YoloFlow
Real-time Object Tracking in Video
CS 229 Course Project

Konstantine Buhler John Lambert Matthew Vilim

Departments of Computer Science and Electrical Engineering
Stanford University
{buhler,johnwl,mvilim}@stanford.edu

Abstract

We reimplement YOLO, a fast, accurate object detector, in TensorFlow. To perform inference, we leverage weights that were trained for over one week on GPUs using ImageNet data, a publicly-available dataset containing several million natural images. We demonstrate the ability to reproduce detections comparable with the original implementation. We learn the parameters of the network and compare mean average precision computed from pre-trained network parameters. Furthermore, we propose a post-processing scheme to perform real-time object tracking in live video feeds.

1. Introduction

Performing real-time object detection and classification within images is an active area of research with applications ranging from autonomous warfare to transportation, sports, and consumer photography. Perception in autonomous vehicles relies upon fast, accurate object detection capabilities.

2. Related Work

Redmon *et al.*'s [13] YOLO (*You Only Look Once*) real-time object detection network achieves high performance on the PASCAL Visual Object Classes (VOC) 2007 Object Detection Challenge. Their work is inspired by the GoogLeNet and Network in Network [8] models for image classification [15].

Redmon *et al.*'s work is especially notable for two major strengths. First, their model solves in an end-to-end fashion what was considered in the not-far-distant past two separate problems in computer vision literature: object detection and object classification. Second, their model presents an efficient solution to an enduring problem in computer vision: *how does one go about producing an arbitrary number of detections in an image while using fixed dimensional input, output, and labels?* YOLO avoids computationally expensive region proposal steps that detectors like Fast R-CNN[4] and Faster-RCNN[14] require. However, since the time of YOLO's publication, newer models such as Single-Shot Multi-Box Detectors [9] seem to offer improvement in mAP with reduced GPU inference time [6]. YOLO uses grid cells as *anchors* to detections, much like Faster R-CNN and Multi-Box.

3. Dataset and Features

We use the PASCAL VOC 2007, a set of RGB images labeled with bounding box coordinates and class categories. The CNN learns high-quality, hierarchical features automatically, eliminating the need for hand-selected features.

4. Methods

4.1. TensorFlow Implementation

YOLO is implemented as a 32 layer deep convolutional neural network (DNN). The open source implementation released along with the paper is built upon a custom DNN framework written by YOLO's authors, called *darknet* [1]. This application provides the baseline by which we compare our implementation of YOLO [2]. Redmon *et al.* have released several variants of YOLO. For our purposes, we chose the variant outlined in [13]. In places in which the paper lacks details, we refer to the baseline *darknet* implementation to resolve ambiguities.

[1] github.com/pjreddie/darknet
[2] https://github.com/johnwlambert/YoloTensorFlow229

4.2. CNN Architecture

YOLO reframes object detection as a single regression problem, straight from image pixels to bounding box coordinates and class probabilities. YOLO divides the input image into an $S \times S$ grid. If the center of an object falls into a grid cell, that grid cell is responsible for detecting that object. A description of the CNN architecture can be seen in Figure 1.

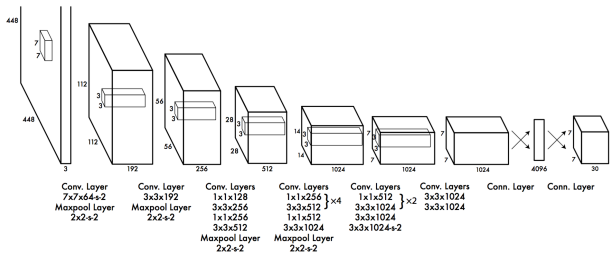

Figure 1. The convolutional neural network architecture.

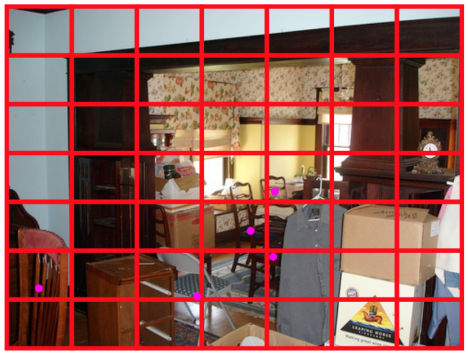

Figure 2. Observe the object centers (see pink dots) within each grid cell. These respective grid cells will be "responsible" for these chair objects.

4.3. The Loss Function

YOLO's loss function must simultaneously solve the object detection and object classification tasks. This function simultaneously penalizes incorrect object detections as well as considers what the best possible classification would be. We employ the stochastic gradient descent optimization method offered by TensorFlow[10] with the Adam optimizer [7] to minimize the cost function. We implement the following loss function, composed of five terms:

$$\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{obj} (x_i - \hat{x}_i)^2 + (y_i - \hat{y}_i)^2$$

$$+\lambda_{coord} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{obj} (\sqrt{w_i} - \sqrt{\hat{w}_i})^2 + (\sqrt{h_i} - \sqrt{\hat{h}_i})^2$$

$$+ \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{obj} (C_i - \hat{C}_i)^2$$

$$+\lambda_{noobj} \sum_{i=0}^{S^2} \sum_{j=0}^{B} 1_{ij}^{noobj} (C_i - \hat{C}_i)^2$$

$$+ \sum_{i=0}^{S^2} 1_{i}^{obj} \sum_{c \in classes} (p_i(c) - \hat{p}_i(c))^2$$

Note that 1_i^{obj} equals one if "object appears in cell i." 1_{ij}^{obj} equals one if the "jth bounding box predictor in cell i is "responsible" for that detection."

For each object in the ground truth, only a single bounding box can be responsible for that object. Suppose $S = 7$, $B = 2$ and there are two objects in a given image. Then, of the $7 * 7 * 2 = 98$ predictions of bounding boxes in the image, only 2 will be "responsible" for an object since there are only two objects. We calculate the bounding box predictor j that is responsible for an object by choosing the bounding box with the largest intersection over union (IoU) from the grid cell i where the object center is known to lie. Thus, in our given example, for each of the two objects in the image, we would loop through the $j = 0, .., B$ predictions in the known grid cell i (where the object center lies) and find j with the highest IoU value.

4.3.1 Ambiguities in Loss Function Definition

The authors' description of the setup of the cost function is extremely concise, leading to two main ambiguities. First, how does one assign class probabilities when two boxes of different class probabilities are found in one grid cell? Second, the authors define 1_{obj}^{ij} as *jth bounding box predictor in cell i is responsible for that prediction*, but it is unclear if i is learned dynamically or if it is part of the ground truth. We call the former interpretation *Interpretation 1* and the latter *Interpretation 2*. We evaluated loss function convergence under both interpretations.

5. Experimental Results

5.1. Weight Conversion

To simplify the port of YOLO from *darknet* to TensorFlow, we begin by using a pre-trained model. This greatly simplifies debugging as we can obtain immediate results from our execution without the need to implement a loss

function and back propogation to adjust the weights. Furthermore, training a new model is naturally slow. For example, our pre-trained model was trained by [13] for a week on the latest high-performance GPUs.

We select one of YOLO's pre-trained models, *yolo_small.weights* provided on the author's website [3]. These weights are represented as a large binary blob of 32 bit floating points for each layer's weight/bias values. We modify the source of *darknet* to export the weights in a format convenient for use with Python, comma-separated values (CSV). Parsing these large weight files (approximately 1GB) takes several minutes, so after they have been initialized in TensorFlow, we save a compressed TensorFlow *.ckpt* file for quick reload (available for download on our project's Github page).

5.2. Inference Validation

As the original YOLO paper serves only as a summary of YOLO, it omits many details. Details of the 2D convolution parameters and connection between convolutional and fully connected layers were taken from the *darknet* implementation. Validation and debug is drastically simplified in comparison to developing a new neural network model. Each input and output of our model can be compared with that of the *darknet* implementation to ensure correctness or discover bugs. For example, during development the weights were being read in an improper sequence at a particular convolutional layer. Comparing the output of the two network's allowed the problem to be quickly narrowed down and corrected.

We have validated the outputs of both networks match on the pre-trained model for the test image shown in Figure 3. We parse the prediction output in order to calculate and draw bounding boxes (see Figure 3). We achieve the same inference result in comparison with *darknet*, shown in Figure 4.

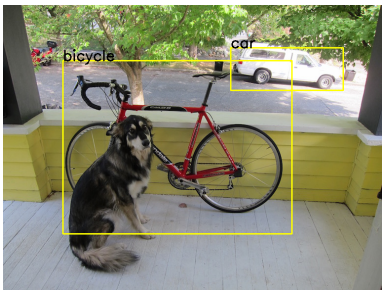

Figure 3. Using a data set of 5011 images from the PASCAL VOC 2007 Challenge, we train on 80% of the images with a dropout probability of 0.8 and a learning rate of $1 \cdot 10^{-5}$, and validate on 10% and test on 10%

.

[3] pjreddie.com/darknet/yolo/

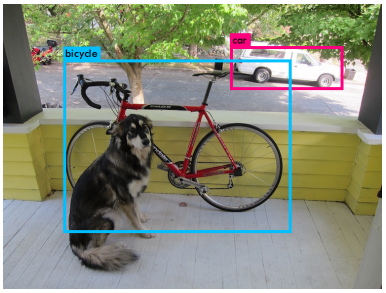

Figure 4. Redmon *et al.*'s [13] detection result on the same image.

5.3. Real-time object tracking

We extend YOLO to track objects within a video in real-time. YOLO is designed to process images in sequence; thus, it has no concept of temporal or spatial continuity between sequential frames in a video. For example, while video frames may be fed into YOLO sequentially, YOLO cannot determine which object detected in one frame corresponds to the same object in the next frame. To overcome this limitation, we introduce a post-processing phase illustrated in Figure 5.

We use k-means clustering to identify objects between frames. We choose a short rolling window of three frames across which we accumulate all the objects detected in that time period. We then cluster these objects into the appropriate number of clusters based on the number of objects detected per frame. As YOLO's detection capabilities are not perfect, certain objects may be intermittently dropped. The rolling window improves the chances that if an object is not detected in one frame, it will be detected in an adjoining frame to allow continuous tracking [4].

During clustering, we define the distance d between two images $\mathcal{I}_1, \mathcal{I}_2$, given dimensions x, y and color channels c as:

$$d(\mathcal{I}_1, \mathcal{I}_1) = \sum_x \sum_y \sum_c (\mathcal{I}_1^{(x,y,c)} - \mathcal{I}_2^{(x,y,c)})^2 \qquad (1)$$

In this case, $\mathcal{I}^{(x,y,c)}$ denotes the color intensity of the c color channel for pixel x, y.

This method of clustering works well if the images are very similar and bounding boxes do not change significantly in size. However, large offsets or changes in coloring (e.g. change in lighting) can create difficulty in identifying similar images. More sophisticated methods of measuring image similarity have been developed [17].

5.4. mAP Validation

As a sanity check on our implementation, we executed our model on the VOC 2007 data sets. Using a learning rate

[4] A sample video is provided at: https://drive.google.com/file/d/0BzlId5XmJ-sNajdWalI0cEtyQ0k/view?usp=sharing

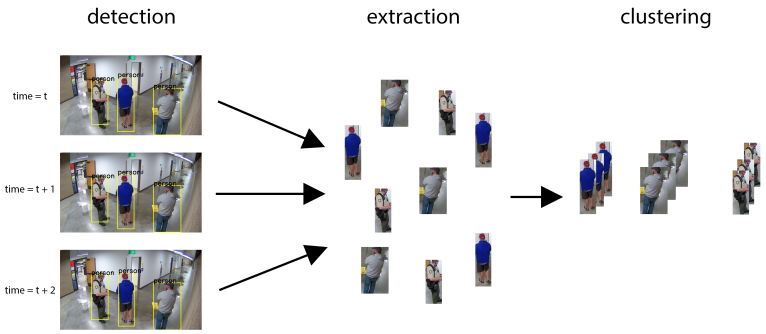

Figure 5. Clustering objects across time for real-time tracking

of $\alpha = 1 \cdot 10^{-2}$, along with initializations of weights drawn from a random normal distribution $\sim \mathcal{N}(\mu, \sigma = 0.35)$ immediately yielded NaNs for loss values, suggesting divergence. Presuming that the norm of the weights was too large during initialization, we used Xavier initialization [5]. This solution resolved the problem as long as the magnitude of the learning rate α was kept low (e.g. $\alpha = 1 \cdot 10^{-5}$). We qualitatively verified that mean average precision was high between predicted and ground truth bounding boxes.

5.5. Speed Testing

Speed is one of the most limiting factors in the ability to produce detections in video with high frame rate. We are interested in investigating how the convolution, max pooling, and matrix multiplication operations in TensorFlow compare with those implemented in C/C++ and in CUDA for the *darknet* framework. A speed comparison benchmark between various applications and implementations is pictured in Figure 8.

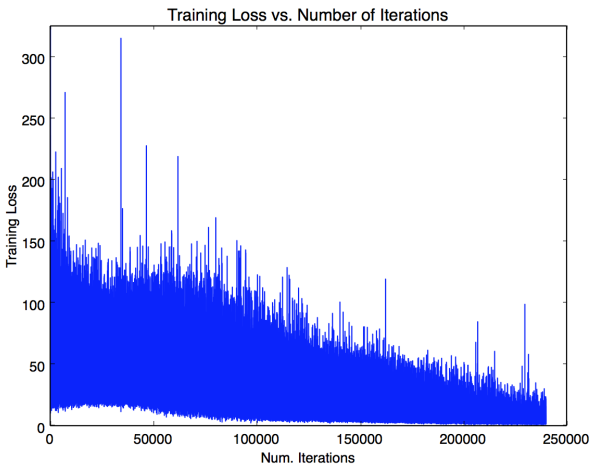

Figure 6. Loss convergence under *Interpretation 1* of the loss function.

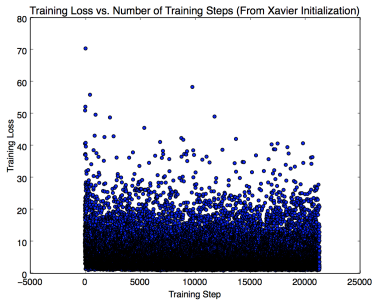

Figure 7. Model loss versus number of training iterations under *Interpretation 2* of the loss function. Note that this model does not converge.

6. Test Set Evaluation

Modern object detection challenges rely upon a metric called *mean average precision* (mAP). We compute the average precision (AP) separately for each class by sorting the detections by their confidences and moving down the sorted list, and then subsequently average over the APs for each class to compute mAP. A detection is considered a true positive if it has IoU with a ground-truth box greater than some threshold (usually 0.5) (mAP **@**0.5) We then combine all detections from all test images to draw a precision / recall curve for each class; AP is the area under the curve [3], computed via the Riemann sum:

$$AvgPrecision = \sum_{k=1...N} P(k)\Delta recall(k)$$

where $P(k)$ is the precision at every possible threshold value, $\Delta r(k)$ is the change in recall, and k takes on every possible recall value found in the data. To compute mAP, we use:

$$MAP = \frac{\sum_{q=1}^{Q} AvgPrecision(q)}{Q}$$

for Q number of classes. For PASCAL VOC, $Q = 20$.

Precision captures how accurate the reports are given by the algorithm:

$$Precision = \frac{M_{ii}}{\sum_k M_{ji}} = \frac{TP}{TP + FP}$$

Recall measures how many ground truths can be found by the algorithm:

$$Recall = \frac{TP}{TP + FN}$$

In PASCAL VOC, a bounding box reported by an algorithm is considered correct if its area intersection over union with a ground truth bounding box is beyond 50%. [12]. Formally, detections are considered true or false positives based on the area of overlap with ground truth bounding boxes. To be considered a correct detection, the area of overlap a_o between the predicted bounding box B_p and ground truth bounding box B_{gt} must exceed 50% by the formula[1]:

$$a_0 = \frac{areaB_p \cap B_{gt}}{areaB_p \cup B_{gt}}$$

We use the more recent calculation of average precision [2] [11] that does not evaluate the area under the precision-recall curve at 11 points $\in \{0, 0.1, ..., 0.9, 1\}$ but rather at all data points.

The precision value $precision(k)$ is the proportion of samples with rank smaller or equal than $k - 1$ that are positive (where ranks are assigned by decreasing scores). $recall(k)$ is instead the percentage of positive samples that have rank smaller or equal than $k - 1$. For example, if the first two samples are one positive and one negative, $precision(3)$ is $\frac{1}{2}$. If there are in total 5 positive samples, then $recall(3)$ is $\frac{1}{5}$.

Moving from rank k to rank $k + 1$, if the sample of rank $k+1$ is positive, then both precision and recall increase; otherwise, precision decreases and recall stays constant. This gives the PR curve a characteristic saw-shape. For an ideal classifier that ranks all the positive samples first the PR curve is one that describes two sides of the unit square [16]

6.1. Loss Convergence Analysis

In our model implementation, we limit the number of objects that can be penalized in boxes $j = 0, .., B$ to one. Redmon *et al.* do not enforce such a limitation, but within the context of a computational graph and without explicit tensor slice assignment capabilities, the composition of required Boolean functions on tensors in order to implement this functionality is not immediately obvious. Thus, we pass in only one ground truth bounding box per grid cell at a time into the loss function, instead of two.

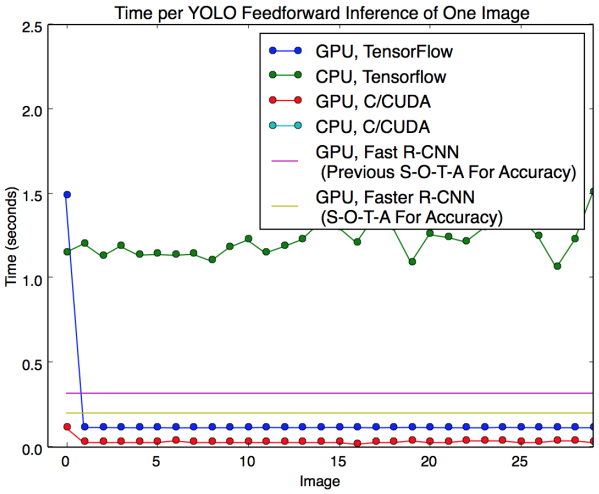

Figure 8. A comparison of inference time required per detection algorithms.

Speed Comparison of Feedforward Inference
(Averaged over 30 Image Passes)

	Time on CPU (seconds)	GPU (seconds)
Baseline YOLO CUDA/C Implementation	6.80918921676	0.0348387086452
TensorFlow YOLO Implementation	1.2247150739	0.162782869031

Figure 9. Feedforward Inference Speed on GPU and 3.1 GHz CPU machines of TensorFlow versus CUDA/C implementations

7. Conclusion and Future Work

We have demonstrated and verified a functional port of YOLO from *darknet* to TensorFlow. More work is needed to improve the robustness of the image clustering for real-time video tracking. For example, the existing implementation is limited in the scenarios it can successfully track; image fidelity and frame rate must be high, and objects cannot move at high velocity. More complex algorithms used to calculate image similarity as discussed by [17] would improve tracking capability.

In further work, we intend to more deeply examine the time tradeoff between GPU inference with varying models. We posit that our batch size of 1 limits our comparison because most of the time may be accounted for by memory swaps on and off of the GPU.

In future work, we intend to resolve the ambiguities in the original loss function by contacting the original authors of [13] and report the mAP across all data set splits of our self-trained model.

References

[1] Mark Everingham and John Winn. *The PASCAL Visual Object Classes Challenge 2012 (VOC2012) Development Kit*. 2012. URL: http://host.robots.ox.ac.uk/pascal/VOC/voc2012/VOCdevkit_18-May-2011.tar.

[2] Mark Everingham et al. "The Pascal Visual Object Classes Challenge a Retrospective". In: *International Journal of Computer Vision (ICJV)* (2014). URL: http://arkitus.com/files/ijcv-14-everingham-pascal.pdf.

[3] M. Everingham et al. "The PASCAL Visual Object Classes (VOC) Challenge". In: *International Journal of Computer Vision* 88.2 (2010), pp. 303–338. URL: http://host.robots.ox.ac.uk/pascal/VOC/pubs/everingham10.pdf.

[4] Ross B. Girshick. "Fast R-CNN". In: *CoRR* abs/1504.08083 (2015). URL: http://arxiv.org/abs/1504.08083.

[5] Xavier Glorot and Yoshua Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS10). Society for Artificial Intelligence and Statistics*. 2010.

[6] Jonathan Huang et al. "Speed/accuracy trade-offs for modern convolutional object detectors". In: *CoRR* abs/1611.10012 (2016). URL: http://arxiv.org/abs/1611.10012.

[7] Diederik P. Kingma and Jimmy Ba. "Adam: A Method for Stochastic Optimization". In: *CoRR* abs/1412.6980 (2014). URL: http://arxiv.org/abs/1412.6980.

[8] Min Lin, Qiang Chen, and Shuicheng Yan. "Network In Network". In: *CoRR* abs/1312.4400 (2013). URL: http://arxiv.org/abs/1312.4400.

[9] Wei Liu et al. "SSD: Single Shot MultiBox Detector". In: *CoRR* abs/1512.02325 (2015). URL: http://arxiv.org/abs/1512.02325.

[10] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: http://tensorflow.org/.

[11] Sancho McCann. *Average Precision*. URL: https://sanchom.wordpress.com/tag/average-precision/.

[12] *Princeton University COS429 Computer Vision Final Project: PASCAL VOC Classification Challenge*. URL: http://vision.princeton.edu/courses/COS429/2015fa/ps/final_project/final_project.pdf.

[13] Joseph Redmon et al. "You Only Look Once: Unified, Real-Time Object Detection". In: *CoRR* abs/1506.02640 (2015). URL: http://arxiv.org/abs/1506.02640.

[14] Shaoqing Ren et al. "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks". In: *Advances in Neural Information Processing Systems (NIPS)*. 2015.

[15] Christian Szegedy et al. "Going Deeper with Convolutions". In: *CoRR* abs/1409.4842 (2014). URL: http://arxiv.org/abs/1409.4842.

[16] *Tutorials: Plotting AP and ROC Curves*. 2013. URL: http://www.vlfeat.org/overview/plots-rank.html.

[17] Liwei Wang, Yan Zhang, and Jufu Feng. "On the Euclidean distance of images". In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 27.8 (2005), pp. 1334–1339.