W}_c^{(k)}; \underbrace{f_e^{(k)}(\boldsymbol{W}_e^{(k)}; \boldsymbol{X}_i^{(k)})}_{\boldsymbol{H}_i^{(k)}}), y_i^{(k)}) \tag{5}$$

Figure 4. Optimization Approach for F_c

Group Transfer Algorithm: During each round of training, the client will perform stochastic gradient descent for a defined period of epochs before the state is sent to the large model server. The state contains logits and extracted feature. The server trains a much larger model using the aggregate states and logits. The server then send the large model train logits to the edge devices. This loop is executed over defined epoch iterations. This framework is then applied to a target CNN architecture.


Algorithm 1 Group Knowledge Transfer. The subscript s and k stands for the server and the kth edge, respectively. E is the number of *local* epochs, T is the number of communication rounds; η is the learning rate; $\boldsymbol{X}^{(k)}$ represents input images at edge k; $\boldsymbol{H}^{(k)}$ is the extracted feature map from $\boldsymbol{X}^{(k)}$; $\boldsymbol{Z}_s$ and $\boldsymbol{Z}_c^{(k)}$ are the logit tensor from the client and the server, respectively.

```
1:  ServerExecute():
2:  for each round $t = 1, 2, ..., T$ do
3:      for each client $k$ in parallel do
4:          // the server broadcasts $\boldsymbol{Z}_c^{(k)}$ to the client
5:          $\boldsymbol{H}^{(k)}, \boldsymbol{Z}_c^{(k)}, \boldsymbol{Y}^{(k)} \leftarrow$ ClientTrain($k, \boldsymbol{Z}_s^{(k)}$)
6:      $\boldsymbol{Z}_s \leftarrow$ empty dictionary
7:      for each local epoch $i$ from 1 to $E_s$ do
8:          for each client $k$ do
9:              for $idx, \boldsymbol{b} \in \{\boldsymbol{H}^{(k)}, \boldsymbol{Z}_c^{(k)}, \boldsymbol{Y}^{(k)}\}$ do
10:                 $\boldsymbol{W}_s \leftarrow \boldsymbol{W}_s - \eta_s \nabla \ell_s(\boldsymbol{W}_s; \boldsymbol{b})$
11:                 if $i == E_s$ then
12:                     $\boldsymbol{Z}_s^{(k)}[idx] \leftarrow f_s(\boldsymbol{W}_s; \boldsymbol{h}^{(k)})$
13:     // illustrated as "transfer back" in Fig. 1(a)
14:     for each client $k$ in parallel do
15:         send the server logits $\boldsymbol{Z}_s^{(k)}$ to client $k$
16:
17: ClientTrain($k, \boldsymbol{Z}_s^{(k)}$):
18: // illustrated as "local training "in Fig. 1(a)
19: for each local epoch $i$ from 1 to $E_c$ do
20:     for batch $\boldsymbol{b} \in \{\boldsymbol{X}^{(k)}, \boldsymbol{Z}_s^{(k)}, \boldsymbol{Y}^{(k)}\}$ do
21:         // $\ell_c^{(k)}$ is computed using Eq. (7)
22:         $\boldsymbol{W}^{(k)} \leftarrow \boldsymbol{W}^{(k)} - \eta_k \nabla \ell_c^{(k)}(\boldsymbol{W}^{(k)}; \boldsymbol{b})$
23: // extract features and logits
24: $\boldsymbol{H}^{(k)}, \boldsymbol{Z}_c^{(k)} \leftarrow$ empty dictionary
25: for $idx$, batch $\boldsymbol{x}^{(k)}, \boldsymbol{y}^{(k)} \in \{\boldsymbol{X}^{(k)}, \boldsymbol{Y}^{(k)}\}$ do
26:     $\boldsymbol{h}^{(k)} \leftarrow f_e^{(k)}(\boldsymbol{W}_e^{(k)}; \boldsymbol{x}^{(k)})$
27:     $\boldsymbol{z}_c^{(k)} \leftarrow f_c(\boldsymbol{W}_c^{(k)}; \boldsymbol{h}^{(k)})$
28:     $\boldsymbol{H}^{(k)}[idx] \leftarrow \boldsymbol{h}^{(k)}$
29:     $\boldsymbol{Z}_c^{(k)}[idx] \leftarrow \boldsymbol{z}_c^{(k)}$
30: return $\boldsymbol{H}^{(k)}, \boldsymbol{Z}_c^{(k)}, \boldsymbol{Y}^{(k)}$ to server
```

Figure 5. Group Knowledge Transfer Algorithm

4.3. Claims and Evidence

Claim 1: FedGKT has similar performance to centralized neural network training

Evidence 1: The following table *Paper: 4.2 Result of Model Accuracy* shows that FedGKT on both ResNet-56 and ResNet-110 have similar performance to FedAvg and Centralized learning.

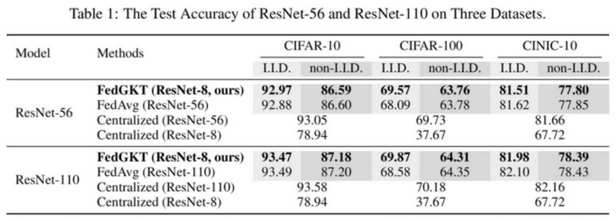


Table 1: The Test Accuracy of ResNet-56 and ResNet-110 on Three Datasets.

Model	Methods	CIFAR-10		CIFAR-100		CINIC-10	
		I.I.D.	non-I.I.D.	I.I.D.	non-I.I.D.	I.I.D.	non-I.I.D.
ResNet-56	**FedGKT (ResNet-8, ours)**	**92.97**	**86.59**	**69.57**	**63.76**	**81.51**	**77.80**
	FedAvg (ResNet-56)	92.88	86.60	68.09	63.78	81.62	77.85
	Centralized (ResNet-56)	93.05		69.73		81.66	
	Centralized (ResNet-8)	78.94		37.67		67.72	
ResNet-110	**FedGKT (ResNet-8, ours)**	**93.47**	**87.18**	**69.87**	**64.31**	**81.98**	**78.39**
	FedAvg (ResNet-110)	93.49	87.20	68.58	64.35	82.10	78.43
	Centralized (ResNet-110)	93.58		70.18		82.16	
	Centralized (ResNet-8)	78.94		37.67		67.72	

Figure 6. Testing Accuracy of Learning Paradigms

Claim 2: Compared to SL, FedGKT has much lower data costs in terms of communication

Evidence 2: According to the following graph *Paper: 4.3 Efficiency Evaluation*, FedGKT consumes approximately half the communication bandwidth required by the SL method of FL.

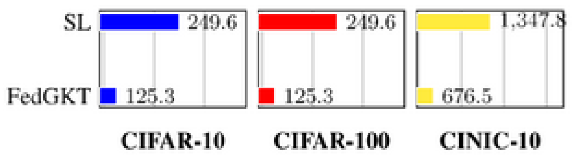


Figure 7. Communication Cost of FL (SL vs. FedGTK)

Claim 3: FedGKT scales well with number of edge devices and maintains accuracy trends with respect to model size

Evidence 3: According to the series of figures *Paper: 4.4 Ablation Study Table 4,5* scaling from 8 to 128 edge nodes does not have a significant difference in inference accuracy. In addition, the inference accuracy increases with respect to layer size, as expected.

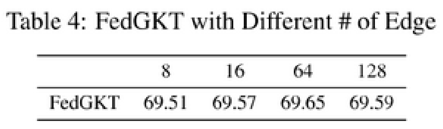


Table 4: FedGKT with Different # of Edge

	8	16	64	128
FedGKT	69.51	69.57	69.65	69.59

Figure 8. Inference Accuracy of FedGKT Running ResNet-4

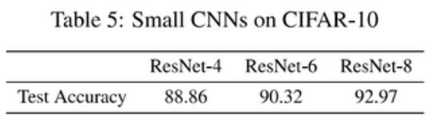


Table 5: Small CNNs on CIFAR-10

	ResNet-4	ResNet-6	ResNet-8
Test Accuracy	88.86	90.32	92.97

Figure 9. Inference Accuracy with Respect to Model Size

4.4. Critique and Discussion:

The FL approach introduced by the authors show promise of improvemnet of computational efficiency with regards to reduction of network bandwidth necessary and FLOPS necessary for FL. However, the paper had some key weaknesses. The paper performed evaluations of FL mainly on its own platform called FedAvg. It would have been a more comprehensive review of the implementation if the paper had other implementation of FL besides FedAvg and one evaluation made with respect to communication bandwidth. Furthermore, Figure 4's value in the paper is hard to interpret as they are metrics with no comparisons for FedGKT. The ablation studies and their respective tables to not have appropriate labels and the reviewer does not know key characteristics behind the table. The performance improvements in terms of FLOPS improvements were claimed but not supported.

5. Literature Review: End-to-End CNN Inference

Paper: *DeltaCNN: End-to-End CNN Inference of Sparse Frame Differences in Videos [CVPR 2022] https://arxiv.org/abs/2203.03996* (Mathias Parger, 2022)

5.1. Storyline

High-Level Motivation: CNN architecture utilized for object detection and image classification has been known to be computationally intensive. Especially so when considering real-time processing of video data. Various approaches have been considered for tackling the computational efficiency issue with the most researched topic and implementation being finding computational items within the inference pipeline that could be converted from a dense calculation to a sparse calculation. However, previous implementation contain in-memory continuity issues that are not architecture aware, which could cause issues at run-time, causing the sparse calculation to be slow. An implementation is offered (DeltaCNN) for typical CNN layers and non-linear layers where video properties related to similar scenes are taken advantage. The implementation also executes these sparse calculations in a way where errors do not add up over time. The implementation is heavily compares against the dense reference implementation *cuDNN* with results heavily favoring the new implementation.

Prior Work: Most video CNN architecture try to employ a fine-grained feature generation pipeline in key video frames with coarse update path for each frame in-between. There are certain techniques used by specific architectures.

- **Update Truncation:** RRM and CBInfer are two architecture that increase sparsity by truncating insignificant frame updates the input features without the loss of accuracy. On the contrary Skip-Convolution truncates the output features. A combination of techniques of the three architectures are used by DeltaCNN. Particular attention was placed on single pixel update triggers used by Skip-Convolution as it is crucial for not accumulating loss

- **Caching:** Most architectures cache input and output feature maps between each layer to process differences to increase sparsity before accumulating outputs. The entire operation is dense as the entire feature map is saved. This has significant memory overhead, which will impact performance. An approach was considered where key convolutional layer caching would take place. However, this can lead to significant errors as pooling and activation functions do not respond well to selective layer caching.

- **Sparse Dataset:** Architectures like SBNet and SSC can perform sparse CNN processing resulting in real-time improvement in performance. However, there are two drawbacks: the architectures rely on sparsity existing in the datasets and both architectures were mainly developed with still images in mind.

Research Gap: The target research gap relates to fully-sparse pipeline involving no dense caching approaches, and not relying on existing sparsity within inference dataset. An architecture capable of taking advantage of frame to frame similarities and reducing total storage of information would allow for faster inference. This would open up deployment opportunities in more markets as speed of inference becomes faster.

Contributions: The major contributions of the paper included a feature update architecture involving state maintenance of feature map deltas. Compounding errors needed to be considered during state updates, which are promptly addressed by leveraging linearities involving CNN layers. The non-linear layer and truncating updates are combined into one layer. Memory optimizations are introduced where update masks are represented as binary numbers on a per-kernel-size basis instead of per-pixel basis.

5.2. Proposed Solutions:

Delta Value Propagation: Delta outputs between two frames are used as inputs as convolutional layers are linear operators. To solve the non-linear delta value update problem, the activation layer delta is calculated at iteration-time. To prevent run-off errors during truncation, update values are aggregated up to the last layer where an update had to be made. This results in update masks being generated according to the following equations:

Layer Update: $\delta_y = f(x^A + x^T + \delta_x) - f(x^A)$

Truncation Update: $x_i^A = x_{i-1}^A + x_{i-1}^T + \delta_x$

- δ_x : Input Linear Layer Delta
- δ_y : Output Linear Layer Delta
- $f()$: Non-Linear Function
- x^A : Linear Layer Accumulation
- x^T : Truncation Accumulation
- i : Frame count

GPU Design Consideration: The update mask is created for truncation based on a binary mask. A fixed tile size is used to iterate over the input to find truncation locations. If truncation is not required, then no pixels in the tile will see an update. This results in much better memory utilization. A per tile sparsity also results in significant reduction in floating point operations.

5.3. Claims and Evidence:

Claim 1: Paper proposed full-pipeline sparse calculations, including non-linear layers

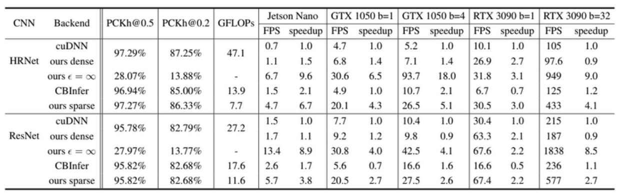
Evidence 1: Mathematics is introduced that show an approach to deal with non-linear layers in terms of maintaining delta states and managing delta accumulations.

Claim 2: GPU memory optimizations are performed to reduce required memory bandwidth

Evidence 2: Binary update masks and tile based feature map updates bring bandwidth optimizations as they result in direct reduction in memory usage and floating point operations.

Claim 3: GPU based demonstration of DeltaCNN shows 7x inference speed improvement over cuDNN.

Evidence 3: Based on the following table *Paper: 5.3 Additional Evaluations, Table 1*, we can see FPS improvement across the board on all variations of hardware and CNNs.



CNN	Backend	PCKh@0.5	PCKh@0.2	GFLOPs	Jetson Nano		GTX 1050 b=1		GTX 1050 b=4		RTX 3090 b=1		RTX 3090 b=32	
					FPS	speedup	FPS	speedup	FPS	speedup	FPS	speedup	FPS	speedup
HRNet	cuDNN	97.29%	87.25%	47.1	0.7	1.0	4.7	1.0	5.2	1.0	10.1	1.0	105	1.0
	ours dense				1.1	1.5	6.8	1.4	7.1	1.4	26.9	2.7	97.6	0.9
	ours $\epsilon = \infty$	28.07%	13.88%	-	6.7	9.6	30.6	6.5	93.7	18.0	31.8	3.1	949	9.0
	CBInfer	96.94%	85.00%	13.9	1.5	2.1	4.9	1.0	10.7	2.1	6.7	0.7	125	1.2
	ours sparse	97.27%	86.33%	7.7	4.7	6.7	20.1	4.3	26.5	5.1	30.5	3.0	433	4.1
ResNet	cuDNN	95.78%	82.79%	27.2	1.5	1.0	7.7	1.0	10.4	1.0	30.4	1.0	215	1.0
	ours dense				1.7	1.1	9.2	1.2	9.8	0.9	63.3	2.1	187	0.9
	ours $\epsilon = \infty$	27.97%	13.77%	-	13.4	8.9	30.8	4.0	42.5	4.1	67.6	2.2	1838	8.5
	CBInfer	95.82%	82.68%	17.6	2.6	1.7	5.6	0.7	16.6	1.6	16.6	0.5	236	1.1
	ours sparse	95.82%	82.68%	11.6	5.7	3.8	20.5	2.7	27.5	2.6	67.4	2.2	577	2.7

Figure 10. FPS vs. Hardware/CNN/Backend

5.4. Critique and Discussion

In general, the papers claims are valid, especially the inference speedup that was experienced using DeltaCNN. However, I would like to see additional metric comparisons with respect to memory optimization evaluations as this would strengthen the memory optimization claim. There were verbal mentions of evaluation metrics pertaining to FLOPS reduction, model stability, and implementation overhead. Again, metrics with visuals would strengthen the papers case for the claims it makes. An abaltion study could provide further context behind improvements and could potentially open up new research insights that could be taken further.

6. Implementation:

6.1. Implementation Motivation:

The contribution for this study involves re-implementation of several CNN architectures and providing some basic evaluation metrics regarding inference performance on object detection using pre-trained weights. The re-implementation focus will not be on the CNN architecture themselves, but automation of infrastructure surrounding the inference workflow. The objective is to explore modern frameworks and toolings that can provide a turnkey solution to model implementation and rapid prototyping in a production environment, as taking a model from a jupyter notebook runtime to a product integration stage is often times not straight forward due to hardware, OS, and supporting library dependencies of a production environment. In order to maintain appropriate scope for the study, the following assumptions and scoping requirements were made:

- Focus of the implementation revolves around the minimal dependency approach to implementation
- This implementation does not provide a user facing GUI other than API endpoints
- Reference to model output for implementation valida-

tion is provided. However, evaluation metrics such as mean average precision (mAP) was not calculated

- The models themselves are running in CPU only mode

Improvements to the implementation will be discussed as part of potential future work. As the implementation platform itself is highly modular, implementations of additional feature support or improvements would be a straightforward process.

6.2. Implementation Plan and Setup:

To emphasize on the point of the implementation motivation, working with software written in a research environment in terms of re-implementation of the model for product deployment or experiment verification is a difficult process. Standardization of dependencies and infrastructure in these environments can improve research reproducability, product resiliency to anomaly effects, and generally improved maintainability. Drawing from motivation again, the implementation was based on the following requirements:

- Deployment must have minimal pre-requisite dependencies
- Model deployment must be user hardware and operating system agnostic
- Model deployment must be fault tolerant to failures
- Model deployment is atomically scalable
- Application support for hot-reloading during development

Heavy emphasis was put on the dependency management, infrastructure standardization, and deployment architecture. The specific implementation of Sparse R-CNN, DETR, and Faster R-CNN will involve the use of OpenMMLab's *MMDetection* framework, which is an extension of PyTorch while introducing higher level model inference abstractions.

The COCO 2017 validation dataset, alongside its validation annotations are used as part of the inference pipeline

The framework is capable of extracting models and weights from an open source model repository and provide bounding box, object detection results, as part of its inference capability.

The executed experiment consist of the following:

1. Download defined dependencies related to infrastructure
2. Build entire software stack using one command: *tilt up*
3. Once the system is done building and all components are operational, execute a series of API calls to infer a user defined number of images from the COCO 2017 validation dataset (Lin et al., 2014)
4. User then can log into the S3 bucket deployed with the application to download images that have gone through the inference pipeline

Reduced Deployment Dependencies: One extenuating issue faced by many researchers that are trying to use previous work done, is the state of transitive dependencies used by the author at the time of implementing their contribution. An example is the author of the Sparse R-CNN architecture, used syntax and methods relevant to PyTorch 1.X, which has since been deprecated. PyTorch has since moved on to Version 2 (by conventional versioning convention, signals breaking changes to backwards compatibility) and the original implementation provided by the paper no longer is operational. In order to remediate this issue, a dependency manager can ensure the install process takes into account transitive dependencies used during development.

Platform Agnostic Deployment: In order to further reduce limitations on run-times, a declarative approach to resource management is used through the use of Kubernetes using the Docker runtime. This allows for any platform, regardless of make and hardware/software composition, can deploy the product or experiment, as long as those platforms support some form of container runtime. To adjust scope for this implementation, I will be using Kubernetes with the Docker container runtime, which is an abstraction of containerd in UNIX systems, and on the x86 cpu architecture. There is additional benefit of the fact that such declarative approach can be package with the use of Helm, allowing the scientist or developer to add additional compute, storage, or i/o dependencies with needing to consider making breaking changes to the model application.

Scalability and Fault Tolerance: When it comes to building out products, or a suite of experiments to compare research findings, scalability often comes up as a talking point. In a jupyter notebook runtime, it is difficult to build out a system that would be capable of distributed computing. It is generally ad hoc training with a testing phase. This, however does not consider the questions regarding floating point operation efficiency or perceived architectural gains in the real world, where compute resources are provisioned and deployed differently with respect to memory allocation (both VRAM and RAM) and compute allocation (distributed sharing of CUDA cores or CPU cores). Using Helm with kubernetes allows declarative compute and infrastructure scalability, whereas for the software application, a monorepo pattern distributing workload over a event-based system is implemented. Fault tolerance is built into the architecture on both the software and hardware point of view.

Detailed implementation will be shared in the implementation details section. Here is an itemized breakdown of contribution by original and supplemented:

Original Contribution

- All declarative IaC configs
- All declarative dependency management configs
- Backend distributed system (ASGI application with distributed task messeging)
- Functions defining inference tasks themselves

External Contribution

- The models themselves (Sparse R-CNN, DETR, Faster R-CNN)
- Model implementation framework (PyTorch, MMDetection)
- Server,Client Infrastructure (FastAPI, Redis, Minio)
- Transitive Dependencies (helper libraries)

6.3. Implementation Details:

There are three implementation section regarding the application:

- Ops
- Application
- Storage

The above sections of the application aims to address the design requirements mentioned previously.

Dependency Management and Infrastructure (Ops): The project started out by having maintaining reproducability and hardware/software agnostic paradigm in mind. The first item that was considered was finding a package manager for python that could supply not only a maintained graph of transitive dependencies, but maintain a lock file for all packages. This approach would guarantee all dependencies for the specific build is present so users can deploy or experiment without needing to refactor for deprecated dependencies. The tool that was used for that is **Poetry**.

Poetry allowed us to also group dependencies by function (such as task worker, inference engine, web) to optimize for the helm application (explained shortly) deployment (Poetry, 2023).

In order to have platform agnostic resource allocation, the project uses the Minikube implementation of Kubernetes, which is a runtime orchestration platform, often using the container runtime, to declaritively define infrastructure parameters. For this project, there are 2 user defined atomic systems (there are other dependent systems like the Redis task queue and minio s3 bucket for storing images), the web application and the task runners responsible for running inference tasks. Each atomic system has specified declaration about its steady state, network configuration, fault tolerance design. The kubernetes cluster controller, in this instance, Minikube, maintains a control loop to makes sure all declared configurations are being kept. This includes restarting system runtimes upon failure and applying previous state of runtime to the best of its ability. These declarations can then be packaged and installed independent of system using the helm tool.

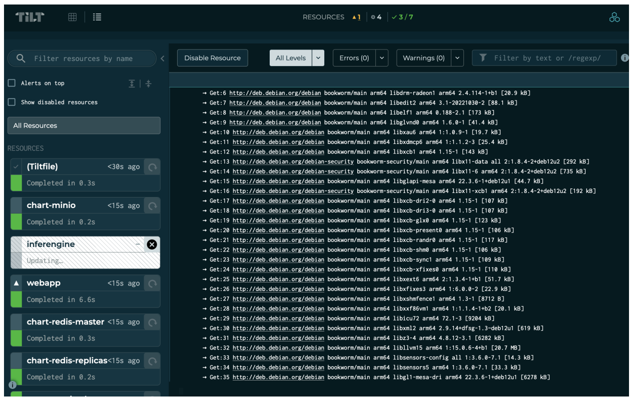
Example Helm Deployment:

```
inferengine:
  name: inferengine
  labels:
    app: cnninfer
    component: inferengine

  image:
    repository: islam25/inferengine
    pullPolicy: IfNotPresent
    # Overrides the image tag whose default is the
    tag: ""

  resources:
    limits:
      cpu: 100m
      memory: 256Mi
    requests:
      cpu: 250m
      memory: 512Mi

  autoscaling:
    enabled: true
    minReplicas: 1
    maxReplicas: 100
    targetCPUUtilizationPercentage: 50
    targetMemoryUtilizationPercentage: 50

  env:
   - name: REDIS_PASSWORD
     valueFrom:
       secretKeyRef:
        name: chart-redis
        key: redis-password
```

```
- name: MINIO_USER
  valueFrom:
   secretKeyRef:
     name: chart-minio
     key: root-user
- name: MINIO_PASSWORD
  valueFrom:
   secretKeyRef:
     name: chart-minio
     key: root-password
```

The following is a helm deployment for the **inferengine** application that acts as a task runner. The declarations include what docker image to build, hardware resources required, in what condition to scale these workers depending on per pod workload, and access information for both the Redis task queue and the minio s3 bucket.

Now that the development of these configurations are set, we can use a minikube cluster controler like **tilt** to deploy the application in one command.



```
(Tiltfile)            <30s ago
Completed in 0.3s
chart-minio           <15s ago
Completed in 0.2s
inferengine
Updating…
webapp                <15s ago
Completed in 6.6s
chart-redis-master    <15s ago
Completed in 0.3s
chart-redis-replicas<15s ago
Completed in 0.2s
```

```
→ Get:6 http://deb.debian.org/debian bookworm/main arm64 libdrm-radeon1 arm64 2.4.114-1+b1 [20.9 kB]
→ Get:7 http://deb.debian.org/debian bookworm/main arm64 libedit2 arm64 3.1-20221030-2 [88.1 kB]
→ Get:8 http://deb.debian.org/debian bookworm/main arm64 libelf1 arm64 0.188-2.1 [173 kB]
→ Get:9 http://deb.debian.org/debian bookworm/main arm64 libglvnd0 arm64 1.6.0-1 [41.4 kB]
→ Get:10 http://deb.debian.org/debian bookworm/main arm64 libxau6 arm64 1:1.0.9-1 [19.7 kB]
→ Get:11 http://deb.debian.org/debian bookworm/main arm64 libxdmcp6 arm64 1:1.1.2-3 [25.4 kB]
→ Get:12 http://deb.debian.org/debian bookworm/main arm64 libxcb1 arm64 1.15-1 [143 kB]
→ Get:13 http://deb.debian.org/debian-security bookworm-security/main arm64 libx11-data all 2:1.8.4-2+deb12u2 [292 kB]
→ Get:14 http://deb.debian.org/debian-security bookworm-security/main arm64 libx11-6 arm64 2:1.8.4-2+deb12u2 [735 kB]
→ Get:15 http://deb.debian.org/debian bookworm/main arm64 libglapi-mesa arm64 22.3.6-1+deb12u1 [44.7 kB]
→ Get:16 http://deb.debian.org/debian-security bookworm-security/main arm64 libx11-xcb1 arm64 2:1.8.4-2+deb12u2 [192 kB]
→ Get:17 http://deb.debian.org/debian bookworm/main arm64 libxcb-dri2-0 arm64 1.15-1 [107 kB]
→ Get:18 http://deb.debian.org/debian bookworm/main arm64 libxcb-dri3-0 arm64 1.15-1 [107 kB]
→ Get:19 http://deb.debian.org/debian bookworm/main arm64 libxcb-glx0 arm64 1.15-1 [123 kB]
→ Get:20 http://deb.debian.org/debian bookworm/main arm64 libxcb-present0 arm64 1.15-1 [106 kB]
→ Get:21 http://deb.debian.org/debian bookworm/main arm64 libxcb-randr0 arm64 1.15-1 [117 kB]
→ Get:22 http://deb.debian.org/debian bookworm/main arm64 libxcb-shm0 arm64 1.15-1 [106 kB]
→ Get:23 http://deb.debian.org/debian bookworm/main arm64 libxcb-sync1 arm64 1.15-1 [109 kB]
→ Get:24 http://deb.debian.org/debian bookworm/main arm64 libxcb-xfixes0 arm64 1.15-1 [110 kB]
→ Get:25 http://deb.debian.org/debian bookworm/main arm64 libxext6 arm64 2:1.3.4-1+b1 [51.7 kB]
→ Get:26 http://deb.debian.org/debian bookworm/main arm64 libxfixes3 arm64 1:6.0.0-2 [22.9 kB]
→ Get:27 http://deb.debian.org/debian bookworm/main arm64 libxshmfence1 arm64 1.3-1 [8712 B]
→ Get:28 http://deb.debian.org/debian bookworm/main arm64 libxxf86vm1 arm64 1:1.1.4-1+b2 [20.1 kB]
→ Get:29 http://deb.debian.org/debian bookworm/main arm64 libicu72 arm64 72.1-3 [9204 kB]
→ Get:30 http://deb.debian.org/debian bookworm/main arm64 libxml2 arm64 2.9.14+dfsg-1.3~deb12u1 [619 kB]
→ Get:31 http://deb.debian.org/debian bookworm/main arm64 libz3-4 arm64 4.8.12-3.1 [6282 kB]
→ Get:32 http://deb.debian.org/debian bookworm/main arm64 libllvm15 arm64 1:15.0.6-4+b1 [20.7 MB]
→ Get:33 http://deb.debian.org/debian bookworm/main arm64 libsensors-config all 1:3.6.0-7.1 [14.3 kB]
→ Get:34 http://deb.debian.org/debian bookworm/main arm64 libsensors5 arm64 1:3.6.0-7.1 [33.3 kB]
→ Get:35 http://deb.debian.org/debian bookworm/main arm64 libgl1-mesa-dri arm64 22.3.6-1+deb12u1 [6278 kB]
```

Figure 11. Tilt Build System

As a result, our approach now has tightly defined dependencies that are locked to provide consistent compatibility across a large variety of platform. On top of that, you have a fault tolerant system that can restart itself if any runtime faces hardware base anomalies and recovers.

Application Architecture (Application): The idea is to provide an application platform that will allow the use the flexibility to extend its feature set through defining declarative workloads and letting the ASGI appication in tandem with the task queue and task runners perform the task and its distribution.

From a maintainability standpoint. Strong typing is introduced through the introduction of Pydantic data transfer objects (DTO), where specific struct types can be created to express any data that may be handled at different points within the whole application space (pyd).

The approach to maintain application flexibility was to use the monorepo pattern, where each application or atomic system would be made of individual components that can be imported as a library. This way, applications such as the uvicorn ASGI application and the task runners can share the same message queue broker.

The general architecture can be visualized by the following:

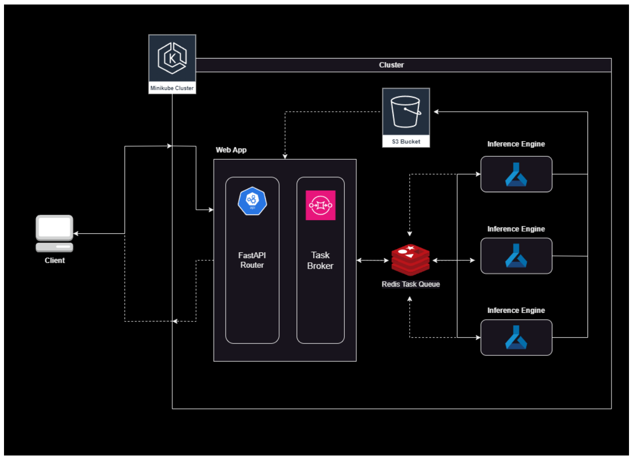


Figure 12. Application System Architecture

Note: Dotted arrows are message return arrows while solid arrows represent message task execution flow.

The application stack consists of the following:

- **Web**
  - Uvicorn - ASGI Server
  - FastAPI - Backend ASGI application
  - Redis Task Broker - Communicates with Redis Task Queue
- **Messege Broker**
  - Redis - Both functions as a task queue and messege database
- **Inference Engine**
  - MMDetection - Framework for PyToch model inferencing
  - Dramatiq - Task Executor/Runner
  - PyCOCOTools - COCO Image Repo API Wrapper

The application architecture follows the event driven asynchronous task execution pattern where each task executors role can be defined as a micro service.

An example inference pipeline would be the following:

1. **Client** accesses the api over the swagger api visualizer and executes an inference command with the number of images the client would like infered

2. The **FastAPI** application receives the request and executes the functions defined for that specific request route.

3. The **Redis Task Broker** serializes the instructions for the task that is to be executed, and sends it to the **Redis Task Queue**

4. Depending on the number of models that will be executing the inference on the group of pictures the **Inference Engine** will spawn multiple runtimes to execute these inferences in parallel fashion. These tasks are executed when **Dramatiq** picks off tasks from the queue.

5. Both the original image and the resultant bounding box images are sent to an S3 bucket for object store.

6. Finally, each inference engine will then use **Dramatiq** again to send the status of the job back through the pipeline in reverse until the ASGI application returns a success/fail code to the client.

At the current instance, the user would need to go into the ASGI application source to select which model to infer with as the service for pipeline execution based on user model input was not completed. The intuitive fix for this would be to use a specified database for persistent state storage in between tasks and not rely on global variables, which can cause the inference engines to have corrupted variable data. In addition, the bounding box results could not be saved from the dramatiq task executors as they are complex python objects that require special serializers to be formatted to json before other micro-services can use the inference response to perform evaluation calculation (dq, 2023).

Example Inference Result (Sparse R-CNN):

Figure 13. COCO Image: Before Inference

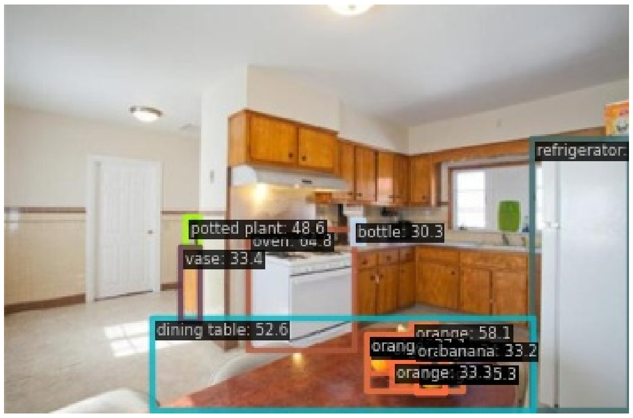


Figure 14. COCO Image: After Inference

7. Conclusion:

The deployment of the helm application on the Minikube kubernetes cluster was successfull. The application was able to resolve all transitive dependencies while maintaining compatibility with the version of tools and software used to first develop the experiment application. The application is also now capable of being deployed as a part of any kubernetes cluster and can also be used as a helm dependency on supporting applications. On top of that, the application can leverage the monorepo pattern to expand on the features available and assigning it to a respective FastAPI api route. The autoscaling feature of declarative kubernetes also allows for multiple workloads to be executed using the Dramatiq asynchronous task operator while using Redis as the task queue. This allows horizontal scalability on both hardware compute and application task distribution.

Despite the achievements of the implementation, there are clear action items that can be taken to address continued usability.

1. Correct implementation of type safety with Pydantic involves appropriate use of JSON serializer and deserializers to pass results of tasks back to intended recipients. Pydantic models themselves cannot be passed into Dramatiq return statements as the messege queue cannot support python objects. This is the main reason why evaluation metrics such as mean average precision could not be calculated as part of this implementation.
    - **Solution:** Implement serialization capabilities as part of the defined DTO models. This would allow for Pydantic models to be directly passed to and from the messege queues as part of the pipeline.
2. There was a lack of appropriate front-end application as part of the current implementation. Right now the application can be executed over the Swagger API documentation functionality. However, the user experience is not ideal.
    - **Solution:** Implement a front-end application capable of serving user appropriate experiment feature such as designing declarative experiments, which would be used to formulate the workflow.
3. This iteration of the IaC did not have correct implementation of GPU passthrough as part of the helm deployment. This forced all inferences to be performed in cpu only mode, which is not the approach to calculating true model performance.
    - **Solution:** Identify approach for NVidia helm GPU pass-through and implement it. This would allow for better variations in testing as it allows for infiniband and pcie passthrough of training data, which has much higher bandwidth for I/O operations (Nvidia, 2023).

These next steps towards improving the above implementation would further improve CNN deployment velocity, and provide an overall stable experience towards creating experiments or products and deploying them.

References

Pydantic overview. https://docs.pydantic.dev/latest/. Accessed: 2023-11-09.

Infrastructure as code, 2022. URL https://www.redhat.com/en/topics/automation/what-is-infrastructure-as-code-iac.

Dramatiq. https://dramatiq.io/cookbook.html, 2023. Accessed: 2023-11-10.

Chaoyang He, Murali Annavaram, S. A. Group knowledge transfer: Federated learning of large cnns at the edge. In *Proceedings of the Neural Information Processing System (NeuralIPS 2020)*, Vancouver, CA, 2020. IEEE.

Lin, T., Maire, M., Belongie, S. J., Bourdev, L. D., Girshick, R. B., Hays, J., Perona, P., Ramanan, D., Dollár, P., and Zitnick, C. L. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014. URL http://arxiv.org/abs/1405.0312.

Mathias Parger, Chengcheng Tang, C. D. T. C. K. R. W. M. S. Deltacnn: End-to-end cnn inference of sparse frame differences in videos. In *Proceedings of the Computer Vision and Pattern Recognition (CVPR 2022)*, New Orleans, LA, 2022. IEEE.

Nvidia, C. NVidia kubernetes gpu operator. 2023. URL https://docs.nvidia.com/datacenter/cloud-native/gpu-operator/23.9.0/index.html.

Peize Sun, e. a. Sparse r-cnn: End-to-end object detection with learnable proposals. In *Proceedings of the Computer Vision and Pattern Recognition (CVPR 2021)*, Nashville, TN, 2021. IEEE.

Poetry, C. Poetry project, 2023. URL https://python-poetry.org/docs/.