YOLOv4 and YOLOv4-tiny Object Detection

Safa Shaikh, Arjun Ahuja

April 24, 2021

1 INTRODUCTION

YOLO is a state of the art object detection model and object detection methods are constantly improving since the introduction of convolutional neural networks and deep learning. YOLOv4 has come out in 2020 and so has the compressed model YOLOv4-tiny. It is important to study the performance and tradeoffs between the two models because object detection is in high demand, especially for use on edge devices where a model needs to provide inferences in real-time. Object detection is used by major companies in many applications such as tailoring recommendations based on image searches or social media activity, and the realm of applications for this technology are vast.

To determine what YOLO model works best, we compare YOLOv4 vs YOLOv4-tiny, comparing architectures and training and test performance. We want to answer the question of whether the smaller model reduces accuracy and if there is a significant or negligible difference.

2 ARCHITECTURE

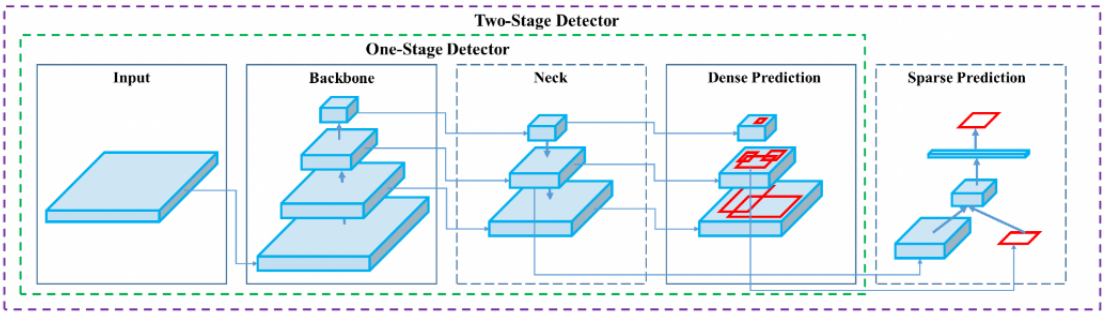
Yolov4 is built on the principles for achieving the best possible interleaved combination of Speed, Accuracy and Parallel Computation. Without sacrificing either, it manages to surpass the yolov3 and other real time object detectors by using optimal amalgamation of techniques from Bag of Freebies and Bag of Specials along with architecture design that results into an efficient real time object detector. Authors designed the overall architecture from a GPU training perspective which is unheard of, making it reliable to reproduce the results across conventional GPU skews.

2.1 OVERVIEW OF OBJECT DETECTION MODELS

A modern detector is usually composed of four parts:

1. Input: Image, Patches, Image Pyramid
2. Backbone: a backbone which is pre-trained on ImageNet (examples: VGG, ResNet, ResNeXt,or DenseNet)
3. Neck: Object detectors developed in recent years often insert some layers between backbone and head, and these layers are usually used to collect feature maps from different stages. We can call it the neck of an object detector.

4. Head: a head which is used to predict classes and bounding boxes of objects. Head part is usually categorized into two kinds, i.e., one-stage object detector and two-stage object detector. (Two stage detector example: fast R-CNN, faster R-CNN, R-FCN,and Libra R-CNN, one stage detector example: YOLO, SSD,and RetinaNet)



2.2 YOLOv4 Architechture

YOLOv4 consists of:

1. Backbone: CSPDarknet53
2. Neck: SPP(Spatial Pyramid Pooling), PAN(Path Aggregation Network)
3. Head: YOLOv3

2.3 Key Discriminators of YOLOv4

Bag of Freebies

Usually, a conventional object detector is trained off-line. Therefore, researchers always like to take this advantage and develop better training methods which can make the object detector receive better accuracy without increasing the inference cost. We call these methods that only change the training strategy or only increase the training cost as "bag of freebies." Yolov4 uses the following methods as bag of freebies:

1. Bag of Freebies (BoF) for backbone: CutMix and Mosaic data augmentation, DropBlock regularization,Class label smoothing
2. Bag of Freebies (BoF) for detector: CIoU-loss, CmBN, DropBlock regularization, Mosaic data augmentation, Self-Adversarial Training, Eliminate grid sensitivity, Using multiple anchors for a single ground truth, Cosine annealing scheduler, Optimal hyper-parameters, Random training shapes

A few key discriminators are Mosaic, CutMix, DropBlock regularization, and Self Adversarial Training. Mosaic tiles 4 images together which teaches the model to find smaller objects and pay less attention to the background. CutMix takes sections of some images and pastes them into other images. DropBlock regularization drops portions of the image where those sections hare hidden from the first layer. Self Adversarial Training is finding the portion of an image that network is most reliant on and then editing that portion of image so the network can generalize.

Bag of Specials

For those plugin modules and post-processing methods that only increase the inference cost by a small amount but can significantly improve the accuracy of object detection, we call them “bag of specials”. Yolov4 uses the following Bag of Specials to improve accuracy:

1. Bag of Specials (BoS) for backbone: Mish activa-tion, Cross-stage partial connections (CSP), Multi-input weighted residual connections (MiWRC)
2. Bag of Specials (BoS) for detector: Mish activation,SPP-block, SAM-block, PAN path-aggregation block,DIoU-NMS

An important one to remember is DIoU NMS, which chooses best object from overlapping bounding boxes by using Intersection over Union (IoU).

3 Differences between YOLOv4 and YOLOv4-tiny

The primary difference between YOLOv4 tiny and YOLOv4 is that the network size is dramatically reduced. The number of convolutional layers in the CSP backbone are compressed to feature maps of sizes 26x26 and 13x13 where they connect to the detector head. The number of YOLO layers are two instead of three and there are fewer anchor boxes for prediction. YOLOv4-tiny also uses Leaky ReLU instead of the Mish activation function.

YOLOv4-tiny has only 6.789 billion floating point operations compared to 59.570 billion FLOPs in YOLOv4.

4 Implementation

4.1 Datasets

We used Roboflow’s publicly available mask dataset[2], we observed the results of YOLOv4 and YOLOv4-tiny on the dataset without adding any extra augmentation(apart from Mosaic and CutMix which is a part of YOLOv4 framework) and after adding the following augmentations

1. Flip: Horizontal
2. Crop: 0% Minimum Zoom, 60% Maximum Zoom
3. Rotation: Between -5° and +5°
4. Shear: ±5° Horizontal, ±5° Vertical
5. Grayscale: Apply to 10% of images
6. Hue: Between -25° and +25°
7. Saturation: Between -25% and +25%
8. Brightness: Between -25% and +25%
9. Exposure: Between -10% and +10%
10. Blur: Up to 1px

The following table shows the dataset size and the train, validation and test set split for dataset with augmentation and dataset without augmentation:

	Training Set Size	Validation Set Size	Test Set Size
Without augmentation	105	29	15
With augmentation	315	29	15

4.2 Environment Setup

We trained YOLOv4 and YOLOv4-tiny in their native framework, which is Darknet. Google Colab was the primary interface to load data and run these commands. We configured our environment to work with GPUs by ensuring Nvidia CUDA drivers were installed and then we checked the architecture code for the type of GPU being used. This value is used in the makefile to build the darknet framework. We also compiled with LIBSO=1 to enbale some useful functions to be able to run object detection in real-time on a webcam stream and on saved video. After cloning and building Darknet, we were able to download roboflow datasets and start training.

4.2.1 GPUs

We trained both YOLOv4 and YOLOv4 tiny on three GPUs:

1. T4
2. V100
3. P100

4.2.2 Framework

We trained YOLOv4 and YOLOv4-tiny in their native framework, which is Darknet. Darknet is an open source neural network framework written in C and CUDA. Darknet does not have a user-friendly Python API most commands were run through command-line interface.

We used the makefile listed by YOLOv4 author on their github page[3] to configure Darknet for our notebook.

4.3 Steps to Train YOLOv4 and YOLOv4 tiny

A brief summary of the steps is as follows:

1. Configure our GPU environment on Google Colab
2. Install the Darknet YOLOv4 training environment
3. Download our custom dataset for YOLOv4 and set up directories
4. Configure a custom YOLOv4 training config file for Darknet
5. Train our custom YOLOv4 object detector
6. Reload YOLOv4 trained weights and make inference on test images

We trained YOLOv4 for 2000 iterations and YOLOv4-tiny for 4000 iterations since that is what was recommended for sufficient training. We trained both models on both datasets, which makes a total of 4 training runs.

5 Results and Observations

5.1 Runtime Metrics

	YOLOv4-tiny	YOLOv4
Tesla T4	2189 s = 36 min	12000 s = 200 min
T4 with extra augmentation	2199 s = 37 min	15696 s = 262 min
Tesla P100	1653 s = 28 min	8070 s = 135 min
P100 with extra augmentation	2210 s = 37 min	11000 s = 183 min
Tesla V100	1400 s = 23 min	4600 s = 77 min
V100 with extra augmentation	1950 s = 33 min	6900 s = 115 min

Figure 5.1: Runtime metrics over Tesla T4, P100, and V100 GPUs

5.1.1 Observations

:

1. With Augmentation V100 performs best among all GPUs, with training time of 23 minutes for YOLOV4 tiny and 77 minutes for YOLOv4. With Augmentation there is a similar story, V100 performs best among all GPUs, with training time of 33 minutes for YOLOV4 tiny and 115 minutes for YOLOv4. (Runtime order V100 < P100 < T4.)
2. YOLOV4-tiny is approximately 4-6 times faster than YOLOV4 even though there is almost a 10 times difference between number of FLOPs between them. The reason can be the number of iterations we are training it for, its 2000 for Tiny while its 4000 for YOLO.
3. Runtime does not necessarily scale linearly with amount of augmentation added, in the given scenario adding 3x augmentation just adds 1.7x extra time.

5.2 Prediction time

	V100(milliseconds)	P100(milliseconds)	T4(milliseconds)
YOLOv4-tiny	1.42	2.710000	5.388000
YOLOv4	11.803	20.922000	43.945000

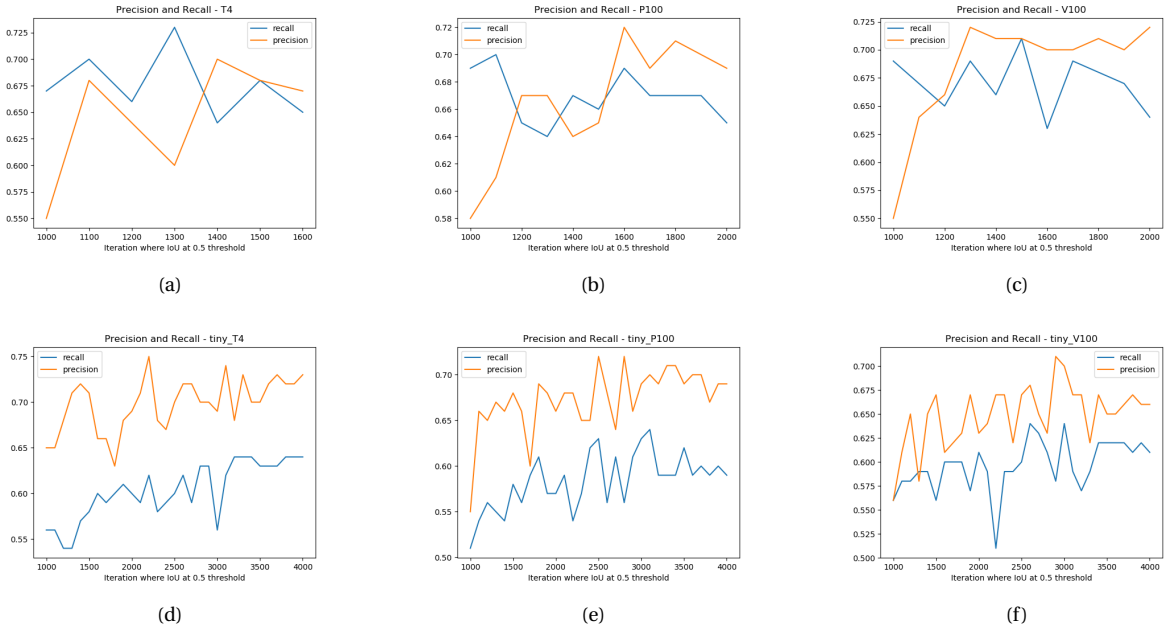
5.3 Images Processed per second

	V100	P100	T4
YOLOv4-tiny	704	370	172
YOLOv4	84	48	22

As expected V100 performs best among all GPUs, but the performance of T4 isn't bad either. V100 can process 704 images per second, while its 370 and 172 for P100 and T4 respectively. YOLOV4-tiny is considerable faster than YOLOv4 in prediction. We see a difference of 10 times in runtime which is same as expectation based on FLOPs.

5.4 Precision Recall Analysis

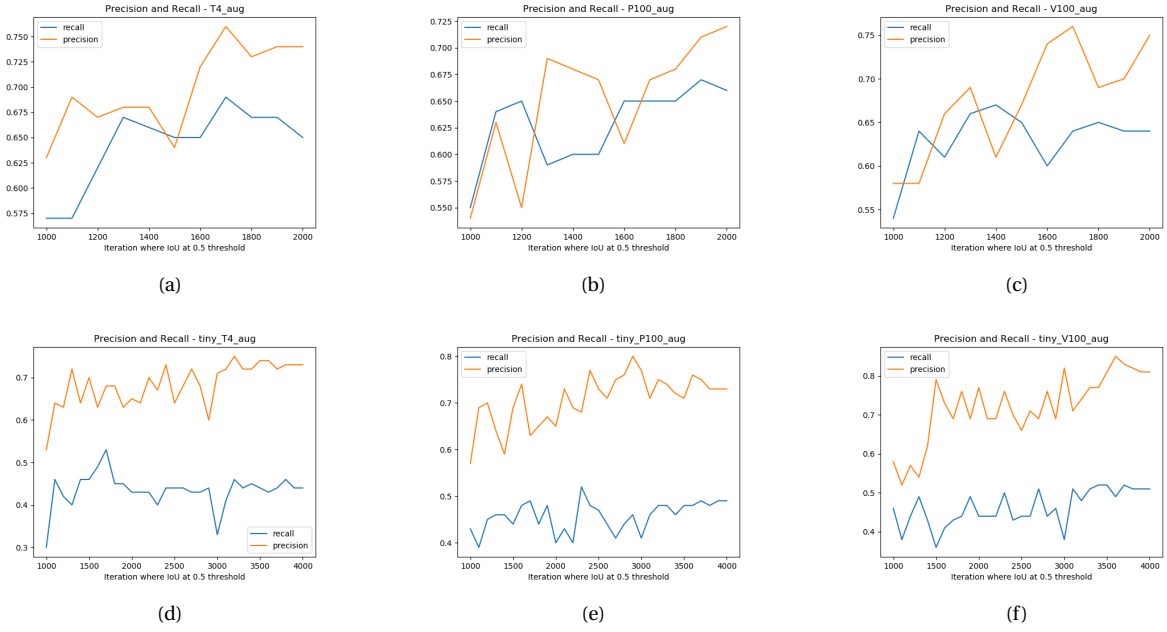
See the below precision recall curves for two cases, With Augmentation and Without Augmentation:



(a) (b) (c)

(d) (e) (f)

Figure 5.2: Precision and Recall values where IoU threshold is at least 0.5. Top row is YOLOv4 and bottom is YOLOv4-tiny



(a) (b) (c)

(d) (e) (f)

Figure 5.3: (Augmented Dataset) Precision and Recall values where IoU threshold is at least 0.5. Top row is YOLOv4 and bottom is YOLOv4-tiny

5.4.1 Observations

1. As can be seen from the curves max precision values without augmentation for YOLOV4 is slightly higher than YOLOV4-tiny (0.725 vs 0.7).

2. The max precision values with augmentation is higher than precision values without augmentation for both YOLOV4 and YOLOV4-tiny. (0.75 for YOLOv4 vs 0.725 earlier, and 0.8 vs 0.7 earlier.).

3. Interestingly for our use case tiny with Augmentation performs even better than normal YOLOv4 Model. (0.8 for tiny vs 0.75 for YOLOV4).

4. The problem with Tiny is that it has significantly lower recall values as compared to YOLOv4. This can be an important factor in real world systems.

5. Although inference testing was done on images, running both models on video and webcam produced slightly different results. We noticed more false positives in the YOLOv4 tiny model on the news video we ran it on.

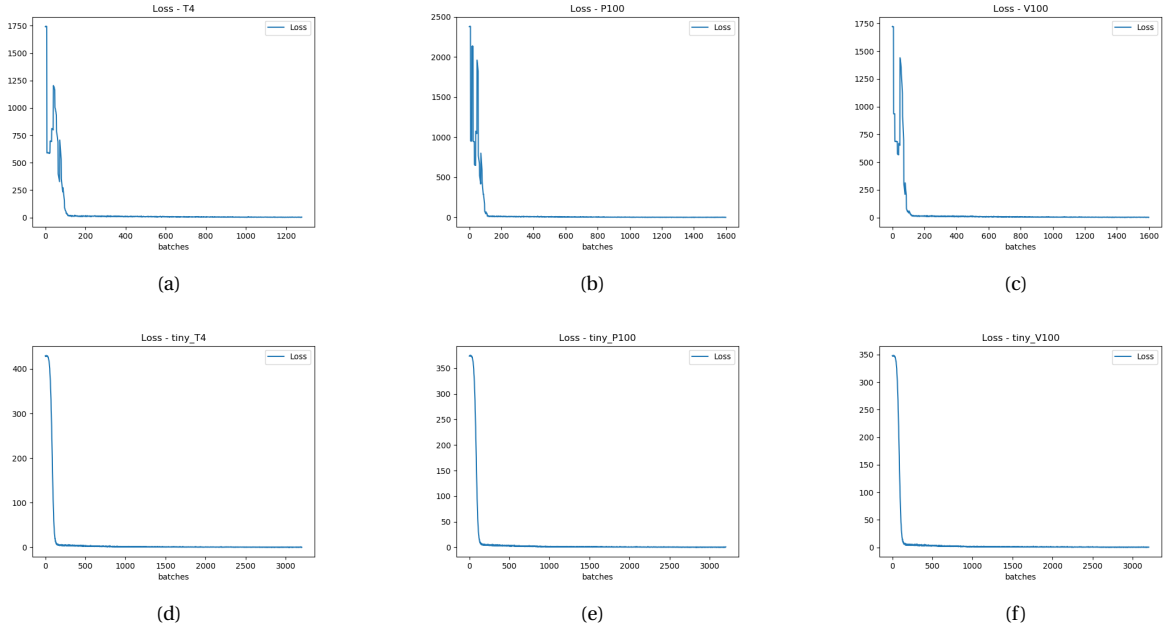


Figure 5.4: Loss Values. Top row is YOLOv4 and bottom is YOLOv4-tiny

5.5 Loss Analysis

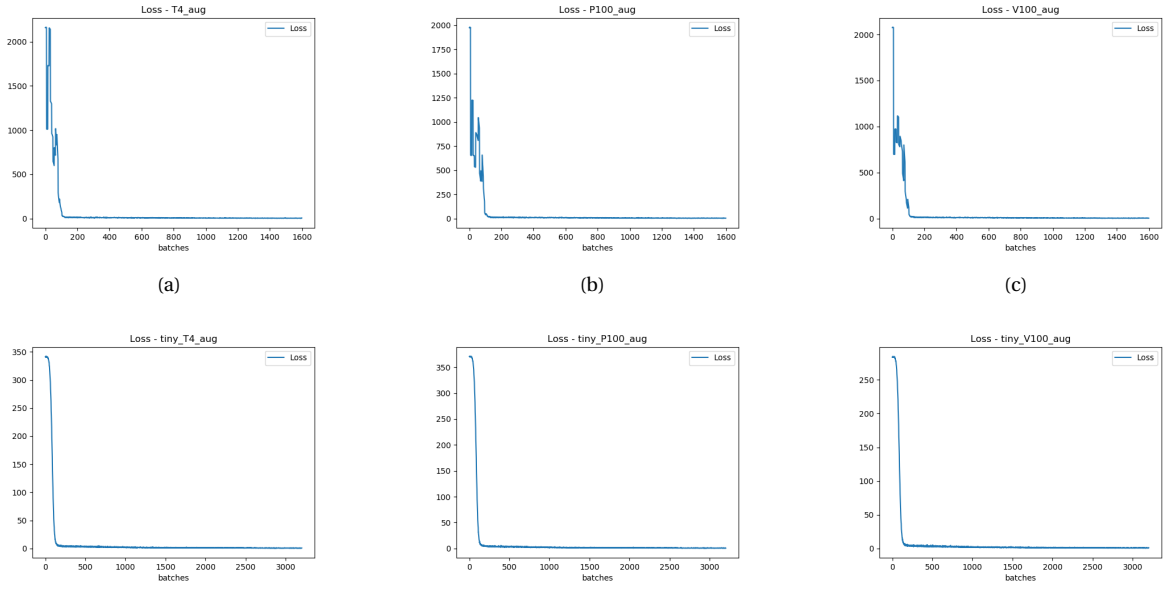
Please refer to figure 5.4 and 5.5 trend of loss values during training time.

1. The Loss graph for Tiny is very stable for use case of Masks (With loss almost always decreasing.) While the loss graph for YOLOv4 model shows some abrupt jumps.

2. Both Tiny and YOLOv4 reach around 0 training loss around the same number of batches processed(150).

3. Loss curves across GPUs follow nearly the same pattern as expected.

4. We do not see much difference between loss curves for models with and without augmentation.



(a) (b) (c)

(d) (e) (f)

Figure 5.5: (Augmented Dataset) Loss Values. Top row is YOLOv4 and bottom is YOLOv4-tiny

5.6 Photo/Video Results

Refer to figure 5.6 for results Both models were able to detect mask wearers accurately on all test images. However, when it came to real-time data, using webcam or video, YOLOv4-tiny produced more false positives than YOLOv4. You may refer to our Github repository [4] to view these videos. See Figure 5.7

6 Conclusion

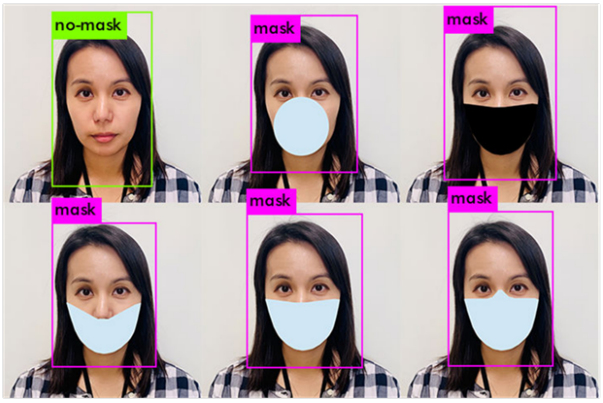
In conclusion we can see that YOLOv4 is a very robust model for object detection and it is able to detect objects with approximately 100% accuracy on stationary images and does a decent job on live video as well. If we were to provide a recommendation, we would pick YOLOv4 due to the lower false positive rate on video. If the use case was to only detect on stationary images however, YOLOv4-tiny is recommended due to its fast training and inference time. YOLOv4-tiny has the advantage of fast detection, so it may work better on edge devices as well. But if precision and accuracy is desired, sufficient hardware (a GPU) is available, and the inference time is not of significance YOLOv4 is recommended. On GPUs we did not see significant perceivable difference in detection(though numerically there is) even when running YOLOv4 and YOLOv4 tiny on video(Consider the frames per second, even on the slowest GPU, T4 we got 22 images per second on YOLOv4 and 172 images per second on YOLOv4 tiny). For a few experiments that we ran we found the false positive rate of YOLOv4-tiny to be higher than YOLOv4. This also supports our conclusion to use YOLOv4 whenever possible.

References

[1] Solawetz, Jacob; *Breaking Down YOLOv4*; https://blog.roboflow.com/a-thorough-breakdown-of-yolov4/

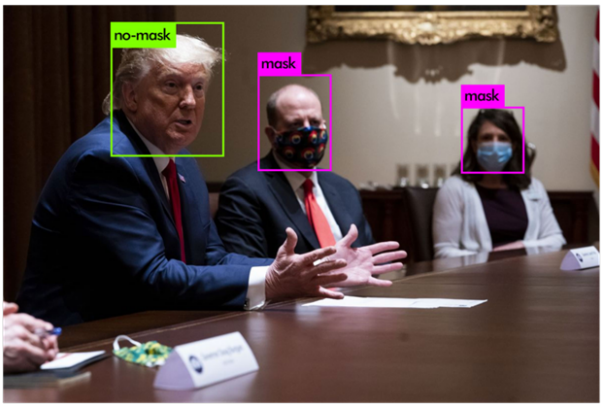
[2] Mask dataset `https://public.roboflow.com/object-detection/mask-wearing`

[3] Darknet Makefile `https://github.com/AlexeyAB/darknet/blob/master/Makefile`

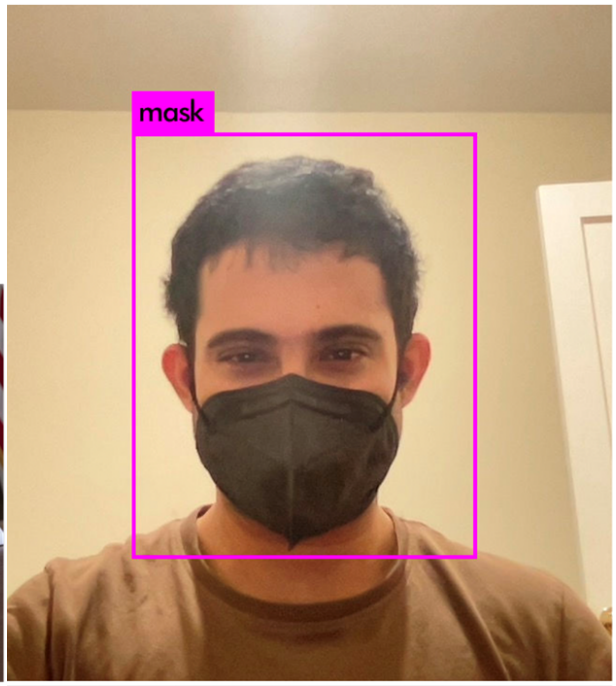


(a)

(b)

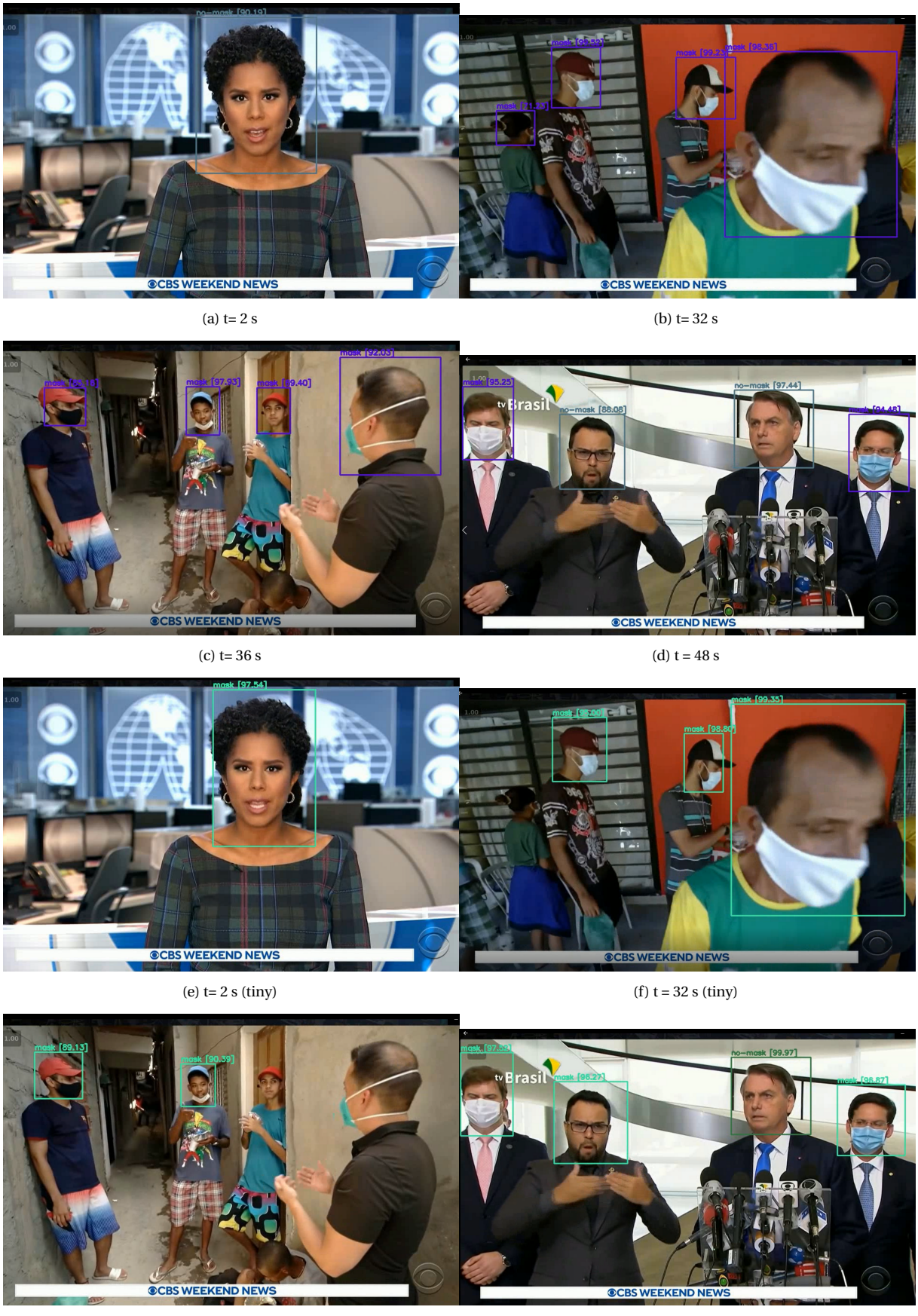


(c)



(d)

Figure 5.6: Sample Inference Images run on both YOLOv4 and YOLOv4-tiny



(a) t= 2 s

(b) t= 32 s

(c) t= 36 s

(d) t = 48 s

(e) t= 2 s (tiny)

(f) t = 32 s (tiny)

(g) t = 36 s (tiny)

(h) t = 48 s (tiny)

Figure 5.7: Video Inference Results. YOLOv4 results on top row and YOLOv4-tiny results on bottom row.

[4] https://github.com/safashaikh/YOLOv4-vs-YOLOv4-tiny

[5] Bochkovskiy, Alexey, et al. YOLOv4: Optimal Speed and Accuracy of Object Detection. 23 Apr. 2020, arxiv.org/pdf/2004.10934.pdf.

[6] JIANG, ZICONG, et al. Real-Time Object Detection Method for Embedded Devices. arxiv.org/pdf/2011.04244.