Efficient few-shot learning for object detection through transfer learning and data augmentation using Tiny-YOLOv3

https://github.com/Tobias-Rohde/FewShotObjDet

Tobias Rohde, CSE599G1: Deep Learning

Introduction

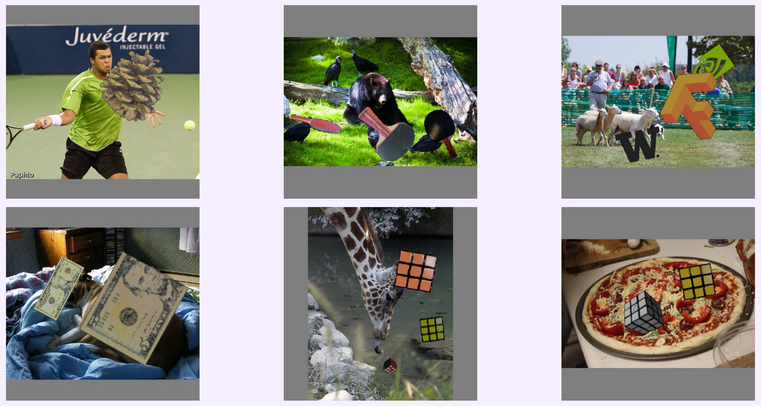
Training object detection models is a time consuming and potentially expensive task, especially if one wants to train a detector for an object that does not already exist in a publicly available dataset. One needs many pictures of the object in different surroundings and needs to annotate the objects with class id and bounding box information. This motivates few-shot learning approaches, where one attempts to train a detector with as little as a single image of a new object.

In this project I present an efficient approach which relies on transfer learning and on generating artificial training images using heavily augmented versions of new objects.

Data and Augmentation

I am using 3000/500 training/validation split from the COCO validation set as my base dataset. I have mostly used a Rubik's Cube as a test object.

Examples: Rubik's Cube, Pinecone, Dollar bills, Table tennis paddle, Logos



Augmentation in detail:

1. Uniformly select a base image, and resize it to 416x416, letterboxing to keep scale
2. With probability 1/10, don't insert any object. (*This brought down the objectness loss)*
3. Uniformly choose 1, 2 or 3 objects to insert
4. For each object:
   1. Uniformly **resize** object such that its smaller side length takes up between 1/10 and 1/2 of the smaller side length of the base image
   2. With probability 1/2, **flip** object **horizontally**
   3. Randomly **shift Hue** of object by between -32/360 and 32/360 in HSV color space. *(This helped prevent overfitting to colors)*
   4. Random **gamma correction** with gamma between 1/3 and 3/2. (*This improved performance under different lightning conditions)*
   5. With probability 1/5, make image **gray scale**
   6. Randomly **rotate** object between 0 and 359 degrees. (*This improved performance with different camera angles)*
   7. With probability 1/8, apply **blur**. (*This improved performance when detecting while moving the camera)*
      1. With probability 1/2 perform **motion blur**
         1. With probabilities 1/4 each, perform horizontal, vertical, top-left to bottom-right diagonal or bottom-left to top-right diagonal blur. Kernel size is chosen uniformly to be either 3 or 5.
      2. Otherwise perform regular **box blur**
   8. Ensure that there is not significant overlap between inserted objects, which hurts performance. I am using "intersection over minimum area" instead of IOU
   9. Keep track of the bounding boxes of the object and accumulate in **yolo_n_truth** and **yolo_truth_boxes** tensors for training *(See network architecture)*

Network Architecture and Transfer Learning

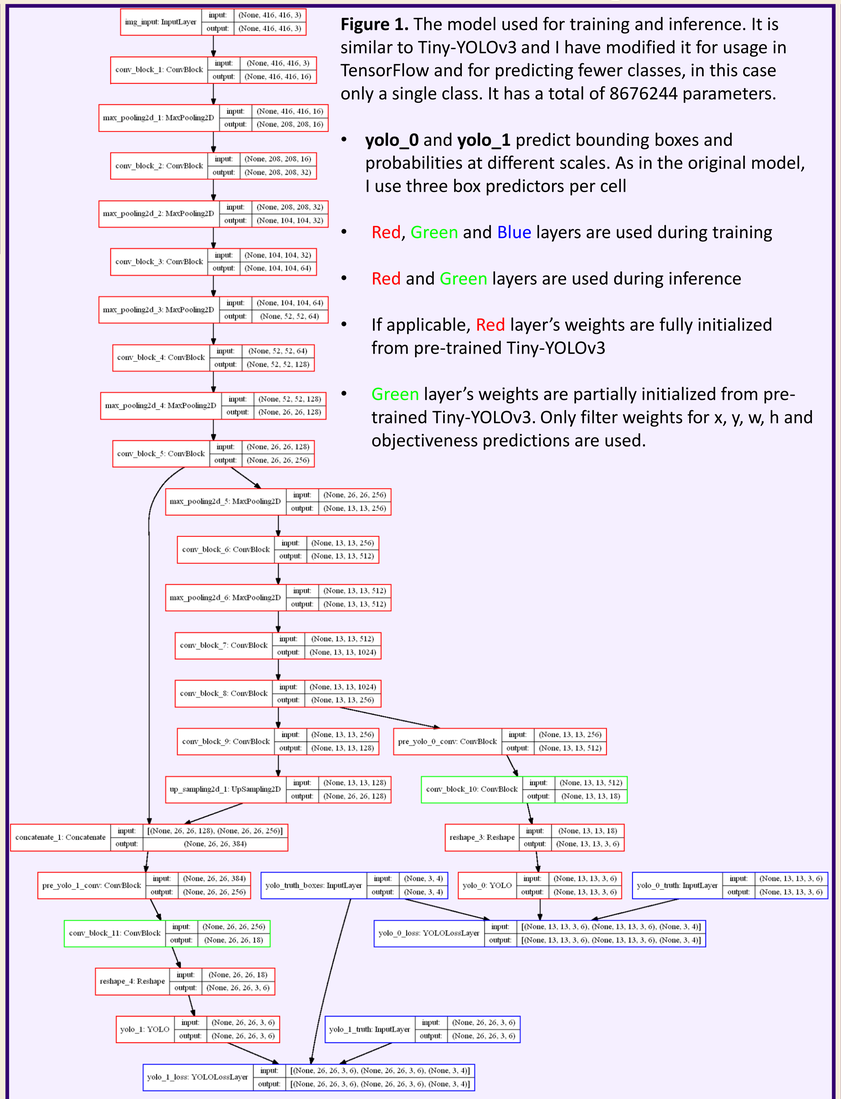


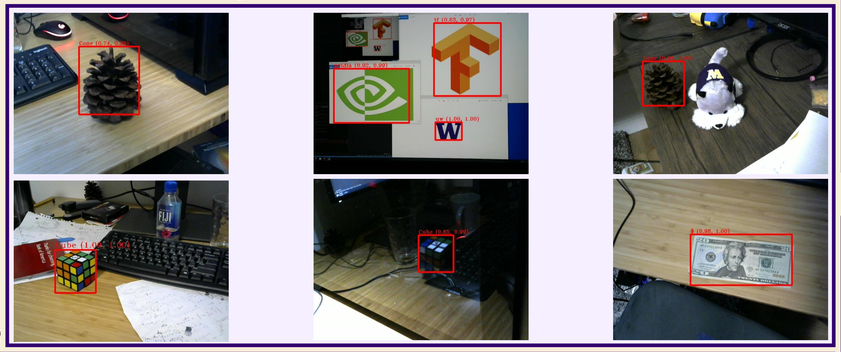
Figure 1. The model used for training and inference. It is similar to Tiny-YOLOv3 and I have modified it for usage in TensorFlow and for predicting fewer classes, in this case only a single class. It has a total of 8676244 parameters.

- **yolo_0** and **yolo_1** predict bounding boxes and probabilities at different scales. As in the original model, I use three box predictors per cell
- Red, Green and Blue layers are used during training
- Red and Green layers are used during inference
- If applicable, Red layer's weights are fully initialized from pre-trained Tiny-YOLOv3
- Green layer's weights are partially initialized from pre-trained Tiny-YOLOv3. Only filter weights for x, y, w, h and objectiveness predictions are used.

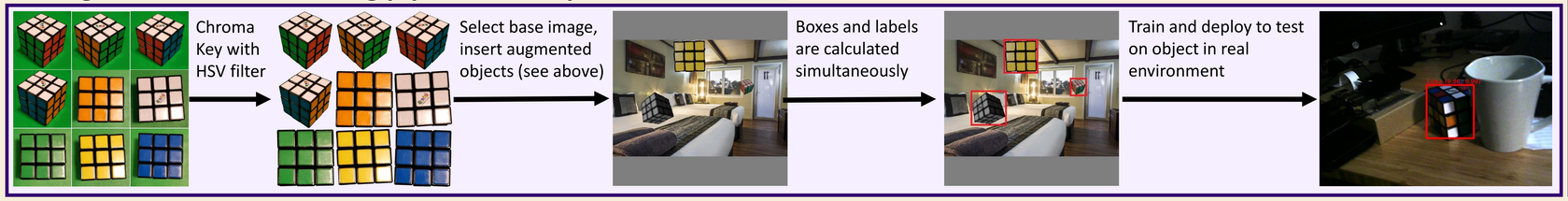
Benchmarks for predicting single image at a time (neglecting camera latency):

- **180 FPS** on a TITAN RTX (TF 2.0)
- **7 FPS** on a Jetson Nano (TF 1.14, no TensorRT)

Prediction examples



Data generation and training pipeline example: Rubik's Cube



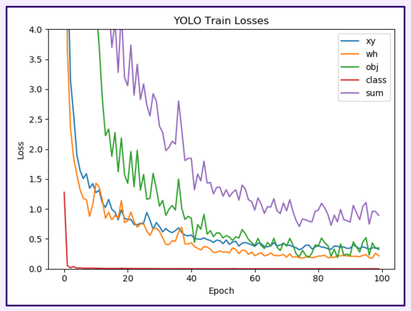
Training and Inference

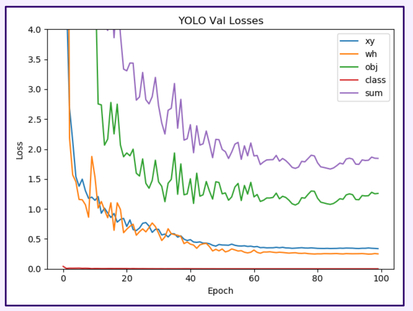
I built a data generator that generates new samples on the fly. Before training, a single validation set is generated to ensure a consistent performance metric. The training generator generates entirely new samples each epoch, making overfitting unlikely.

I use **Adam** with an **initial learning rate** of 0.002 and **weight decay** of 0.0005. I use **learning rate decay** with patience 5 epochs and a minimum learning rate of 7.8125e-6. I am using an **early stopping** condition with patience 15 epochs and train for at most 128 epochs. Training usually takes around 1 hour on a TITAN RTX.

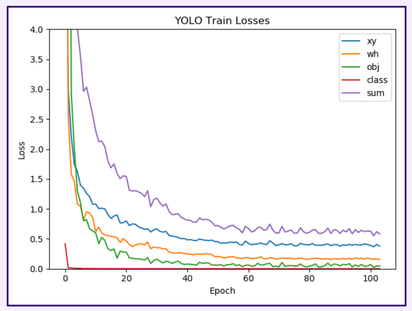
The loss is a 4-part function, penalizing the x,y predictions, w, h predictions, objectness and class probability predictions separately. For calculating the objectness loss, I use an IOU threshold of 0.6 when determining if a prediction matches a truth box. During training, I plot the 4 losses separately to analyze how the different components affect the loss and to reduce the overall loss effectively. I also look at the loss for single images to tune augmentation.

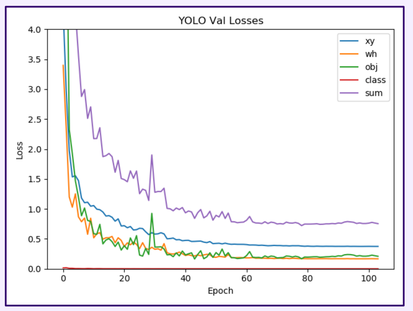
Below are two of the earlier training plots for the model trained on a new object:





One can see that the objectness does not lower after 60 epochs. By targeted adjustment of the augmentation and the loss function, I managed to decrease the objectness loss and thus the overall loss. I also improved optimizer hyperparameters to make training smoother:





For inference, I am using an objectness and class probability threshold of 0.5 combined with a modified version of non-max-suppression (0.45 threshold), in which I additionally use intersection over min area (0.9 threshold) to eliminate small boxes contained in larger boxes.

It is difficult to compare performance across models, since the validation set is always different. To compare models, I generate several validation sets and evaluate each model on each set. I then rank them according to their total loss.

Conclusion and Future direction

The combination of transfer learning and data augmentation is an effective and efficient approach to few-shot object detection. However, the approach can not compete with more complex techniques such as Attention based region proposal networks or feature reweighting. This is likely because the objects do not integrate well into the base images and the network can not learn about the context the object appears in. This is a limitation of my approach that is difficult to overcome, since it operates at the input level rather than feature level.

I want to see if it is possible to learn an encoding for images and objects and then learn to embed the encoded objects in the encoded image more naturally in feature space. Additionally I would like to try more advanced augmentation techniques to make the detectors more robust. To make the model more efficient, I could try network binarization. I also want to automate the chroma keying process.