CS230

Exploring Iterative Pruning in Deep Convolutional Neural Networks

Brian Jackson*
Department of Mechanical Engineering
Stanford University
bjack205@stanford.edu

Abstract

This project demonstrates transfer learning with the YOLOv2 network. The network is successfully retrained on a portion of the KITTI autonomous driving benchmark dataset, achieving a precision of 70% and recall of 83% on car detection when fully retrained. The data is retrained using both COCO labels and KITTI labels. The project set out to explore iterative pruning of the network, and the algorithm for doing so is set forth, but initial attempts to achieve good results have been unsuccessful. Lastly, the project presents a framework to reduce the barrier to entry for applying transfer learning to the YOLOv2 network within the YAD2K framework. This is done to help future projects focus more on novel techniques and less time trying to understand the details of the retraining process for this common Keras implementation of YOLOv2.

1 Introduction

Deep learning has opened up exciting avenues in the fields of computer vision, health care, autonomous navigation, and automated decision-making algorithms, where it has shown remarkable success. This success, however, has often come at considerable computational cost. Even with advances in GPU computing the computational and storage expenses of employing these systems are significant. Decreasing the computational and storage costs of neural networks will enable deployment of neural networks to smaller, computationally-limited platforms—such as mobile devices—where the benefits of neural networks have previously been unavailable.

This project focuses on the problem of image detection and its application to the domain of autonomous vehicles. The primary objective of this project is to investigate iterative pruning techniques on the state-of-the-art YOLOv2 object detection network. The first phase of this project is to retrain the network on KITTI, a realistic autonomous driving data set, using the YAD2K Keras/Tensorflow implementation of YOLOv2 [5], [9], [12]. Another primary contribution of this project to the deep learning community is a set of well-documented functions for applying transfer learning within the YAD2K framework, which suffers from poor documentation and poor assumptions about the incoming data. After retraining of the network, this project investigates iterative pruning to decrease the size of the network.

Recent work done in the Darve Group here at Stanford has shown that a significant amount of the weights of a Deep CNN can be removed with only marginal effects on performance (in fact, about 90% of the weights of the SSD network were removed with only a 2-3% drop in accuracy). However, after a certain threshold any additional pruning had dramatic effects on the accuracy of the network. I

*

CS230: Deep Learning, Winter 2018, Stanford University, CA. (LateX template borrowed from NIPS 2017.)

hope that by applying a similar technique and investigating the significance of the remaining edges I can gain further insight into more efficient neural network topologies.

The objectives of this project are three-fold: 1) retrain YOLOv2 on the KITTI dataset, a realistic and challenging autonomous driving benchmark, 2) develop tools and resources to reduce the barrier to entry for future transfer learning projects using the YAD2k framework, and 3) explore the effects of iterative pruning on the YOLOv2 algorithm. The inputs to this algorithm are images, and the output are bounding boxes and class predictions for a defined set of objects in the image.

2 Related work

Image detection is an extremely hot topic in deep learning, resulting primarily from the recent push to develop autonomous vehicles. VGG-16 [4] was perhaps one of the greatest breakthroughs in object classification and is still used as a baseline architecture for many object detectors. This seminal paper demonstrated the positive benefits of increasing the depth of convolutional networks and set the current trend towards DCNNs. Fast R-CNN [6] used region proposals and started condensing the detection pipeline to get substantial performance gains. The original YOLO [8] ran to the extreme with this idea and condensed everything into a single pipeline, with amazing performance gains but marginal accuracy. SSD [10] and YOLOv2 [11] improved upon the original ideas set forth in YOLO to achieve start-of-the-art performance in object detection in both speed and accuracy.

While some groups are focused on generating state-of-the-art detectors, other have focused on the increasingly large computational requirements of the resultant networks. One approach is to use various techniques to reduce the size of an existing network, similar to the approach taken in this project [7]. Han et. al. use a combination of pruning, trained quantization, and Huffman coding to achieve compression of 35x to 49x on image classifiers. Another approach is to rethink the architecture. Novikov et. al. propose using the Tensor Train format to compress fully-connected layers by 200,000x, resulting in a network compression of 7x. Both approaches show promise, and demonstrate that there is a lot of potential to condense the information encoded within a neural network.

3 Dataset and Features

The Karlsruhe Institute of Technology and the Toyota Technological Institute captures camera data from a car driving around urban and rural areas in a German town. The car was equipped with stereo grayscale and color cameras, a Velodyne Lidar, and a high-precision GPS-IMU localization unit. The data has been labeled and calibrated to overlay the image data onto the point cloud data from the Lidar, providing a reasonable estimation of ground truth. The entire data set includes many different benchmarks relevant to the domain of autonomous driving, such as odometry, optical flow, road detection, depth prediction, and 2D and 3D object detection. I will be using the 2D object detection dataset and its associated tools [2], [3].

The dataset consists of about 7500 images in both the training and the test sets. To reduce computational requirements I used 1000 images, with 100 images in the test set and 100 images in the dev set. The images are of about 1200x370 resolution, after rectification (resolutions vary due to rectification). The images have been classified into 9 categories: `'Car'`, `'Van'`, `'Truck'`, `'Pedestrian'`, `'Person_sitting'`, `'Cyclist'`, `'Tram'`, `'Misc'` or `'DontCare'`. They have also been categorized as truncated (part of the object is outside the image) or occluded (with varying levels of occlusion by other objects in the scene). This project focuses on just using the class and bounding box information, but further work may benefit from including this additional information while training.

Since the YOLOv2 algorithm is fully convolutional, it should work with any arbitrary image size. However, when the image aspect ratio is significantly different from the square images used to train the YOLOv2 network, the default anchor boxes are inappropriate. To simplify the training process, the anchor boxes were not modified, and the images were re-scaled in an input resolution of 608x608 and the pixel values were scaled to be between 0 and 1. More details on input data preparation are given in Appendix A.1.

4 Model

The YOLOv2 CNN is the current state-of-the-art framework for object detection, and is both fast and accurate. Importantly, the model is fully-convolutional, which means any image resolution can be fed into the algorithm. The bulk of the body uses convolutional and max-pooling layers with batch normalization, and includes a pass-through connection from the middle of the network to one of the last layers (similar to a ResNet connection). The final layer is a 1x1 convolutional layer that maps the resulting grid into class and bounding box predictions.

5 Methods

5.1 Retraining the Network

Retraining the network turned out to be an unexpectedly time-intensive portion of the project. This process took such a significant amount of time it left little to no time for the more interesting parts of my original proposal. In an effort to make some valuable contribution to the deep learning community, I developed some intuitive, object-oriented, and well-documented code for retraining YAD2K on a custom dataset. A brief description is included in Appendix A.

5.1.1 Running the Retraining Algorithm

Two different approaches were taken to retrain the model on the KITTI dataset. In all cases, the original weights were used as a starting point. The first approach was to simply use the COCO classes and create a mapping between the COCO classes and the KITTI classes. This mapping is given in Table 1. This retraining process was done in two steps: the first step was to fine-tune the old weights by only changing the top layer for 5 epochs (after this point the dev-loss started increasing), followed by a subsequent training of the entire network for 50 epochs.

The second approach was to discard the weights of the top layer entirely in order to train on the KITTI classes. For this approach, only the top layer was retrained: the rest of the model was left with the original weights. Given more time, better performance could likely be achieved by training lower weights as well. This approach was trained for 500 epochs.

The YAD2K uses the Adam Optimizer within the Keras framework. All the hyperparameters such as the learning rate, beta values, and epsilon were left at their default values. The only hyperparameter that was changed was the batch size, which was set based upon the computing platform available (whether my computer (4) or the Stanford Sherlock cluster (16)).

5.1.2 Testing the Model

Since the original YAD2K implementation did not calculate any metric values, or have any way of comparing the predicted with the expected results, new testing functionality also had to be added. The true positives (TP), false positives (FP), and false negatives (FN) were calculated using Algorithm 1. These values were added for all images and the precision and recall were calculated for each class: $P = \dfrac{TP}{TP + FP}$ for precision, and $R = \dfrac{TP}{TP + FN}$ for recall.

5.2 Pruning the Network

The pruning algorithm was to work as follows: all the trainable weights of the convolutional layers of the network are pooled together and the value for the bottom $n\%$ is calculated (using `np.percentile`, for instance). Then any weight below this value is set to zero using a mask that is computed before the training begins. After each step of gradient descent, the mask is re-applied so that the canceled weights remain zero. The model is then retrained with the canceled weights. The goal was to iteratively prune more weights by removing a larger percentage after each retraining.

Result: Write here the result
matches$_j = 0 \, \forall j$;
$TP, FP, FN = 0$;
foreach *i, p_box in predicted_boxes* **do**
  foreach *t_box in true_boxes* **do**
    $iou_j = IOU(p_box, t_box)$
  end
  $k = \arg\max_j \text{iou}_j$;
  matches$_k$ += 1 ;
  if *true_class$_k$* $== class$ *and* $iou_k > 0.5$ **then**
    TP += 1
  else
    FP += 1
  end
end
foreach *j, t_box in true_boxes* **do**
  if *matches$_j$* $== 0$ **then**
    FN += 1
  end
end

Algorithm 1: Calculation of TP, FP, FN for a single image

KITTI	Car	Van	Truck	Pedestrian	Person_sitting	Cyclist	Tram
COCO	car	car	truck	person	person	bike	train

Table 1: Mapping between KITTI and COCO classes

6 Experiments/Results/Discussion

6.1 Retraining the Network

The precision and recall values for each of the approaches is shown in Table 2. A reasonable detection using COCO classes is shown in Figure 1. As shown in the table, there isn't a single approach that works better than the others. However, it is surprising that training the entire network didn't really help improve the results that much. This is likely due to the fact that the network is being trained on a relatively small amount of data and ends up overfitting the training set. Overfitting clearly happens when fine-tuning, as shown by the increase in the dev-loss shown in Figure 2a, but it's effect is marginalized by selecting the weights from the epoch with the lowest dev-loss.

Fine-tuning the network using the COCO classes converged much faster than training the network with the KITTI classes, as expected. Performance was also better on the large majority of the classes in both precision and recall. This suggests that when fine-tuning needs to happen quickly, the best option is to use the existing classes and simply map the classes in the new dataset to the existing classes and fine-tune only the top layers.

Another interesting challenge with the KITTI dataset is the unequal distribution of the classes: there are significantly more cars than any of the other classes. As shown by the results, the car class has fairly reasonable numbers, where it's not uncommon for other classes to have 0 or 100 % precision or recall, given the relatively small numbers of objects. Again, training and testing on a larger portion of the dataset should help with this issue.

It's quite possible that there is a "sweet spot" for the number of layers to retrain. Instead of training the entire network or just the top layer, perhaps only training the top 5 layers will give the best results, since the low-level features of the earliest layers shouldn't need to change. By allowing more depth to the learning, however, the algorithm may be able to better encode the notion of the "Misc" and "DontCare" KITTI labels, which may be difficult for even humans to understand (they were not included in the report since they are not used as metrics for the KITTI benchmarks).

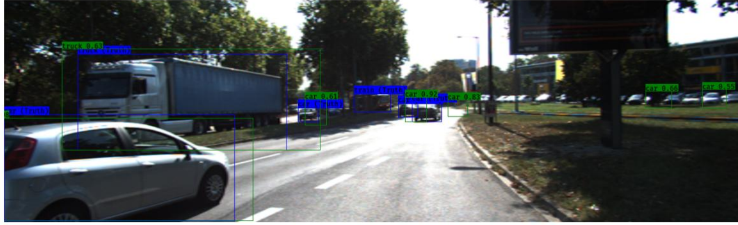
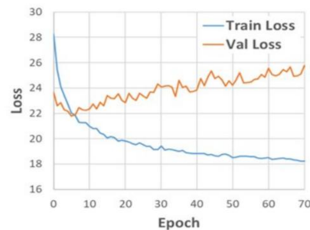
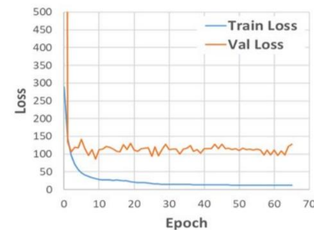


Figure 1



(a) Fine Tuning



(b) Full Train

Figure 2: Training losses for the retraining with COCO labels.

6.2 Pruning the Network

After retraining the network and only pruning a couple percent of the weights, the predictions were completely off (boxes lined the outer edges of the image). This is likely caused by an implementation detail and should be able to be resolved with more work. The developed framework for retraining with YAD2K should help provide better organization and aid in debugging these issues, but the issues were not able to be resolved within the time constraints of the project.

7 Conclusion/Future Work

This project demonstrates a successful retraining of the YOLOv2 algorithm on a portion of the KITTI dataset. When applying transfer learning to a new datasets, it appears the best solution when time and computational resources are limited is to map the new classes to the existing classes the network was trained on.

This project also serves as a valuable demonstration of retraining the YOLOv2 network with the YAD2K framework. Given that the original framework suffered from poor documentation, design decisions, and assumptions about the data, this project provides a new suite of classes designed to make the retraining process more straight-forward for users who may be new to Keras and Tensorflow. Given considerable amount of issues raised on the YAD2K GitHub repository regarding retraining, I believe this is a valuable contribution to the deep learning community and should allow future projects to focus more on novel work, instead of getting bogged down in the details of the retraining process.

There is a considerable amount of future work that can be done, including retraining the network on the entire KITTI dataset, applying data augmentation techniques, modifying the framework to use more information from the data labels such as levels of occlusion and truncation, and using the K-means clustering to define new anchor boxes for wide images. It would also be interesting to investigate the effect of freezing different numbers of layers and effect this has on the time to converge, precision, and recall.

The greatest area of future work is pruning the network. Despite all attempts being unsuccessful for this project, further work and debugging should solve the implementation issues and allow investigation of pruning the convolutional layers, hopefully yielding valuable insights about the distribution of weights within the minimal network that still provides descent performance.

	Person	Cyclist	Car	Train	Truck
Original	0.37 / 0.47	0.00 / 0.00	0.44 / 0.60	0.00 / 0.00	0.21 / **1.00**
COCO-FT	**0.70 / 0.53**	0.20 / 0.11	0.65 / 0.82	**1.00 / 0.11**	**1.00** / 0.67
COCO-Full	0.47 / 0.33	**1.00 / 0.23**	0.70 / **0.83**	0.00 / 0.00	**1.00** / 0.23
KITTI-FT	0.67 / 0.18	0.00 / 0.00	**0.97** / 0.53	0.00 / 0.00	0.83 / 0.36

Table 2: Precision and recall results for classes given in both the KITTI and COCO datasets. Precision is listed on the left, recall on the right. These values were calculated using a score threshold of 0.7 and an IOU threshold of 0.5. As stated in Algorithm 1, predicted boxes with an IOU greater than 0.5 with the true boxes were counted as true detections.

8 Contributions

The entire project was carried out by the author, with some directional help from my advisor, Dr. Eric Darve in the Stanford Mechanical Engineering Department. Some initial code for pruning the network was also contributed by Ziyi Yang, a member of the Darve Group.

Code

The code for the project is located at `https://github.com/bjack205/PrunedYOLO`. A cleaned-up version of the code for retraining with YAD2k is located at `https://github.com/bjack205/yad2k_retrainer`.

References

[1] A. Collette, *Hdf5 for python*, 2008.

[2] A. Geiger, P. Lenz, and R. Urtasun, "Are we ready for autonomous driving? the kitti vision benchmark suite," in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2012.

[3] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun, "Vision meets robotics: The kitti dataset," *International Journal of Robotics Research (IJRR)*, 2013.

[4] K. Simonyan and A. Zisserman, "Very Deep Convolutional Networks for Large-Scale Image Recognition," pp. 1–14, 2014, ISSN: 09505849. DOI: 10.1016/j.infsof.2008.09.005. arXiv: 1409.1556. [Online]. Available: http://arxiv.org/abs/1409.1556.

[5] F. Chollet *et al.*, *Keras*, 2015.

[6] R. Girshick, "Fast R-CNN," *Proceedings of the IEEE International Conference on Computer Vision*, vol. 2015 Inter, pp. 1440–1448, 2015, ISSN: 15505499. DOI: 10.1109/ICCV.2015.169. arXiv: 1504.08083.

[7] S. Han, H. Mao, and W. J. Dally, "Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding," pp. 1–14, 2015. DOI: abs/1510.00149/1510.00149. arXiv: 1510.00149. [Online]. Available: http://arxiv.org/abs/1510.00149.

[8] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," 2015, ISSN: 01689002. DOI: 10.1109/CVPR.2016.91. arXiv: 1506.02640. [Online]. Available: http://arxiv.org/abs/1506.02640.

[9] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, *et al.*, "Tensorflow: Large-scale machine learning on heterogeneous distributed systems," *arXiv preprint arXiv:1603.04467*, 2016.

[10] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C. Y. Fu, and A. C. Berg, "SSD: Single shot multibox detector," *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9905 LNCS, pp. 21–37, 2016, ISSN: 16113349. DOI: 10.1007/978-3-319-46448-0_2. arXiv: 1512.02325.

[11] J. Redmon and A. Farhadi, "YOLO9000: Better, Faster, Stronger," 2016, ISSN: 0146-4833. DOI: 10.1109/CVPR.2017.690. arXiv: 1612.08242. [Online]. Available: http://arxiv.org/abs/1612.08242.

[12] allanzelener, *Yad2k*, https://github.com/allanzelener/YAD2K.git.

A Brief Description of the YAD2K Retraining Framework

A.1 Preparing the Data

There are 4 expected inputs to the YAD2K model:

1. Image data: Numpy array of floats of shape $(m, h, w, 3)$, rescaled from 0 to 1.
2. Label data: Numpy array of floats of shape $(max_boxes, 5)$, where each row is in the format $[x, y, w, h, class]$, where x and y are coordinates to the center of the box, and all coordinates are given in decimal fractions of the image sizes.
3. Detectors Mask: Numpy array of booleans of shape $(g_h, g_w, num_anchors, 1)$, where g_h and g_w are the height and width of the output grid, respectively. The dimensions of the output grid are simply calculated by dividing the input image size dimensions by 32.
4. Matching True Boxes: Numpy array of floats of shape $(g_h, g_w, num_anchors, 5)$

To simplify the data preparation process, I wrote three Python classes: `DataExtractor`, `DataCompiler` and `Yad2kData`. The `DataExtractor` class is set up as a super class, where child classes are to implement the functionality that is unique to each dataset, and has liberal comments throughout about expected data types and formats. This class has built-in functions to generate the last two inputs, given the input labels (based off of code already in YAD2K).

Another major obstacle when using the YAD2K framework was the fact the original training function assumed all the training data could fit into memory. This precluded the use of large datasets like KITTI. I developed the framework to work with hdf5 [1] files and Python generators, so that the data can quickly be read in by batch into the training algorithm (see the Keras `fit_generator` function). The `DataCompiler` class uses the `DataExtractor` class to read the raw data and save it to an hdf5 file. This file is then the only input to the `Yad2kData` class that interacts directly with the training algorithm.

Lastly, the original YAD2K algorithm was not clear on how to handle arbitrary image sizes. Since the KITTI images have a very wide aspect ratio, it was not appropriate the treat them as square. An effective solution is to define two separate image sizes: `image_size`, the size of the desired output image, and `image_data_size`, a square size, divisible by 32, that will is fed into the network for training the prediction.

A.2 Running the Training Algorithm

Once the data preparation steps are complete, it is fairly straight-forward to use the newly implemented `Trainer` class to train the network. It should be noted that the Keras model with the Lambda loss function should be used for training. The model must be modified to be used for testing.