THE OPTIMIZATION OF A DEEP LEARNING BASED ARCHITECTURE FOR OBJECT DETECTION WITH LIMITED COMPUTATIONAL RESOURCES

A Thesis

Presented to the

Department of Computer Science

and the

Faculty of the Graduate College

University of Nebraska

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

University of Nebraska at Omaha

by

Trever Anderson

December 2019

Supervisory Committee:

Pei-Chi Huang

Myoungkyu Song

Rui Zhao

THE OPTIMIZATION OF A DEEP LEARNING BASED ARCHITECTURE FOR OBJECT DETECTION WITH LIMITED COMPUTATIONAL RESOURCES

Trever Anderson, MS

University of Nebraska, 2019

Advisor: Pei-Chi Huang

The abundance of Deep Convolutional Neural Networks for object detection necessitates the use of taxonomies when choosing a viable network for a given set of computational constraints. One of the fundamental methods of classification is the distinction between one stage and two stage detection networks or architectures. In general, one stage architectures are considered to be faster but less accurate than two stage architectures because of the absence of a region proposal component. With limited computational resources, a one stage architecture would therefore serve as a better baseline for accuracy and speed improvements. We surveyed the research and chose five models to participate in a comparison where four of those models are one stage architectures. Network components, intricacies, similarities, and runtime performance are compared to show the improvements in an objective and measurable manner. We introduce a different approach for testing runtime performance where standalone model implementations and a small common library take the place of a unified detection framework. This approach alleviates the need of model reimplementation and is more flexible. We conclude with a discussion of runtime performance against official accuracy for all architectures.

Contents

1 Introduction

Object detection has attracted the attention of many researchers over the past seven years with over 5,000 publications [3]. This proliferation in large part can be attributed to the excessive interest in the Deep Convolutional Neural Network (DCNN). The maturation from a simple feature extractor [1] to a taxonomy of architectures that are applicable across numerous computational contexts suggests that the DCNN is capable of adapting to change, and thus future proof to some degree. The artifacts of this maturation were grouped by commonality, and thus one stage and two stage detectors [3], [7] became canonical categories for meta-architectures. Given a detector, it is considered two stage if it performs classification and regression on specific areas of interest generated by an upstream component in the network. If the detector lacks this upstream component and instead draws static box grids that exhaustively cover one or more upstream layers, then it is considered one stage. One of the most popular one stage detectors is the Single Shot Multibox Detector (SSD) [4] which classifies and localizes objects by using static anchor boxes placed at multiple feature map scales. On the other hand, for two stage detectors, Region-based Convolutional Neural Networks (R-CNN) [2], widely known as the first incarnation in the hugely successful R-CNN, Fast-RCNN [5], Faster-RCNN [6] line of DCNNs, provided the first results of convolving region proposals over convolutional blocks. The subsequent improvements incrementally changed parts of R-CNN to convolutional blocks until the entire network was fully convolutional and could be trained end-to-end by a given backpropagation implementation. Faster-RCNN is now considered a canonical two stage detector and a great starting point for proposal based detection. The reason for the architectural divergence between networks like SSD and Faster-RCNN is applicability in a given computational context. As far as this research knows, no single architecture is applicable across all computing contexts through hyperparameter adjustments alone, which justifies the comparison herein.

2 Related Work

2.1 Region Proposals and Detector Performance

Huang et. al. [7] showed on an even playing field that among the Faster-RCNN, Region-based Fully Convolutional Networks (R-FCN) [9], and SSD meta-architectures, SSD with MobileNet-V1 [12] is the fastest model combination. Even with hyperparameters adjusted for speed, Faster-RCNN and R-FCN are slower than SSD for all feature extractors in terms of inference time. Inference time is measured by including all pre and post processing when performing prediction on a given image [7].

Table 1 suggests that trading accuracy for speed with two stage detectors does not achieve one stage detector performance. This is because they must overcome the computational overhead of region proposals to compete with one stage detectors. Huang et. al. [7] does not provide accuracy numbers for R-FCN and Faster-RCNN when region proposals are lower than 10 as 15 AP is a low score for a two stage detector. Huang et. al. [7] provide more accuracy values for R-FCN and Faster-RCNN with different feature extractors and the result is a higher inference time because MobileNet-V1 is the smallest feature extractor in the comparison. Table 1 serves as evidence to the conclusion that

Type	Architecture	Feature Extractor	Region Proposals	AP	Time (ms)
One Stage	SSD	MobileNet-V1	N/A	19.3	40
One Stage	SSD	MobileNet-V2 [21]	N/A	22	50
One Stage	SSD	ResNet-101	N/A	24	60
Two Stage	R-FCN	MobileNet-V1	10	15	80
Two Stage	Faster-RCNN	MobileNet-V1	10	15	80
Two Stage	R-FCN	MobileNet-V1	50	17	80
Two Stage	Faster-RCNN	MobileNet-V1	50	19	90
Two Stage	R-FCN	MobileNet-V1	300	16	90
Two Stage	Faster-RCNN	MobileNet-V1	300	20	110

Table 1: Sorted by region proposals. To produce a respectable accuracy, the minimum of region proposals is 10. R-FCN and Faster-RCNN are not as accurate as SSD when using 10 region proposals and inference time only grows when more region proposals are used. Switching feature extractors for R-FCN and Faster-RCNN only produces higher inference times, such as Inception [21]. All numbers were derived from the same COCO [16] dataset split [7].

it is better to optimize accuracy in a one stage architecture than to optimize inference time in a two stage detector via region proposal reduction.

2.2 Model Comparison

To ensure a valid model comparison, Huang et al. [7] went to great lengths by building a unified detection framework with tensorflow's [8] python API. Committed under the research folder in the official tensorflow models repository, this framework or detection pipeline utilizes tensorflow's estimators framework [10] and features implementations for Faster-RCNN, SSD, and R-FCN, several data augmentation techniques, and multiple feature extractors. It supports multiple datasets, dynamic configuration via protobuf files, tensorboard integration, a model zoo [14], and simple extension by implementing novel base classes. This research did not use Huang et al.'s detection pipeline [7] because it did not possess the man power, time, or computational resources to port novel model implementations. Each model would have taken months to port, train, and achieve official model accuracy. Instead, a collection of standalone implementations serves in place of a unified detection framework.

3 Model Comparison

In this section, we compare the single shot multibox detector (SSD), RetinaNet, CornerNet, CornerNet-Squeeze, CornerNet-Saccade, their architectures and loss functions, and performance and adjustments.

3.1 Single Shot Multibox Detector (SSD)

SSD structure

In a canonical implementation of SSD, Oxford's renowned Visual Geometry Group (VGG) [15] feature extractor serves as the backbone. A few of its final layers are inputs to convolutional layers that regress and classify bounding box locations. Extra feature map layers can be added to the end of VGG and these layers can serve as inputs to

Architecture	Dataset	AP
Fast[5]	train	19.7
Fast[22]	train	20.5
Faster[6]	trainval	21.9
ION[22]	train	23.6
Faster[31]	trainval	24.2
SSD300[4]	trainval35k	23.2
SSD512[4]	trainval35k	26.8

Table 2: Comparison between SSD with Fast-RCNN, Faster-RCNN, and ION on the COCO dataset.

the regression and classification layers. Since feature maps are successively reduced in spatial dimensions, SSD performs classification and regression at multiple feature map scales. The number of VGG layers and extra layers for classification and regression is configurable as is the feature extractor itself. The classification and regression convolutional layers serve as input to the loss function.

SSD Loss

SSD places static anchor boxes on every feature map that is configured to serve as a basis for classification and regression. These feature maps are tiled with static anchor boxes such that each location has six of them [4]. Since there are thousands of anchor boxes needed to cover these feature maps exhaustively, a matching algorithm filters anchor boxes for a given example's loss calculation. The union of the set of anchor boxes that had the highest Intersection over Union (IOU) for a given ground truth and the set of anchor boxes that had an IOU higher than some threshold with at least one ground truth forms the set of anchor boxes used in computing loss for a given example. Liu et al. [4] used a matching threshold of .5. If SSD only used anchor boxes with the highest IOU overlap for a given ground truth, then the loss target would be smaller and produce less accurate predictions. The inclusion of the threshold match anchors provides a reasonable estimate for box regression. All matched anchors are used in regression loss, L_{loc}, such that smooth L1 loss is computed between the network's prediction and the offset between a matched anchor and its groundtruth. Despite filtering the initial set of anchors with matching, the result is a set of anchor boxes where most are negatives. Hard negative mining filters the anchors and enforces a negative to positive ratio of 3:1. Liu et al. [4] found that this ratio leads to faster convergence and stabilizes training. The smooth L1 loss is added to the softmax loss, L_{conf}, to obtain the total loss value. Please refer to Liu et. al. [4] equations 1, 2, and 3 for the specific equations.

$$L = L_{conf} + L_{loc} \tag{1}$$

SSD Performance and Adjustments

SSD is one of the most popular one stage detection architectures because its simple and flexible design allows it to achieve respectable performance numbers with a variety of configurations. Table 2 shows a comparison between SSD at square resolutions of 300 and 512, Fast-RCNN, Faster-RCNN, and Inside-Outside Net (ION) [22]. This table

Architecture	FPS	batch size	Boxes	Input resolution
SSD300[4]	46	1	8732	300x300
SSD512[4]	19	1	24564	512x512
SSD300[4]	59	8	8732	300x300
SSD512[4]	22	8	24564	512x512

Table 3: Comparison between SSD architectures with different batch sizes, boxes, and input resolutions. An increase in resolution increases the number of anchor boxes which decreases FPS.

shows that the SSD architecture can scale with respect to image resolution on the COCO dataset. Table 3 shows that as image resolution is increased, the number of anchor boxes is increased to cover the larger feature maps which decreases frames per second. Liu et al. [4] note that the box tiling scheme can be adjusted or replaced by a different scheme that is more advanced. The number of boxes at each feature map location, the aspect ratios of each box, and the scale of the boxes can be altered to produce different performance results [17].

3.2 RetinaNet

RetinaNet Structure
RetinaNet's backbone network consists of ResNet followed by Feature Pyramid Network (FPN) [13], where ResNet performs the relatively simple task of providing the initial feature map representations. FPN accepts these representations as input and further enhances them to provide a basis for object detection and classification. At each spatial location of each FPN level, static anchor boxes are drawn at multiple scales and different aspect ratios to provide a target when computing regression and classification loss. A classification subnet is attached to each FPN level to provide class predictions where its parameters are shared across all FPN levels. Similarly, a regression subnet is attached to each FPN level and its parameters are shared across all FPN levels. Parameters are not shared between the classification and regression subnets and each serve as an input to the loss function. At inference time, the predictions are fed into non maximum suppression to obtain the final predictions.

RetinaNet Loss
The discovery of class imbalance producing undesirable loss quantities and subsequently dominating the gradient provoked Lin et al. [17] to introduce a new cross entropy loss called focal loss, given by $FL(p_t)$. Instead of using hard example mining [4] or some other variant, Lin et al. designed focal loss to progressively reward the network for correctly classified examples and to include all examples in loss calculation. As the prediction for a given example increases and approaches one, the modulating factor reduces the loss of the $log(p_t)$ term by a factor of $(1 - p_t)^y$, where p_t is the prediction and y is the focusing parameter. Misclassified examples are nearly unaffected as the modulating factor is close to one. With focal loss, RetinaNet is able to bypass the need for anchor box filtering and includes all anchors in its loss calculation.

$$FL(p_t) = -(1 - pt)^y log(p_t) \tag{2}$$

Depth	Scale	AP	Time (ms)
50	400	30.5	64
50	500	32.5	72
50	600	34.3	98
50	700	35.1	121
50	800	35.7	153
101	400	31.9	81
101	500	34.4	90
101	600	36.0	122
101	700	37.1	154
101	800	37.8	198

Table 4.a: Comparison of RetinaNet on the COCO dataset with ResNet-50 and ResNet-101 backbones. $y = 2$; 9 anchors per feature map location over 3 scales and 3 aspect ratios. Accuracy numbers were obtained from the test-dev set. Increases in image scale correlate with increases in prediction time (ms). ResNet-101 provides less of an accuracy increase over ResNet-50 for alike image scales vs. increasing image scale [17].

Method	Batch size	NMS thr	AP
OHEM	128	.7	31.1
OHEM	256	.7	31.8
OHEM	512	.7	30.6
OHEM	128	.5	32.8
OHEM	256	.5	31.0
OHEM	512	.5	27.6
OHEM 1:3	128	.5	31.1
OHEM 1:3	256	.5	28.3
OHEM 1:3	512	.5	24.0
FL	n/a	n/a	36.0

Table 4.b: Online Hard Example Mining (OHEM) [18], OHEM (1:3) [4], and Focal Loss (FL) are compared with focal loss on the COCO dataset. $y = 2$; 9 anchors per feature map location over 3 scales and 3 aspect ratios; 600 image scale; ResNet-101 with FPN [17].

Each anchor is assigned to at most one ground truth box if the IOU overlap is greater than or equal to 0.5 [17]. This assignment means that the anchor is considered positive and its corresponding class vector is assigned a one at the index representing the class of the anchor's assigned ground truth box. An anchor is assigned background and considered negative if its IOU overlap is between [0, 0.4). All anchors with overlap [0.4,0.5) are ignored during training. Smooth L_1 loss is used on the network's predictions and the offsets between the anchor boxes and their assigned ground truths. The total loss is the sum of the focal loss and regression loss.

RetinaNet Performance and Adjustments

Table 4.a shows that RetinaNet's accuracy is more affected by changes in image scale than changes in network complexity. ResNet-101 beats ResNet-50 in AP when comparing alike image scales with the biggest difference being at scale 800 where ResNet-101 has a 2.1 AP advantage over ResNet-50. This increase is minimal compared to the AP increase for a given backbone over multiple image scales. ResNet-50 receives a 5.2 AP increase when comparing image scales 800 and 400 and ResNet-101 receives a 5.9 AP increase over the same two image scales. Prediction time is similarly more affected by an increase in image scale than an increase in network complexity. When comparing alike image scales, the largest increase between ResNet-50 and ResNet-101 is scale 800 where ResNet-101 takes 45ms more per image. This difference is small when compared to an increase in image scale for a given ResNet configuration. ResNet-50 experiences an increase of 89ms when going from scale 400 to scale 800. For the same scales, ResNet-101's prediction time increases 117ms. Thus, increases in image scale affect RetinaNet's

α	AP
.10	0.0
.25	10.8
.50	30.2
.75	31.1
.90	30.8
.99	28.7
.999	25.1

Table 4.c: Accuracy values for cross entropy loss with a weighting factor α and $\gamma = 0$. When $\gamma = 0$, focal loss is equivalent to cross entropy loss.

γ	α	AP
0	.75	31.1
0.1	.75	31.4
0.2	.75	31.9
0.5	.50	32.9
1.0	.25	33.7
2.0	.25	34.0
5.0	.25	32.2

Table 4.d: Comparison of focal loss with different values of γ and optimal values for α[17].

#Sc	#AR	AP
1	1	30.3
2	1	31.9
3	1	31.8
1	3	32.4
2	3	34.2
3	3	34.0
4	3	33.8

Table 4.e: Comparison of different scales and aspect ratios on the COCO dataset. $\gamma = 2$; ResNet-50 feature extractor; 600 image scale. Multiplication of #sc and #ar is the number of anchor boxes at each feature map location. Saturation occurs after two scales and three aspect ratios [17].

performance more than increases in network complexity in terms of prediction time and accuracy. Table 4.b proves that focal loss is a more innovative approach than Online Hard Example Mining (OHEM) [18] and its variant OHEM 1:3 [4]. By including all examples in loss calculation, focal loss achieves the highest AP of 36. Tables 4.c and 4.d show vanilla cross entropy loss with a weighting factor α and focal loss with different values of γ respectively. The difference between the best setting of γ and α in Table 4.d and the best setting of α in Table 4.c, 2.9 AP, shows the importance of focal loss and its modulating factor in producing higher AP scores. Table 4.e shows a sweet spot at two scales and three aspect ratios for anchor boxes per FPN feature map location. One scale and one aspect ratio produces a respectable AP of 30.3. Naively adding more anchor boxes at each feature map location will not yield higher AP as a point of saturation exists after two scales and three aspect ratios.

3.3 CornerNet

CornerNet Structure

CornerNet is different from conventional one stage architectures, such as SSD, in that it does not use anchor boxes and multiple feature map scales for prediction. Its backbone network, Hourglass [20], was designed for human pose estimation, but Law et al. [19] repurposed Hourglass for object detection by constructing a bounding box as a pair of top left and bottom right points. The structure of Hourglass is a composition of hourglass modules. An hourglass module is a composition of residual modules. A residual module consists of a series of three convolutions and a skip connection and is heavily influenced by the original residual block [11]. Please refer to Law et. al.'s work [19] for a pictorial representation of these modules. The output of each hourglass

module serves as a basis for prediction and loss computation, where Newell et al. [20] call this intermediate supervision. Law et al. [19] decided to use two hourglass modules which serve as input to four prediction modules, that is, two prediction modules per hourglass module output. Each prediction module contains convolutional structures for corner pooling, heatmaps, embeddings and offsets. The latter three serve as input to three functions whose sum is the total loss.

CornerNet Loss

Hourglass feature map outputs are passed through the corner pooling layer to determine whether a given location is a bottom right or top left corner. These feature maps serve as the basis for loss calculation of heatmaps, embeddings, and offsets. Heatmaps punish the network via a two dimensional gaussian where the center is the positive location and the surrounding locations are negative. The network is progressively rewarded by pushing up its prediction on the positive location similar to RetinaNet's focal loss. The negative locations punish the network by taking a percentage of the networks prediction multiplied by the remaining loss of its prediction. That percentage is high for negatives around the edge of the gaussian and decreases towards the center. To keep track of pairs of points, Law et al. [19] borrow Associative Embedding from Newell et al. [20] and assign embeddings to each point in each pair as part of example creation. The pull and push loss force the network to move predicted points such that their embedding distance approaches the distance of the ground truth embeddings. Offset loss is necessitated by the loss of precision when mapping points from the input image to the heatmaps. The ratio of the heatmap size to the input size is used to scale a given (x,y) input location to a heatmap location which produces a real valued number for x and y. Law et al. [19] use the floor function to estimate the mapped location and the difference between the real valued location and estimated location is the target for offset loss. The offset loss is then the Smooth L1 loss between the network's offset prediction and the aforementioned difference. The heatmap, pull, push, and offset loss are L_{det}, L_{pull}, L_{push}, and L_{off} respectively. α, β, γ are weights of L_{pull}, L_{push}, and L_{off} respectively. The summation of the L_{det}, L_{pull}, L_{push}, and L_{off} loss values is the total loss. Please refer to Law et. al. [19] equations 1, 2, 3, 4, 5, 6, 7, and 8 for more information.

$$L = L_{det} + \alpha L_{pull} + \beta L_{push} + \gamma L_{off} \tag{3}$$

CornerNet Performance and Adjustments

Table 5 shows that corner pooling has a discernible effect on CornerNet's performance. The ablation study in Table 6 examined the effects of removing the reduction penalty in the two dimensional gaussian heatmaps. Without a penalty reduction, the optimization problem is more difficult and AP drops 5.5 points. Law et al. [19] use a dynamic radius to define the size of the heatmaps and when it is swapped for a fixed radius AP drops. We assume penalty reduction was applied in this experiment to illustrate the difference between a fixed and dynamic radius. The AP of small objects is less affected compared to overall AP, medium, and large object AP. In Table 7, all accuracy measurements show a significant drop when Hourglass is swapped for ResNet-101 and FPN. Accuracy is increased at all levels when Hourglass is combined with an anchor box based meta-architecture similar to Lin et al. [17], but it is less accurate for all

	AP	AP^{50}	AP^{75}	AP^s	AP^m	AP^l
w/o corner pooling	36.5	52.0	38.8	17.5	38.9	49.4
w/ corner pooling	38.4	53.8	40.9	18.6	40.5	51.8

Table 5: Ablation study on the effect of corner pooling on the COCO dataset [19].

	AP	AP^{50}	AP^{75}	AP^s	AP^m	AP^l
w/o reducing penalty	32.9	49.1	34.8	19.0	37.0	40.7
fixed radius	35.6	52.5	37.7	18.7	38.5	46.0
object-dependent radius	38.4	53.8	40.9	18.6	40.5	51.8

Table 6: Ablation study on reduction penalty in the heatmaps. Experiments were performed on the COCO dataset [19].

measurements than CornerNet with Hourglass. This disparity in accuracy suggests that CornerNet is heavily dependent on choice of feature extractor [19] and that anchor box based meta-architectures do not take advantage of Hourglass features as well as key point based meta-architectures.

3.4 CornerNet-Lite

CornerNet-Lite improved the performance of CornerNet by reducing the amount of pixels to process and reducing the amount of processing per pixel. To achieve the former, CornerNet-Saccade predicts regions of interest for the subsequent layers to further examine. To achieve the latter, CornerNet-Squeeze incorporated ideas from MobileNet-V1 [12] and SqueezeNet [24] to reduce the complexity of the Hourglass feature extractor [20]. These improvements together are referred to as CornerNet-Lite, but both are separate networks where CornerNet-Saccade is a two stage detector and CornerNet-Squeeze is a one stage detector.

CornerNet-Saccade
CornerNet's computationally expensive backbone and one stage detection architecture are the culprits in its high inference time of over one second per image [23]. To preserve CornerNet's high accuracy while increasing its efficiency, Law et. al. [23] turned CornerNet into a two stage detector and split the duties of Hourglass. The first Hourglass module predicts three attention maps for detection of small, medium, and large objects respectively. Saccades are generated from the attention maps and serve as input to the second hourglass which performs classification and regression to obtain the final detections.

	AP	AP^{50}	AP^{75}	AP^s	AP^m	AP^l
FPN (w/ ResNet-101) + Corners	30.2	44.1	32.0	13.3	33.3	42.7
Hourglass + Anchors	32.9	53.1	35.6	16.5	38.5	45.0
Hourglass + Corners	38.4	53.8	40.9	18.6	40.5	51.8

Table 7: Comparison of CornerNet with ResNet-101 backbone and Hourglass network with anchor box based prediction. Experiments were performed on the COCO dataset [19].

CornerNet-Squeeze
Law et. al. [23] noted that CornerNet spends most of its computational resources in Hourglass-104. To reduce the complexity of Hourglass-104, the residual modules are replaced by SqueezeNet's fire module [24] and the standard 3x3 convolution in the fire module is replaced by MobileNet-V1's [12] depth wise separable convolution. This new Fire Module is the building block of the reduced Hourglass backbone which has far fewer parameters than its CornerNet [19] counterpart as Table 9 illustrates. CornerNet-Squeeze uses the same loss and hyperparameters found in CornerNet [23].

CornerNet-Saccade and CornerNet-Squeeze Performance
Law et. al. [23] included an inference time and accuracy comparison between CornerNet-Saccade, CornerNet-Squeeze, YOLOv3, RetinaNet, and CornerNet. The comparison between YOLOv3 and CornerNet-Squeeze was tilted in favor of YOLOv3 because it is implemented in C and CornerNet-Squeeze is implemented in Python. Despite this advantage, CornerNet-Squeeze outperforms YOLOv3 at all measured accuracy points. CornerNet-Squeeze also beats RetinaNet until accuracy climbs over 36, where the accuracy of CornerNet-Squeeze saturates and RetinaNet is able to show gains in accuracy for trades in inference time [23]. CornerNet-Saccade appears to outmatch RetinaNet at all measured accuracy points and achieves the highest AP of the group at 43.2 AP and 190ms inference time.

4 Frames Per Second (FPS) comparison

Table 8 illustrates an FPS comparison between SSD, CornerNet, CornerNet-Saccade, CornerNet-Squeeze and RetinaNet. The models were tested on a system with an Intel core i5 4670K CPU, RTX 2070 GPU, and 16 GB of RAM. Instead of using a unified detection architecture [7], this research developed an ad hoc architecture where each model implementation is its own standalone project. This approach made the comparison feasible, as porting CornerNet, CornerNet-Saccade, CornerNet-Squeeze, and RetinaNet to Huang et. al [7]'s architecture is simply too much work. Each project has its own directory, and each directory has a setup and FPS script. The BASH setup script clones a project and sets up an environment to support inference mode execution. The Python FPS script instantiates pretrained models and uses the common library to compute the FPS measurement. All FPS scripts in the system compute FPS the same way by passing the common library an inference function. The common library times this inference function and divides the result by the size of the batch. FPS numbers are kept to two decimals to break ties between models.
The PyTorch [27] SSD implementation chosen for this experiment is not the official implementation and only supports image sizes of 300 because its anchor computation uses static values. The official SSD implementation was not used because it is implemented in Caffe [26] and CornerNet is implemented in PyTorch [27]. To achieve as fair a comparison as possible, this research chose the PyTorch SSD implementation. Table 8 shows that PyTorch SSD achieves similar numbers to the results Liu et al. [4] achieved in Table 3, and it shows that SSD achieved the highest FPS measurement of 80.61. The RTX-2070 could not handle batch sizes higher than 45 with PyTorch SSD.

CornerNet-Lite[23] introduced CornerNet-Saccade and CornerNet-Squeeze. Its official implementation project, CornerNet-Lite, has implementations for CornerNet-Saccade, CornerNet-Squeeze, and the original CornerNet. The CornerNet-Lite project is a better implementation than the original CornerNet project because it is simpler to use and more flexible. The CornerNet-Lite project was used to measure the FPS of all the CornerNet based models. One drawback of this implementation is that it does not support batch inference, which explains the absence of higher batch sizes with the CornerNet based models.

This research could not find a viable RetinaNet implementation written in PyTorch. The official RetinaNet implementation is apart of Facebook's unified detection architecture, Detectron [28], and thus could not be integrated as easily as the other standalone implementations. A standalone Keras implementation was chosen instead and it produced interesting results. Table 8 shows that Keras RetinaNet can process batch sizes of 256, whereas PyTorch produced out of memory errors at batch size 45. This research hypothesizes that Keras is capable of paging a large tensor into GPU memory rather than copying the entire tensor as PyTorch does, and thus can process arbitrarily large tensors. This implementation feature is useful and shows that RetinaNet can compete with SSD when the batch size is configured appropriately.

Table 8 shows that batch size has a significant effect on FPS. As batch sizes increases, the speed of the model increases. This is probably due to taking advantage of readily available GPU memory, rather than only using a smaller portion for one image. The batch size, however, can be too large and adversely affect performance. The CornerNet-Lite project does not support batch inference and the CornerNet based models suffer from this drawback. At batch size 1, CornerNet-Squeeze is competitive with RetinaNet, but as batch size increases RetinaNet pulls away rather swiftly.

A more obvious factor in model speed is the size of the image. Table 8 shows that as image size increases, model speed decreases. This research did not find it necessary to test for higher image sizes as the pattern can be deduced from Table 8. CornerNet-Saccade, however, produced an interesting result. Law et al. [23] found that CornerNet spent too much time in Hourglass and proposed CornerNet-Saccade to decrease inference time by adding a region proposal component to suggest regions of interest to the classification and regression layers. Thus, CornerNet-Saccade processes less pixels than CornerNet [23]. Table 8 shows that CornerNet-Saccade is less than 1 FPS faster than CornerNet at images sizes 300 and 512. This result is consistent with Law et. al [23] because CornerNet-Saccade was 21 ms faster than CornerNet in their comparison.

There seems to be no correlation between the number of parameters and detector performance. Table 8 shows an interesting case where CornerNet-Squeeze is slower than RetinaNet at batch size 1 for all image sizes, but Table 9 shows that CornerNet-Squeeze has roughly 6.3 million fewer parameters than RetinaNet. SSD has roughly 5 million fewer parameters than CornerNet-Squeeze and outperforms it by over 20 FPS at batch size 1 and image size 300. CornerNet-Saccade has roughly 85 million fewer parameters than CornerNet, but is faster by less than 1 FPS. Thus, performance is probably more affected by the quality of the implementation of the inference function, model architecture, and the necessity of compute intensive interpretation logic than number of parameters alone.

Architecture	Backbone	FPS	Batch size	Image size	Framework
SSD	VGG-16	48.64	1	300x300	PyTorch
SSD	VGG-16	64.26	8	300x300	PyTorch
SSD	VGG-16	79.41	16	300x300	PyTorch
SSD	VGG-16	80.61	32	300x300	PyTorch
SSD	VGG-16	66.65	45	300x300	PyTorch
CornerNet	Hourglass	3.04	1	300x300	PyTorch
CornerNet-Saccade	Hourglass	3.68	1	300x300	PyTorch
CornerNet-Squeeze	Hourglass	26.92	1	300x300	PyTorch
RetinaNet	ResNet-50	33.61	1	300x300	Keras
RetinaNet	ResNet-50	58.93	8	300x300	Keras
RetinaNet	ResNet-50	69.00	16	300x300	Keras
RetinaNet	ResNet-50	72.97	32	300x300	Keras
RetinaNet	ResNet-50	74.04	64	300x300	Keras
RetinaNet	ResNet-50	69.56	128	300x300	Keras
RetinaNet	ResNet-50	68.88	256	300x300	Keras
CornerNet	Hourglass	3.01	1	512x512	PyTorch
CornerNet-Saccade	Hourglass	3.58	1	512x512	PyTorch
CornerNet-Squeeze	Hourglass	24.40	1	512x512	PyTorch
RetinaNet	ResNet-50	26.04	1	512x512	Keras
RetinaNet	ResNet-50	36.45	8	512x512	Keras
RetinaNet	ResNet-50	42.20	16	512x512	Keras
RetinaNet	ResNet-50	23.32	32	512x512	Keras

Table 8: FPS measurement of SSD, CornerNet, CornerNet-Saccade, CornerNet-Squeeze and RetinaNet over different image sizes and batch sizes. All models were tested 10 iterations over a 600 image COCO dataset. Each iteration yields an FPS measurement, and those measurements were averaged over 10 iterations. For batch sizes greater than 1, the time taken by the inference function call is divided by the batch size.

5 Conclusion and Future Work

This research focused on the runtime performance of DCNNs and how it is affected by model architecture, inference function implementation, image size, and runtime configuration. SSD [4], RetinaNet [17], CornerNet [19], CornerNet-Saccade [23], and CornerNet-Squeeze [23] were compared in terms of their architectures, loss functions, and prediction performance. A simple framework was created to facilitate a performance comparison where standalone model implementations were tied together via a common library. This approach differs from unified detection architectures [7], [28] in that it allows for different frameworks to be compared, eliminates the need for reimplementations, and reduces the overall amount of code. Table 10 and Figure 1 work together in showing the FPS and official accuracy of each model in the comparison. CornerNet-Squeeze is the most balanced model, SSD the fastest, and CornerNet-Saccade the slowest but most accurate. There is still room for more models to be analyzed in future comparisons, such as RefineDet [29], CenterNet [30], and YOLO [25]. RefineDet [29] is a one stage detector that performs two stages of anchor refinement. CenterNet builds on Law et.

Architecture	Backbone	Parameters
SSD	VGG-16	26,285,486
CornerNet-Squeeze	Hourglass	31,771,852
RetinaNet	ResNet-50	38,021,812
CornerNet-Saccade	Hourglass	116,969,339
CornerNet	Hourglass	201,035,212

Table 9: Model complexity comparison sorted by number of parameters. All parameters are included, that is, trainable and non trainable.

Architecture	Backbone	Image size	Multi scale	AP
SSD	VGG-16	300x300	no	25.1 [4]
SSD	VGG-16	512x512	no	28.8 [4]
CornerNet	Hourglass-104	511x511	no	40.6 [19]
CornerNet	Hourglass-104	511x511	yes	42.2 [19]
CornerNet-Saccade	Hourglass-54	511x511	no	43.2 [23]
CornerNet-Squeeze	Hourglass	408x408	no	34.4 [23]
CornerNet-Squeeze	Hourglass	511x511	yes	37.1 [23]
RetinaNet	ResNet-50	400x400	no	30.5 [17]
RetinaNet	ResNet-50	500x500	no	32.5 [17]
RetinaNet	ResNet-50	600x600	no	34.3 [17]
RetinaNet	ResNet-50	700x700	no	35.1 [17]
RetinaNet	ResNet-50	800x800	no	35.7 [17]

Table 10: Accuracy of all models in comparison. SSD at image sizes 300 and 512 benefited from an extra data augmentation technique where the image was expanded during training [4], thereby increasing AP from Table 2. CornerNet-Squeeze's image size was calculated by 511 * 0.8 as Law et. al. [23] show in Figure 11 that CornerNet-Squeeze obtains its best results at this image scale.

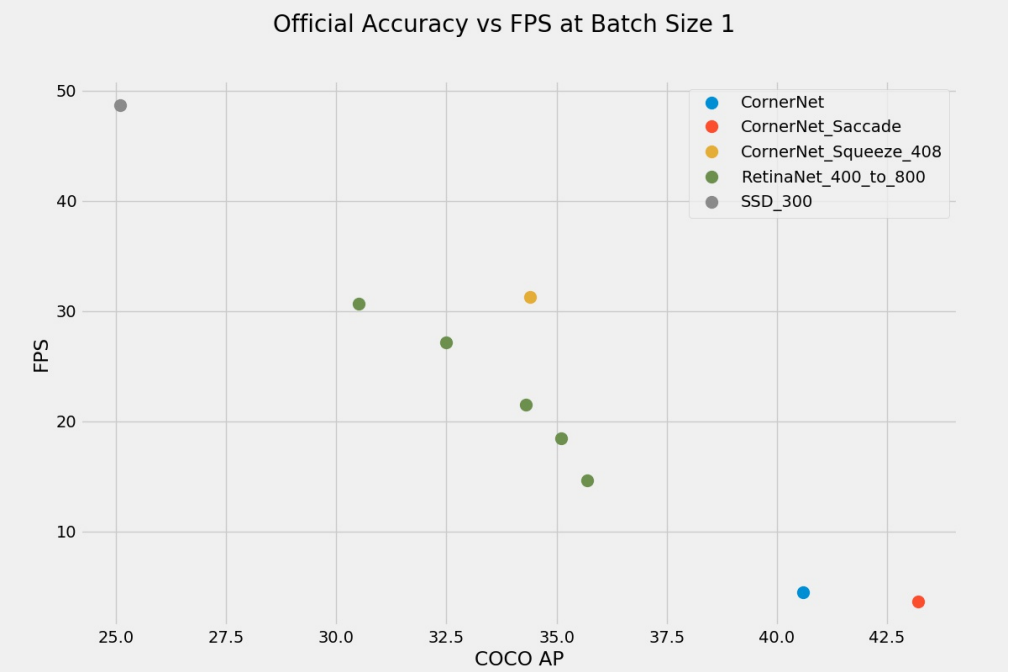

Figure 1: FPS numbers were computed for the models in Table 10 at the official image sizes. The number after the underscore in the network name represents the image size. The fastest RetinaNet point was at image size 400 and the slowest at 800. CornerNet beats CornerNet_Saccade by roughly .9 FPS.

al. [19] by adding two more points to localize the center of a bounding box. YOLO is a single stage detector implemented in C that uses DarkNet, a novel backbone, for feature extraction. Also, memory usage can be added to be measured in this comparison. This measurement could help in classifying the runtime performance of a model. With an ad hoc comparison approach and different implementation frameworks, memory usage might be less reliable than a unified detection architecture [7], [28] because Table 8 showed that unlike Keras, PyTorch does not page tensors into GPU memory. This problem can be mitigated to some extent by matching implementation frameworks, but implementation quality is still of concern.

6 References

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In NIPS, 2012.

[2] R. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. arXiv preprint arXiv:1311.2524, 2013.

[3] Z. Zou, Z. Shi, Y. Guo, and J. Ye. Object Detection in 20 Years: A Survey. arXiv:1905.05055, 2019.

[4] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, and S. Reed. SSD: Single shot multibox detector. arXiv:1512.02325, 2015.

[5] R. Girshick. Fast R-CNN. arXiv:1504.08083, 2015.

[6] S. Ren, K. He, R. Girshick, and J. Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. arXiv preprint arXiv:1506.01497, 2015.

[7] J. Huang, V. Rathod, C. Sun, M. Zhu, A. Korattikara, A. Fathi, I. Fischer, Z. Wojna, Y. Song, S. Guadarrama, K. Murphy. Speed/accuracy trade-offs for modern convolutional object detectors. arXiv:1611.10012, 2016.

[8] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mane, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viegas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, X. Zheng. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. arXiv:1603.04467, 2016.

[9] J. Dai, Y. Li, K. He, and J. Sun. R-FCN: Object Detection via Region-based Fully Convolutional Networks. arXiv:1605.06409, 2016.

[10] H. Cheng, Z. Haque, L. Hong, M. Ispir, C. Mewald, I. Polosukhin, G. Roumpos, D. Sculley, J. Smith, D. Soergel, Y. Tang, P. Tucker, M. Wicke, C. Xia, J. Xie. TensorFlow Estimators: Managing Simplicity vs. Flexibility in High-Level Machine Learning Frameworks. arXiv:1708.02637, 2017.

[11] K. He, X. Zhang, S. Ren, J. Sun. Deep Residual Learning for Image Recognition. arXiv:1512.03385, 2015.

[12] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, H. Adam. MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications. arXiv:1704.04861, 2017.

[13] T. Lin, P. Dollár, R. Girshick, K. He, B. Hariharan, S. Belongie. Feature Pyramid Networks for Object Detection. arXiv:1612.03144, 2016.

[14] TensorFlow models: detection model zoo,
https://github.com/tensorflow/models/blob/
master/research/object_detection/g3doc/detection_model_zoo.md

[15] K. Simonyan, A. Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. arXiv:1409.1556, 2014.

[16] T. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. Lawrence Zitnick, P. Dollár. Microsoft COCO: Common Objects in Context. arXiv:1405.0312, 2014.

[17] T. Lin, P. Goyal, R. Girshick, K. He, P. Dollár. Focal Loss for Dense Object Detection. arXiv:1708.02002, 2017.

[18] A. Shrivastava, A. Gupta, R. Girshick. Training Region-based Object Detectors with Online Hard Example Mining. arXiv:1604.03540 , 2016.

[19] H. Law, J. Deng. CornerNet: Detecting Objects as Paired Keypoints. arXiv:1808.01244, 2018.

[20] A. Newell, K. Yang, J. Deng. Stacked Hourglass Networks for Human Pose Estimation. arXiv:1603.06937, 2016.

[21] S. Ioffe, C. Szegedy. Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. arXiv:1502.03167, 2015.

[22] S. Bell, C. Lawrence Zitnick, K. Bala, R. Girshick. Inside-Outside Net: Detecting Objects in Context with Skip Pooling and Recurrent Neural Networks. arXiv:1512.04143, 2015.

[23] H. Law, Y. Teng, O. Russakovsky, J. Deng. CornerNet-Lite: Efficient Keypoint Based Object Detection. arXiv:1904.08900, 2019.

[24] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, K. Keutzer. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and ¡0.5MB model size. arXiv:1602.07360, 2016.

[25] J. Redmon, A. Farhadi. YOLOv3: An Incremental Improvement. arXiv:1804.02767, 2018.

[26] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, T. Darrell. Caffe: Convolutional Architecture for Fast Feature Embedding. arXiv:1408.5093, 2014.

[27] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. In NIPS-W, 2017.

[28] R. Girshick, I. Radosavovic, G. Gkioxari, P. Dollár, K. He. Detectron. https://github.com/facebookresearch/detectron, 2018.

[29] S. Zhang, L. Wen, X. Bian, Z. Lei, S. Z. Li. Single-Shot Refinement Neural Network for Object Detection. arXiv:1711.06897, 2017.

[30] K. Duan, S. Bai, L. Xie, H. Qi, Q. Huang, Q. Tian. CenterNet: Keypoint Triplets for Object Detection. arXiv:1904.08189, 2019.

[31] COCO:Common Objects in Context. http://mscoco.org/dataset/#detections-leaderboard (2019) [Online; accessed 1-December-2019].