YOLO Net on iOS

Maneesh Apte
Stanford University
mapte05@stanford.edu

Simar Mangat
Stanford University
smangat@stanford.edu

Priyanka Sekhar
Stanford University
psekhar@stanford.edu

Abstract

Our project aims to investigate the trade-offs between speed and accuracy of implementing CNNs for real time object detection on mobile devices. We focused on modifying the YOLO architecture to achieve optimal speed and accuracy for mobile use cases. Specifically, we investigated the effect of a fire layer, as used in SqueezeNet, on classification performance. While this layer did increase the speed of classification, the decrease in accuracy was too great for use on mobile. We then deployed the highest performing tiny-YOLO net, which operates at 8-11 FPS with a mAP of 51%, to an iOS app using the Metal framework. The app can detect the 20 classes from the VOC dataset and translate the categories into 5 different languages in real time. This real-time detection is essentially a mobile Rosetta Stone that runs natively and is thus wifi independent.

1. Introduction

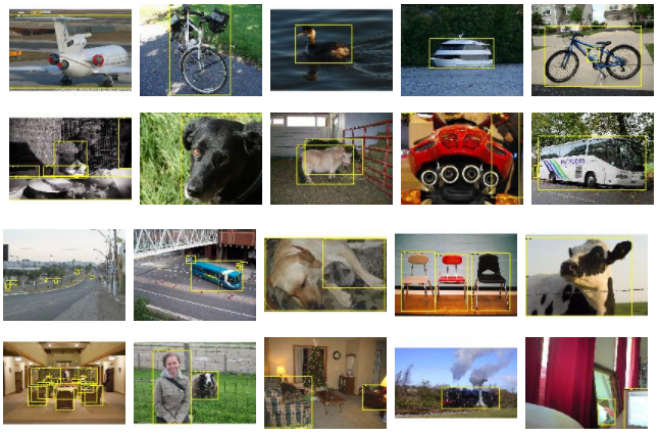
With the growing popularity of neural networks, object detection on images and subsequently videos has increasingly become a reality. However, fast object detection in real time with high accuracy still has large room for improvement. Our project is a combination of application and model modification. It aims to leverage existing work in the field of real time object detection and extend this work to 1) explore the trade offs between speed and accuracy for different models and 2) be deployed to iOS via an iPhone app that uses the phones video camera.

The Darkflow[25] implementation of the Darknet YOLO architecture [21] served as the base for our modifications. The input to our net is images from the VOC 2012 dataset, and the output will be an image with a bounding box, classification, and confidence along with a json object containing all the prediction information.

1.1. Dataset

Our primary dataset is from The PASCAL Visual Object Classes Project, abbreviated as VOC. The VOC challenge was a challenge that ran annually from 2007- 2012, giving participants standardized image datasets for object class recognition, evaluating performance of models via mean average precision (mAP) on their validation and test set. Training/validation data is available for download for each year on the competition website `http://host.robots.ox.ac.uk/pascal/VOC/index.html`.

The VOC 2012 data consists of 20 classes and the train/validation data has 11,530 images containing 27,450 ROI (region of interest) annotated objects and 6,929 segmentations.



We chose this dataset because of the simple, broad categories which prove useful in introductory language learning. In addition, the relatively small number of classes(20) enabled us to train a model that was feasible to run on mobile. For use in the current version of YOLO, images were resized to 416x416 resolution during training and test.

1.2. Methodology Overview

After surveying a number of other real-time object detection algorithms such as DPM, R-CNN, Fast R-CNN, and Faster R-CNN, we have decided to proceed with the YOLO (You Only Look Once) model [21]. YOLO provides fairly accurate object detection at extremely fast speeds.

We focused specifically on modifying and deploying tiny-YOLO [21], a version of YOLO that uses 9 convolu-

tion layers and 6 pooling layers instead of the 24 convolution layers in the traditional YOLO net.

We then used the Metal framework and the Forge library [10] to translate our architecture to Swift for deployment on our iPhones [12].

2. Background and Related Work

We surveyed several object detection architectures and mobile frameworks.

We first investigated Deformable Part Models, which are graphical models that can be used for predictions. According to Girshik et al. these models can be formulated as CNNs [7]. Current literature builds on this work to investigate how deformable convolution and deformable pooling can enhance the transformation modeling capability of the CNN [3]. However, because of our focus on mobile, we shifted our focus to other models with a greater emphasis on parameter reduction.

The first of these models is the RCNN, which proposes regions of interest and classifies regions with an SVM after convolution on each region [6]. The speed of this model is improved upon in Fast-RCNN [5] and Faster-RCNN [23], which speeds up passes by introducing Region Proposal Networks (RPN) to predict proposals from features.

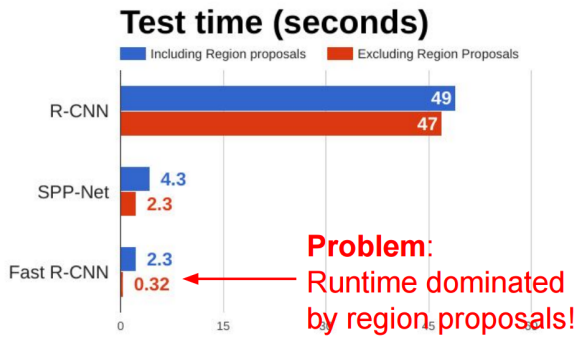


Figure 1. Comparison of RCNN speeds [1]

The RCNN is often used as a benchmark for the speed-accuracy trade off. For work focused exclusively on humans and human poses, Mask RCNN is a promising path. This architecture includes joint coordinates in addition to box coordinate [8].

We then looked into the "You Only Look Once" (YOLO) net [20], and decided this framework would be best suited for the limited resources available on mobile. The original net was improved upon in 2015, and this YOLO v2 serves as the primary focus of our experiments [21].

Our next focus was finding methods of parameter reduction to increase speed without sacrificing accuracy. SqueezeNet [15] provided a great overview of the effect of downsampling on AlexNet [17]. The promising work of SqueezeDet [26], which builds on the SqueezeNet principles and applies them to self-driving car applications, was particularly interesting and validated our thoughts that SqueezeNet had relevant ideas that could transcend AlexNet. While we did survey methods of compression such as one-shot whole network compression with Bayesian matrix factorization, Tucker decomposition on kernel tensor, and fine-tuning to recover accumulated loss of accuracy as presented by Kim et al., [16], we decided the best path forward forward given our prior familiarity with SqueezeNet was to focus on using its principals for parameter reduction in a different model.

We surveyed several architectures that might benefit from downsampling. These included high parameter architectures such as ResNet [9], GoogLeNet [24]. GoogLeNet also provided us with a foundational understanding of how 1x1 filters and concatenation could effectively reduce parameters without losing expressivity. However, we decided that even if heavily optimized, they would still run too slowly on modern mobile devices and continued to focus on faster models with better real time detection potential. We also looked at the current trade offs of various models as presented by Huang et al. [14], which showed that combinations of models are often optimal.

While we decided to use the VOC dataset, we also considered the COCO dataset [18], which has far more categories and may be the focus of future studies. For this study, VOC provided the focus we needed.

Finally, MobileNet from Google [13] has demonstrated that depth-wise separable convolutions can remove the bottleneck of convolution layers with many channels. These "thinner" models and the methodology behind them inform our future work.

We chose to focus on YOLO because it provided a decent baseline for accuracy/speed and was most accessible for modification.

3. Technical Approach

3.1. YOLO

The YOLO model's novel motivation is that it re-frames object detection as a single regression problem, directly from image pixels to bounding box coordinates and class probabilities. This means that the YOLO model only "looks once" at an image for object detection.

It works by dividing an image into S x S grid. Each grid cell predicts B bounding boxes and a confidence score for each box. The confidence scores show how confident the model is that there is an object in that bounding box. This is formally defined as:

$$Confidence = Pr(Object) * IOU_{pred}^{truth}$$

IOU_{pred}^{truth} is defined as the intersection over the union between the predicted box and the ground truth. For reference, our model used $S = 13$ and $B = 5$, as done in the Yolov2 original paper.

Following this, each bounding box has 5 predictions: dx, dy, w, h, and confidence. The (dx, dy) coordinates represent the center of the box relative to the bounds of the grid cell. Each grid cell also predicts C conditional class probabilities, which in mathematical notation is given by $Pr(Class_i|Object)$.

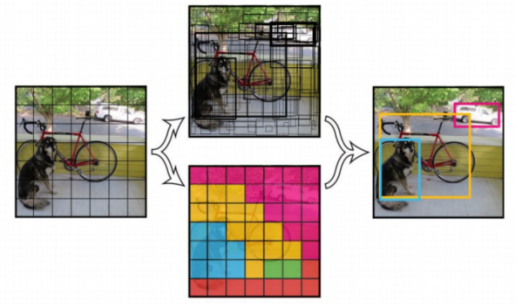


Figure 2. Yolo model regression pipeline (Redmon et. al.)

The formulation of object detection described above allows us to put a neural network architecture on top of this representation and get predictions for objects in image frames. However, certain architectures regarding the final prediction layer greatly influence the number of parameters and therefore overall speed of the implementation. A novel idea introduced by YoloV2 was to completely eliminate the need for a fully connected layer at the end of the network for predictions and instead use a convolution layer that makes predictions. In order to make this possible, the Yolo implementation uses anchor boxes to predict bounding boxes, meaning it predicts offsets instead of direct coordinates and thus accessible for a convolutional layer to make reasonable predictions. The locations for the anchor boxes are given by the configuration files that were downloaded from the Yolo website `https://pjreddie.com/darknet/yolo/`. For our project we focused on models specific for the VOC task, meaning the number of classes for prediction was 20.

Since there are a various architectures that can be implemented for this task, the authors of YOLO described a full YOLO version which achieved optimal accuracy and a more compact YOLO called tiny-yolo that run faster but isn't as accurate. Tiny-yolo was important to our project because it allowed us to get reasonable results when deployed to the limited hardward of a mobile device.

The full architecture yolo-tiny is below (max-pool-2 means max pool of size 2x2 and stride 2, NxN convolution-F-P means convolution of F filters of size NxN with pad P followed by batchnorm):

```
Input 416x416x3
3x3 convolution-16-1
max-pool-2
3x3 convolution-32-1
max-pool-2
3x3 convolution-64-1
max-pool-2
3x3 convolution-128-1
max-pool-2
3x3 convolution-512-1
max-pool-2
3x3 convolution-1024-1
3x3 convolution-1024-1
1x1 convolution-125-1
```

3.2. Fire Layer

A goal of this project was to speed up the implementation of YOLO so that it could classify faster on the limited hardware of a mobile device. As a result, our group researched various architectures that have been used in recent research that minimize the number of parameters in a neural network without compromising on accuracy. Thus, we found the architecture SqueezeNet [15], and realized that we could borrow motifs from SqueezeNet and use them in the YOLO architecture to replace bottleneck layers while hopefully maintaining accuracy. In particular, we looked at SqueezeNet's "fire module," which is a composition of two steps of convolutions with a concatenation at the end. The first layer is called the "squeeze" layer, which is a 1x1 convolution with s_{1x1} filters. The second layer has two separate convolutions, one is 1x1 with e_{1x1} filters and the second is 3x3 with e_{3x3} filters. A diagram of this is shown in figure 3.

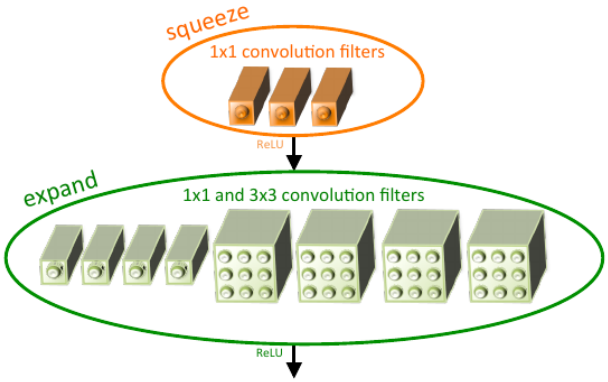


Figure 3. Fire module diagram (Iandola et. al.)

We focused on reducing the apparent bottleneck in the last two layers of the tiny-yolo architecture, where the 3x3

filters with padding of 1 convolutions had 1024 filters, replacing those layers with fire modules. See experiments section for more details.

3.3. Mobile

Since iOS does not support CUDA or OpenCL, we use Metal and Metal Performance Shaders to perform work on the GPUs found in iOS devices. Thus to bring our implementation onto mobile, we had to convert weights into ones that could be interpreted by Metal. We used the open-source "YAD2K" script, which converts the Darknet weights to a Keras format [2], to the compatible format. For new models we could almost directly translate architectures into Swift using Metals convolution and maxpool abstraction and incorporate the weights trained by Darkflow. However, one major drawback of this approach is that that Metal does not currently support an abstracted batchnorm layer, a regularization technique used in the tiny-YOLO implementation.

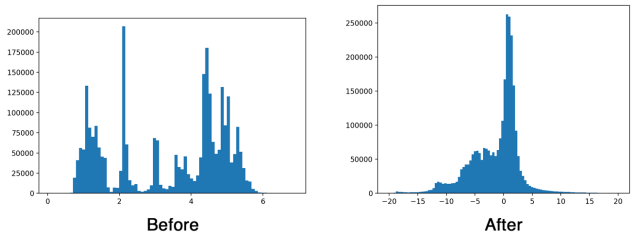


Figure 4. Histogram of output distributions before and after batchnorming for the first convolutional layer. As noted in the Hollemans post, without batchnorming, output distributions tend to become diffuse (left) and accuracy can decrease. Image from Hollemans. [12]

We were able to overcome this hurdle by using a strategy posted by M. Hollemans [12], who addressed the discrepancy by using a folding technique to essentially merge the batch norm weights into the previous convolutional layer [11]. We can use the following equations to compute new weights and bias terms for the convolution layer.

$$w_new = \frac{gamma * w}{sqrt(variance)}$$

$$b_new = \frac{gamma * (b - mean)}{sqrt(variance)} + beta$$

Hollemans' post also offered starter Yolo code that leverages the Forge wrapper library around Metal and was integral to our project [10]. Forge is a collection of helper code that makes it easier to construct deep neural networks using Apple's MPSCNN framework. For example, to set up tiny-yolo using Forge, you can implement the layers in Swift code like below.

```
let leaky = MPSCNNNeuronReLU(device: device, a: 0.1)

let input = Input()

let output = input
        --> Resize(width: 416, height: 416)
        --> Convolution(kernel: (3, 3), channels: 16, activation: leaky, name: "conv1")
        --> MaxPooling(kernel: (2, 2), stride: (2, 2))
        --> Convolution(kernel: (3, 3), channels: 32, activation: leaky, name: "conv2")
        --> MaxPooling(kernel: (2, 2), stride: (2, 2))
        --> Convolution(kernel: (3, 3), channels: 64, activation: leaky, name: "conv3")
        --> MaxPooling(kernel: (2, 2), stride: (2, 2))
        --> Convolution(kernel: (3, 3), channels: 128, activation: leaky, name: "conv4")
        --> MaxPooling(kernel: (2, 2), stride: (2, 2))
        --> Convolution(kernel: (3, 3), channels: 256, activation: leaky, name: "conv5")
        --> MaxPooling(kernel: (2, 2), stride: (2, 2))
        --> Convolution(kernel: (3, 3), channels: 512, activation: leaky, name: "conv6")
        --> MaxPooling(kernel: (2, 2), stride: (1, 1), padding: .same)
        --> Convolution(kernel: (3, 3), channels: 1024, activation: leaky, name: "conv7")
        --> Convolution(kernel: (3, 3), channels: 1024, activation: leaky, name: "conv8")
        --> Convolution(kernel: (1, 1), channels: 125, activation: nil, name: "conv9")
```

After appropriate channel and bounding box processing combined with UI functionality, we could spin up an app that drew bounding boxes with confidence scores in real time. To add additional functionality, we also added language buttons that allow a user to identify objects in 5 different languages (English, Spanish, French, Hindhi, and Chinese).

4. Experiments

Our experiments aimed to determine which net, YOLO, tiny-YOLO, or tiny-YOLO with fire layer modification would be best for deployment on mobile. Our modified tiny-YOLO replaced 3 convolutional layers - the final 1024 and 512 convolutional layers - with two fire layers.

```
...

[convolutional]
batch_normalize=1
filters=256
size=3
stride=1
pad=1
activation=leaky

[maxpool]
size=2
stride=2

[fire]
s_1x1=38
e_1x1=256
e_3x3=256

[fire]
s_1x1=42
e_1x1=512
e_3x3=512

[convolutional]
size=1
stride=1
pad=1
```

```
filters=425
activation=linear
...
```

In addition, we used the YOLO and the tiny-YOLO net architectures to train and test on the VOC 2012 validation dataset and evaluated their speed and accuracy locally using mAP.

We chose to evaluate the models on CPU to mimic the limited computing resources available on modern phones. We then took the model with the highest CPU speed and implemented it in a mobile app.

4.1. Baselines

The VOC dataset is considered "solved" in many contexts. The VOC 2012 competition saw mAP scores of over 90% in each category [4]. The full YOLO version has a reported mAP score of 78% on the VOC 2007 test set after being trained on the VOC 2007 + 2012 sets [21]. The full YOLO net served as the baseline for our experiments. The GPU runs were conducted with a Titan X and served as our benchmark.

We used the Adam optimizer in all training sessions due to its tendency for expedient convergence, and we set a learning rate of $1e^{-3}$ to begin training as suggested in the Darkflow implementation [25]. If the model's loss started thrashing (oscillating between two large numbers) we paused training and resumed from the last checkpoint with a lower learning rate. The lowest rate we used was $1e^{-6}$. We used the default mini-batch size of 64 for training, and we ran CPU evaluations using pre-trained weights from Darknet [21].

4.2. Evaluation

We evaluted the accuracy of networks using mAP scores on the VOC test set. We heavily modified an R-CNN mAP evaluation script [19] to implement our own mAP score evaluator that is compatible with the json output of Darkflow.

To get the mean average precision, we compute the average precision for each of the 20 classes as follows:

$$AvgPrecision = \sum_{k=1}^{n} P(k)\Delta r(k)$$

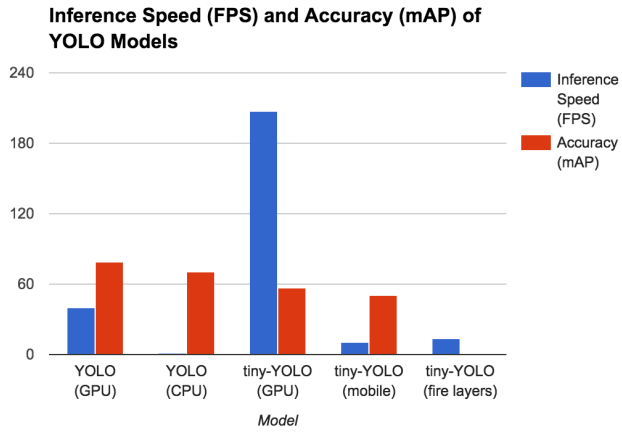
In this equation, $P(k)$ refers to precision at a given threshold k and $\Delta r(k)$ refers to the change in recall between k and $k-1$. This corresponds to the area under the precision recall curve. Thus mean average precision is simply: $AvgPrecision/C$ Where C = 20 for the VOC PASCAL dataset.

To evaluate test time inference speed on CPU, we took 100 images from the VOC validation set and used the Darknet framework to time how long each architecture took to make predictions for all the images. We then divided the total time by the number of images (100) to get an average number of images (frames) per second. To get scores for Mobile, we used the Forge Library to determine the max FPS for our deployed net. The GPU speed references those reported on the Darknet Website [21].

The GPU times were recorded using the NVIDIA Titan X as reported in Darkent [21]. The CPU times were recorded on a 2016 MacBook Pro with 2.7 GHz Intel Core i5. The mobile times were recorded on a standard iPhone 7 running iOS 10.

4.3. Results

We report our findings on speed and accuracy on mobile and GPU in comparison to modern models below.



Graph 1: Empirical Speed and mAP of Locally Trained and Tested Models Using the VOC 2012 Dataset

We include a detailed comparison in Table 1. The tiny-YOLO net significantly outperformed all others in terms of speed, overtaking YOLO by a factor of 5x on both CPU and GPU. We achieved slightly lower accuracies when the nets were tested locally on a sample from the 2012 VOC validation set (rather than on the 2007 sets used to set the benchmarks on the Darknet website). The tiny-YOLO fire layer modification was slightly faster but failed to predict any examples.

The locally tested YOLOs struggled with similar categories as their GPU counterparts, as expected. tiny-YOLO also trained and predicted significantly faster than YOLO because it has a fraction of the layers and parameters. tiny-YOLO-fire had even fewer parameters overall, since although fire layers perform 2 convolutions, we replaced 3 convolution layers with 2 fire layers, reducing overall parameter count. and thus performed slightly faster than tiny-YOLO, as expected.

Model	Speed (FPS)	FLOPS	mAP
SqueezeNet (GPU)	-	2.17 Bn	-
YOLO (GPU)	40	59.68 Bn	**78.6**
tiny-YOLO (GPU)	**207**	0.81 Bn	57.1
YOLO (CPU)	2	59.68 Bn	65
tiny-YOLO (CPU)	6	0.81 Bn	51
tiny-YOLO-fire (CPU)	8	-	< 1
tiny-YOLO (mobile)	11	0.81 Bn	-

Table 1. Best Reported Inference Speed and mAP scores of models compared to our tests of YOLO, tiny-YOLO and tiny-YOLO-fire on CPU

4.4. Discussion of Fire Layer

We attempted to train the modified tiny-YOLO with fire layers from scratch. This required instantiating a blank file and training took around 4 hours on the GPU provisioned by Google Cloud which resulted in a minimum loss of 6.

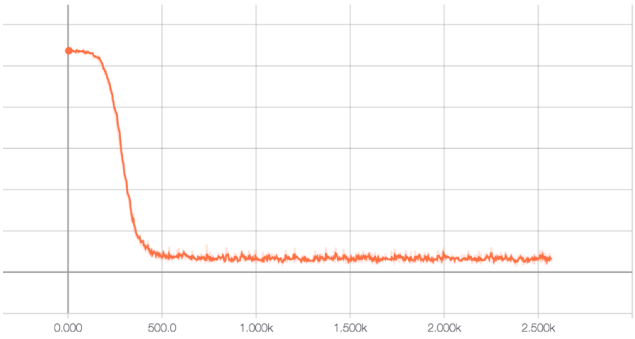


Figure 5. Tensorboard loss convergence graph of tiny-YOLO-fire after 500 iterations

However, although the loss converged, the net still had a mAP of 0 at test time and was unable to make predictions for almost all images. As a result, we attempted to modify the s1x1 layer of the second fire layer to have 46 channels instead of 42 (to better align with the original SqueezeNet implementation[15]), and tuned hyperparameters such as the learning rate. This also had no effect.

We then used the pre-trained weights for the first convolutional layers from tiny-YOLO and fine-tuned the remaining fire layers using these weights as a starting point. To test this technique, we created a small set of 3 images and annotations from the VOC validation set and trained the fire-layer tiny-YOLO on these images in an attempt to overfit and accurately predict these images at test time. The net was unable to make accurate classifications on this dataset as well.

These experiments led us to believe that fire layers are ill suited as replacements to vanilla convolution layers. Intuitively, this makes sense, as the fire layers concatenate outputs leading to very different maps from sequential convolution.

Furthermore, the fire layers added at the end of the architecture squeeze an already aggressively reduced neural net. However, before these layers, the activations are already somewhat well-defined and the parameter count is very low. As a result, additional squeezing without careful precision may result in the loss or scrambling of useful information, making the model far less expressive and unable to make sense of information learned in previous layers. Additionally, while fire layers may have made sense for all-image input without bounding boxes, where concatenated channels revealed information about the interplay of different features in an image, adding text and bounding boxes likely only adds noise to the outputs since there is a disconnect between the meaning of the bounding box parameters and the pixel values.

Further tuning of the s1x1, e1x1, and e3x3 filter sizes and layers would need to take place to ensure that the network captures relevant data in these steps.

Another factor to consider is that while YOLO relies on Euclidean loss, the original SqueezeNet uses Softmax [15].

$$L_{Eucl} = \frac{1}{2}\sum_{i=1}^{n}(x_i - y_i)^2$$

$$L_{Softmax} = -log(\frac{e^{f_{y_i}}}{\sum_j e^{f_j}})$$

These are two fundamentally different ways of measuring loss, with one relying on distance while the other relies on probability distributions.

Empirical evidence suggests that YOLO is not as effective when evaluated using Softmax, with mAP scores dropping as low as 5-10% upon changing to the Softmax loss function [22]. It is possible that the reverse is true for SqueezeNet-inspired architectures and Euclidean Loss, with the incompatible loss function having a strong negative effect on final mAP scores.

4.5. Results of Deployment

The tiny-YOLO net was successfully deployed to the iPhone 7 using Forge [12]. The app only draws bounding boxes if the confidence of prediction is above a user-defined threshold (0.5 in our case). The deployment to mobile actually saw a slightly higher maximum Inference Speed than on laptop CPU because of the direct access to the mobile GPU provided by Metal2. However, the average speed roughly matched the results of the same network on CPU.

The relatively low mAP score is evidenced by the tendency of our app to ignore chairs and struggle to predict bottles. The categories with the lowest average precisions, such as potted plants and sofas, are rarely detected. This is likely because these categories are so

variant in both shape and color and often include a lot of background noise. The app has the highest mAP for the categories of people and cars. To view a demo, visit: `https://www.youtube.com/watch?v=m0_GXm9aN5Q&feature=youtu.be` Images from a run are shown below.

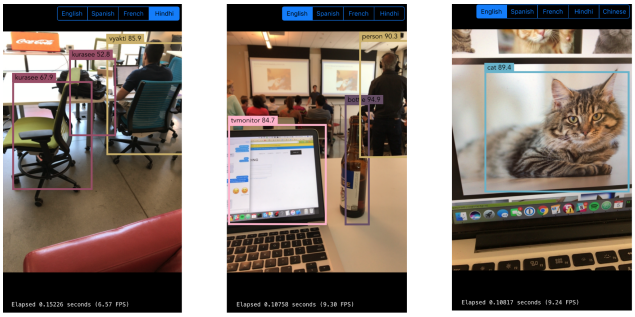


Figure 6. Detections in different languages

5. Conclusions and Future Work

While the fire layer may be promising for speeding up the bottlenecked high-channel layers, the lack of predictions in this project make it hard to draw positive conclusions for its future usefulness. Further tuning on a large number of parameters would need to take place before conclusions can be drawn with this hybrid architecture.

More likely, it seems that layer-pruning and depth-wise separable convolutions, as proposed in Google's MobileNet [13], will provide an effective trade-off between speed and efficiency. These models shrink the number of hyperparameters without making drastic changes to architecture and seem to be the next step in making deep learning practical on mobile. Next steps would investigate MobileNet and potential hybrids between it and YOLO.

Mobile deep learning is quickly becoming more accessible. With a push for increased GPU access on phones, even the full YOLO may one day run in real time on our iPhones. Consequently, the best development strategy to take advantage of mobile deep learning might be to slowly add to existing models such tiny-YOLO to make them approach YOLO accuracy. Nevertheless our current app indicates that object detection for mobile can be done in near real time and hints at many possibilities in the realms of facial recognition, AR, and autonomous driving in the near future.

References

[1] Cs231n lecture 11 slide 80. May 10 2017.

[2] allanzelener. Yad2k. *https://github.com/allanzelener/YAD2K/*, 2017.

[3] J. Dai, H. Qi, Y. Xiong, Y. Li, G. Zhang, H. Hu, and Y. Wei. Deformable convolutional networks. *CoRR*, abs/1703.06211, 2017.

[4] M. Everingham, L. Van Gool, C. K. I. Williams, J. Winn, and A. Zisserman. Assessing the Significance of Performance Differences on the PASCAL VOC Challenges via Bootstrapping. 2012.

[5] R. B. Girshick. Fast R-CNN. *CoRR*, abs/1504.08083, 2015.

[6] R. B. Girshick, J. Donahue, T. Darrell, and J. Malik. Rich feature hierarchies for accurate object detection and semantic segmentation. *CoRR*, abs/1311.2524, 2013.

[7] R. B. Girshick, F. N. Iandola, T. Darrell, and J. Malik. Deformable part models are convolutional neural networks. *CoRR*, abs/1409.5403, 2014.

[8] K. He, G. Gkioxari, P. Dollár, and R. B. Girshick. Mask R-CNN. *CoRR*, abs/1703.06870, 2017.

[9] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. *CoRR*, abs/1512.03385, 2015.

[10] hollance. Forge: A Neural Network Toolkit for Metal. *https://github.com/hollance/Forge*, 2017.

[11] hollance. Yolo2metal open source script, 2017. https://github.com/hollance/Forge/blob/master/Examples/YOLO/yolo2metal.py.

[12] M. Hollemans. Real-time object detection with yolo. *http://machinethink.net/blog/object-detection-with-yolo/*, 2017.

[13] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

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

[15] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *CoRR*, abs/1602.07360, 2016.

[16] Y. Kim, E. Park, S. Yoo, T. Choi, L. Yang, and D. Shin. Compression of deep convolutional neural networks for fast and low power mobile applications. *CoRR*, abs/1511.06530, 2015.

[17] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in Neural Information Processing Systems*, page 2012.

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

[19] rbgirshick. Fast/er r-cnn: https://github.com/rbgirshick/py-faster-rcnn/blob/master/lib/datasets. 2016.

[20] J. Redmon and A. Farhadi. You only look once: Unified, real-time object detection. *https://pjreddie.com/darknet/yolo/*, 2015.

[21] J. Redmon and A. Farhadi. Yolo9000: Better, faster, stronger. *https://pjreddie.com/darknet/yolo/*, 2016.

[22] J. Redmon and A. Farhadi. Yolo9000: Better, faster, stronger. *https://pjreddie.com/publications/yolo/*, 2016.

[23] S. Ren, K. He, R. B. Girshick, and J. Sun. Faster R-CNN: towards real-time object detection with region proposal networks. *CoRR*, abs/1506.01497, 2015.

[24] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. E. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. *CoRR*, abs/1409.4842, 2014.

[25] thtrieu. Darkflow. *https://github.com/thtrieu/darkflow*, 2017.

[26] B. Wu, F. N. Iandola, P. H. Jin, and K. Keutzer. Squeezedet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving. *CoRR*, abs/1612.01051, 2016.