Residual Networks for Vehicle Detection

1 INTRODUCTION

One of the major factors in the recent increase of interest in deep learning is the development and successful application of convolutional neural networks (CNNs) to image recognition. Further development of these networks has led to unprecedented performance in object detection as well, using a family of techniques referred to as region-based convolutional neural networks (R-CNNs) [1]. The ability to classify and locate objects in an image can be useful particularly for self-driving vehicles, which need to track the positions of surrounding objects they could potentially collide with.

In this project, I combine two recent neural network architectures, Faster R-CNN for object detection [4] and deep residual networks for image recognition [5], to train an object detector that detects objects useful for self-driving vehicles. The detector is implemented using Keras and trained on the KITTI dataset for 2D object detection [7].

1.1 Problem Statement

Given an image, locate and classify objects of interest.

Objects are classified into the following categories, as defined in the KITTI dataset [7]: *car*, *person*, *Cyclist*, *DontCare*, *Misc*, *Person_sitting*, *Tram*, *Truck*, and *Van*. The classification output consists of one of these classes and a score from 0.0 to 1.0 representing how confident the model is that its predicted object class is correct.

For each object detected, its location is specified as the coordinates of the top left and bottom right corners of a bounding box containing the object. The output of the detector can then be used to annotate the image and visualize the results. Figure 1 shows the output of one model (*ResNet-50* with no weight regularization) on an image from the KITTI test set.

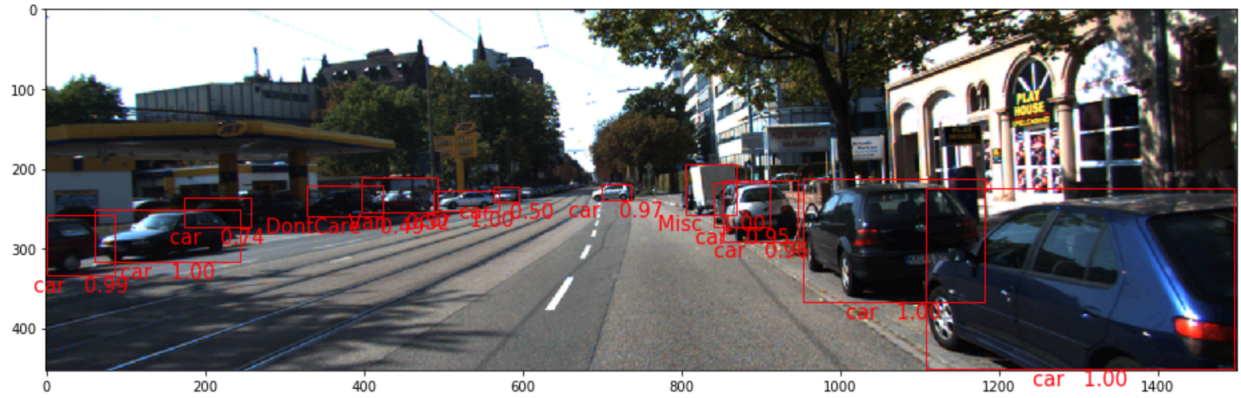

Figure 1: objects detected in image 000018 from the KITTI test set.

1.2 Metrics

The standard evaluation metric for objection detection is the mean Average Precision (mAP) as defined in the PASCAL Visual Object Classes (VOC) Challenge 2007 [6]. The value is calculated by generating predictions for each image in a validation or test set, finding the average precision (AP) for each class independently, and taking the mean across all classes.

To calculate the AP for a class, each object detection of that class is ranked from highest confidence to lowest and classified as either a true positive (TP) or a false positive (FP). The criteria for a true positive detection is that its bounding box has an intersection-over-union (IOU) overlap of at least 0.5 with the bounding box of a ground truth object that has not already been matched with a higher-ranking detection. By counting already-matched detections as false positives, the metric encourages algorithm developers to de-duplicate multiple detections of the same object. With each detection classified, precision and recall values are calculated for each rank *r* using the cumulative TP and FP counts of the detections from rank 1 to rank *r*. Using these cumulative precision and recall values, the AP is the mean of 11 numbers: the max cumulative precision for ranks where the recall is at least 0.0, the same when recall is at least 0.1, all the way up to 1.0. Mathematically:

$$AP = \frac{1}{11} \sum_{r \in \{0.0, 0.1, \ldots, 1.0\}} \max_{\tilde{r}: \tilde{r} \geq r} p(\tilde{r})$$

where *p(r)* is the precision at recall *r*. Using this metric as the measure of success is common in object detection because it ensures that high-scoring algorithms have high precision at every level of recall. Later editions of the PASCAL VOC challenges use a different definition of the metric, but this project uses the 2007 definition.

In addition to the mAP, I will also measure the average time taken to produce object detections for an image, in seconds. The more quickly the detector can produce results, the more useful it is in real-time contexts such as autonomous vehicles.

2 ANALYSIS

2.1 Data Exploration

The models developed are trained on two public image sets for object detection: the combined datasets from the 2007 and 2012 editions of the PASCAL VOC Challenge [6] for general object detection, and the KITTI 2D object detection image set [7].

The PASCAL VOC 2007 dataset includes 2501 images in the training set, 2510 in the validation set, and 4952 in the test set. Images have varying sizes, but are generally around 500 pixels in width and 375 pixels in height. Objects are to be classified into 20 classes: *aeroplane*, *bicycle*, *bird*, *boat*, *bottle*, *bus*, *car*, *cat*, *chair*, *cow*, *diningtable*, *dog*, *horse*, *motorbike*, *person*, *pottedplant*, *sheep*, *sofa*, *train*, and *tvmonitor*.

Each image comes with annotations with metadata about the image and the objects in it. For object detection purposes, the relevant metadata include the following fields:

- The height of the image in pixels.
- The width of the image in pixels.
- A list of the objects in the image and useful metadata about each:
    - The object's class. One of the 20 listed above.
    - The *x* and *y* coordinates of the top left and bottom right corners of the object's bounding box. The *x* and *y* values are pixels from the left and top sides of the image respectively.
    - Whether the object is considered "difficult" to detect. Difficult objects are useful for training but exempt from the mAP calculation.

Figure 2: images 000071, 000076, 000084, and 000088 from the PASCAL VOC 2007 set.

The 5011 images in the *trainval* set (training and validation combined) contain a total of 15662 labeled objects.

The PASCAL VOC 2012 dataset contains an additional 11540 *trainval* images with 31561 objects, and is like the 2007 dataset in all other regards. It enhances the training data with more diverse samples to decrease the likelihood of overfitting.

The KITTI 2D objection detection dataset includes 7481 annotated training images containing 51865 objects. There are also 7518 test images with no annotations. To evaluate network performance, I split the training images into training and validation sets such that the 2/3 of the images with names from *000000* through *004993* inclusive are used for training and the remaining 1/3 are used for validation. Most images have a width of 1242 pixels and a height of 375 pixels, with the rest being slightly different in size.

2.2 Exploratory Visualization

Figures 3 and 4 show example images with ground truth bounding box annotations from the PASCAL VOC and KITTI data sets respectively. In general, PASCAL VOC images contain fewer, larger objects. KITTI images tend to contain numerous, smaller objects. Though it is difficult to draw such a conclusion from a few examples, a broader look at object size statistics shows that the observation holds for not just these examples but generalizes across all images in both sets. Figure 5 shows ground truth object height/width distributions for the PASCAL VOC and KITTI *trainval* sets when images are resized as described in section 3.1. The object size distribution is the most important difference between these two datasets to take into account when adjusting models for KITTI evaluation in section 3.3.

Figure 3: images 003154 and 002374 with ground truth labels from the PASCAL VOC 2007 set.

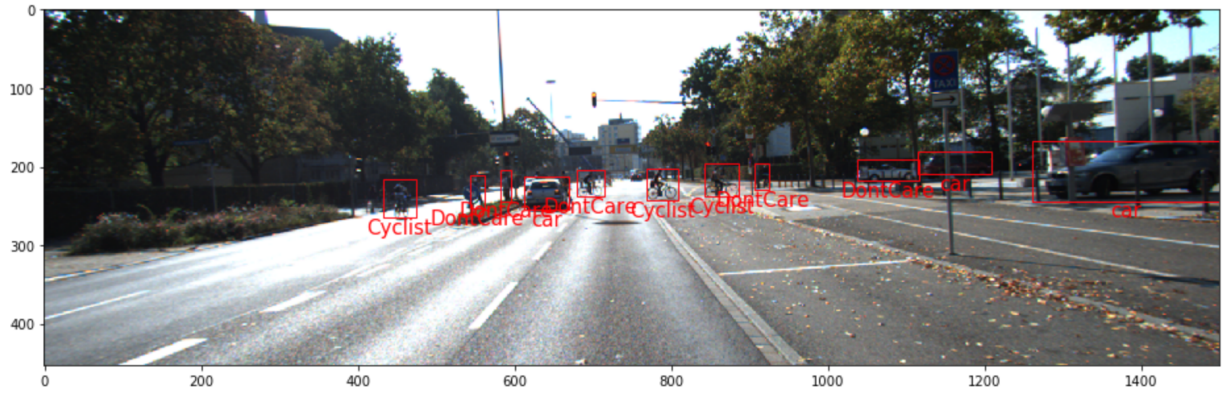

Figure 4: image 006707 with ground truth labels from the KITTI set.

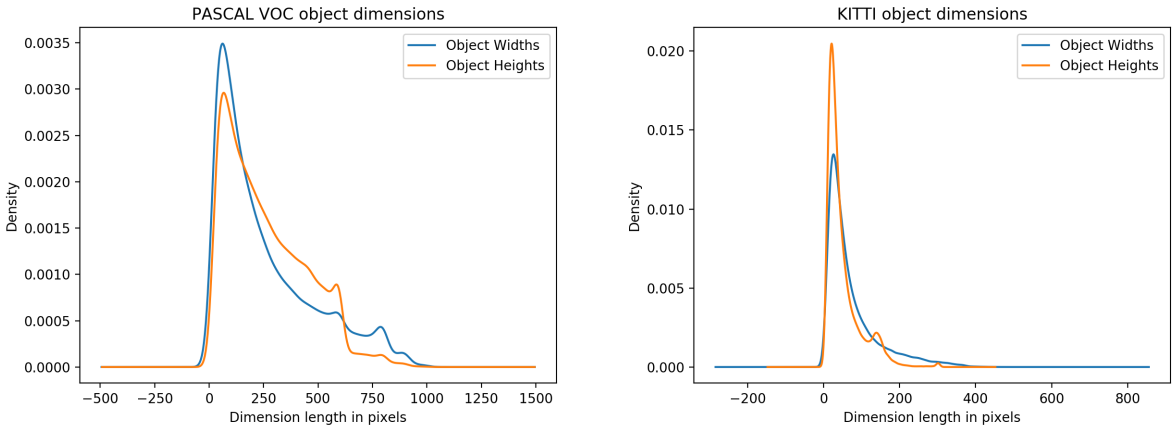

Figure 5: ground truth object width/height distribution in the PASCAL VOC 2007/2012 sets (left) and the KITTI set (right).

2.3 Algorithms and Techniques

Faster R-CNN [3] is a refinement of Fast R-CNN [2], which is itself a refinement of R-CNN [1]. To understand the architecture of Faster R-CNN, we must first consider the history of its predecessors.

2.3.1 R-CNN

The first R-CNN [1] consisted of three modules:

1. a region proposal module using non-deep-learning-based computer vision techniques to propose ~2000 boxes in the image that might contain an object.
2. a CNN that extracts a 4096-dimensional feature vector from each proposed region.
3. a set of SVMs, one per class to classify the region based on the feature vector extracted by the CNN.

Though it achieved state of the art performance at the time, R-CNN was inefficient to run because it forward propagated a 227x227 image through the entire CNN for each of the 2000 regions. For the VOC 2007 images, the entire detection process took 47s per image [2] when using VGG16 [4] as the CNN.

2.3.2 Fast-RCNN

Fast R-CNN speeds up the detection process significantly by replacing modules 2 and 3 of R-CNN to avoid repetitive computations [2]. Instead of making a forward pass through the entire network for each proposed region, Fast R-CNN first makes a single forward pass with the whole image through the convolutional and max pooling layers of the CNN. The resulting convolutional feature map is reused for all regions. For each region, the portion of the convolutional feature map corresponding to that region is fed into a region of interest (ROI) pooling layer which max-pools the convolutional features into a fixed size 7x7 grid, e.g. a 28x14 convolutional feature map would have the features in the 4x2 area at the top left corner max-pooled into the top left feature of the 7x7 grid, and so on. The feature vector output by the ROI pooling layer is then fed into a sequence of fully connected layers which pass their output to two branches:

1. Classification branch: outputs softmax probabilities to classify the region as one of the possible object classes or background (none of the classes).
2. Bounding box regression branch: for each non-background object class, outputs a set of 4 parameters indicating how to transform the region's coordinates into the object's bounding box coordinates, if the region did indeed contain an object of that class.

If there are K object classes, then the classification branch outputs K+1 values, one for each class and an additional for background. The bounding box regression branch outputs 4*K values, t_x, t_y, t_w, t_h defined as follows:

$$t_x = \frac{(x_b - x_r)}{w_r}$$
$$t_y = \frac{(y_b - y_r)}{h_r}$$
$$t_w = \log {}^{w_b}/_{w_r}$$
$$t_h = \log {}^{h_b}/_{h_r}$$

where x_b, y_b, w_b, h_b are the predicted box's center coordinates and its width and height dimensions respectively, and x_r, y_r, w_r, h_r are those of the region proposal. These parameters describe how to translate and resize the proposed region into the bounding box.

When a Fast R-CNN model is constructed based on the VGG16 architecture, the *block5_conv3* layer produces the convolutional feature map fed into the Fast R-CNN module (RoI pooling and the following layers) which replaces the remaining VGG16 layers.

Fast R-CNN is trained using stochastic gradient descent (SGD) on mini-batches of 128 regions, a sample of 64 from each of 2 images. To maintain class balance between objects and background, 25% of the regions in the sample are positives and the rest are negatives. A positive region is one with an intersection over union (IOU) overlap of at least 0.5 with at least one object's ground truth bounding box. A negative region has an IOU of at least 0.1 at least one object, but less than 0.5 with all objects. The training set is augmented to include a horizontally flipped copy of each image. The loss function combines the categorical cross entropy loss from the classification branch with the output of a "smooth" loss function from the bounding box regression branch. The smooth loss function scales quadratically for smaller errors and linearly for large errors so that it is less sensitive to outliers that could otherwise lead to gradient explosion.

Once a VGG16-based Fast-RCNN model is trained, it produces object detections by the following process:

1. Generate region proposals using the same techniques from R-CNN.
2. Feed the image as an input to the base model consisting of the 13 layers in the conv1 through conv5 blocks to output a shared convolutional feature map.
3. For each region, extract the convolutional features corresponding to that region and pass them into the ROI pooling layer of the Fast R-CNN module. The module outputs softmax probabilities for each class and parameters to refine the position and size of the region.
4. Select only the regions classified as non-background and transform each region into a bounding box using the corresponding bounding box transformation parameters. The output object detections consist of a non-background object class, a confidence score for that class, and its bounding box coordinates.
5. Group the detections by class and for each class, apply non-maximum suppression (NMS) to select a subset of the detections for that class. NMS [1] for a given object class sorts the detections of that class by confidence descending, then selects the top N such that no two detections have an IOU overlap greater than 0.7. This technique helps avoid outputting multiple detections of the same object.

2.3.3 Faster R-CNN

Though Fast R-CNN runs significantly faster than R-CNN by computing convolutional features once and sharing the output across all regions, it is still slowed down by the region proposal step, done with techniques such as Selective Search which takes 2s per image. Faster R-CNN [3] resolves this bottleneck by replacing the separate region proposal step with a region proposal network (RPN) sharing the same features used by the Fast R-CNN module. The key observation is that the convolutional feature map can be used for both region proposals and the Fast R-CNN computation for each region.

The RPN begins with a single convolutional layer with filters of dimensions 3x3 sliding over the base network's convolutional feature map with a 1x1 stride. This layer outputs a 512-dimensional

feature for each sliding window position. The output is fed into two branches: one to classify a region as an object or not, and one that produces 4 parameters to transform a region proposal into a bounding box.

At each sliding window position, the network generates multiple region proposals from a pre-determined set of *anchors* centered at that position. An anchor is a box parameterized by its scale and aspect ratio, where the scale is the square root of the desired anchor's area. For example, a scale of 128 and an aspect ratio of 1:1 results in a 128x128 pixel anchor. The Faster R-CNN model trained by Ren *et al*. [3] used scales of 128, 256, and 512 with aspect ratios of 1:1, 1:2, and 2:1, resulting in 9 anchors centered at each position.

Because anchor transformations need to be translation-invariant, both branches are implemented using 1x1 convolutional layers with a 1x1 stride. The classification branch produces one output for each anchor, a binary classification of whether the anchor contains an object. As such, the convolutional layer's output is followed by a sigmoid activation function. The bounding box regression output consists of 4 parameters for each anchor to translate and resize each into a region proposal. The parameters have the same meaning as those learned by the Fast R-CNN module. The RPN module is essentially a simpler version of the Fast R-CNN module with fewer layers and only two classes: object and non-object.

Having defined the RPN, the complete VGG16-based Faster R-CNN object detection process for one image can be described as follows:

1. Feed the image as an input to the base model consisting of the 13 layers in the *conv1* through *conv5* blocks.
2. Feed the *conv5* block output into the RPN module, which slides a convolutional network over the feature map. At each sliding window position, it outputs a confidence score and bounding box regression parameters for each of the 9 anchors centered at that position.
3. Transform each anchor into a region by applying the bounding box transformation according to the predicted parameters for that anchor. Then sort the regions by highest confidence score and apply NMS to select the top N regions not overlapping by more than 0.7 IOU.
4. Pass the selected regions to the Fast R-CNN module, which reuses the shared convolutional feature map. The remaining detection steps proceeds the same way as in Fast R-CNN.

The RPN module is trained using SGD on a small sample of anchors from a single image at a time. The vast majority of anchors are expected to be negatives, so using all of them in the loss function leads to class imbalance. The training process avoids this problem by randomly selecting 256 anchors for a mini-batch such that up to 128 are positives and the remaining are negatives. An anchor is considered a positive if it has an IOU overlap above 0.7 with the ground truth bounding box of any object in the image, or if it is the anchor with the highest IOU for some ground truth box. Non-positive anchors with an IOU overlap less than 0.3 for all ground truth boxes are labeled as negatives. Anchors that fit neither of these criteria are considered neutral and do not contribute to the loss.

The loss function itself has two components: (1) the binary cross entropy loss of the classifier output and (2) the smooth loss function from Fast R-CNN [2] applied to the bounding box regression output of the selected positive anchors only. The two terms are then scaled to contribute roughly equally to the combined loss. The classifier loss is divided by the number of sampled anchors. The regression loss is divided by the number of anchor locations, then multiplying by an arbitrary constant λ to increase match the magnitude of the classifier loss.

The complete Faster R-CNN model is trained using a 4-step process:

1. Train the RPN only by initializing the base network with Imagenet-pretrained weights, then fine-tuning the RPN layers and a selected portion of the base network layers.
2. Train a separate Fast R-CNN model that doesn't share the RPN's base model. The base model is again initialized with Imagenet-pretrained weights. Use the RPN from step 1 to output region proposals to fine-tune the Fast R-CNN layers and a selected portion of the base network layers.
3. Create another RPN with base model weights initialized to those of the model trained in step 2. Freeze those weights and train only the RPN-specific layers. The goal of this step is to retrain the RPN using base model weights known to work for the Fast R-CNN module.
4. Attach another Fast-RCNN module to the base model of the network trained in step 3. Use the RPN module to output region proposals to fine-tune the Fast R-CNN layers only, keeping the base network weights fixed to those learned in step 2.

2.3.4 VGG16

Thus far, Fast R-CNN and Faster R-CNN have been described in terms of an implementation using VGG16 as the base network architecture. Their effectiveness is due to the quality of the convolutional features VGG16 is able to learn. The key innovation of VGG style networks [4] was that using small 3x3 filters in convolutional layers allows for networks with greater depth without an explosion in the number of trainable parameters. VGG16 significantly outperformed older network architectures in image recognition, showing that thinner (smaller filters in each layer), deeper networks learn better features than wider, shallower ones.

The downside of deeper networks is that they are harder to train with today's optimization algorithms [5]. Experiments have shown that arbitrarily adding more layers to a network past a certain depth increases training error, despite the deeper network is a superset of the shallower network. The deeper network could in theory replicate the shallower network by learning the same weights for the shallow network's layers and learning to use each extra layer as an identity mapping. The deepest VGG style network that could be trained effectively was VGG19, with 19 layers.

2.3.5 Residual Networks

Residual networks [5] address deep networks' trouble learning identity mappings by adding shortcut connections between the output of one layer and the input of another layer not directly connected to it. The input to the higher layer becomes the sum of the outputs of the layer directly below it and another layer multiple layers below that. Given some blocks of layers *a*, *b*, and *c*

stacked in that order, suppose a shallower network only layers from block *a* and earlier learns features useful for block *c*. Then the deep network without shortcut connections would have to learn to use *b* as close to an identity mapping in order to preserve block *a*'s features. Adding a shortcut connection between *a* and *c* exposes *c* to *a*'s output features directly so that the network no longer has to learn weights that turn *b* into an identity mapping. Instead, it can learn to use *b*'s features only if they add additional information not provided by *a*'s features. Using the sum of the outputs from *a* and *b* as the input to *c* also means that if it's useful for the deeper network to simply replicate the shallower network ending at *a*, it can learn to weigh *b*'s output as 0, which is easier than having to learn a set of weights that let *b* represent the identity function $f(x) = x$.

To reduce the number of training parameters and thus decrease training time, residual networks also introduce a *bottleneck* architecture for layer blocks. Where a VGG network uses consecutive identical convolutional layers with 3x3 filters in a block [4], residual networks use a 1x1 layer followed by a 3x3 layer followed by another 1x1 layer. Shortcuts only connect 1x1 layers of the same dimensions so that they can be used as identity mappings instead of requiring the network to learn weights to map an output to an input of different dimensions. This allows shortcut connections to be used without additional training parameters.

Finally, residual networks use batch normalization, which speeds up the training of deep networks by normalizing the distribution of individual layers' inputs [10]. Batch normalization had not been discovered at the time of VGG16's development. The combination of these techniques allow for effective training of residual networks hundreds of layers deep, achieving superior image recognition performance to VGG networks on Imagenet.

Given the superior performance of residual networks in image recognition, they must be learning better, higher level features in the convolutional layers. Implementing Faster R-CNN based on residual networks should result in improved region proposal and object detection performance using these better features. In this project I implement Faster R-CNN on top of the 50-layer and 101-layer variants of residual networks, called ResNet-50 and ResNet-101 respectively, and measure the improvement in object detection performance over a VGG16-based model.

2.4 Benchmark

As Faster R-CNN was originally implemented on top of VGG16, I will use the mAP achieved by my own implementation on the PASCAL VOC 2007 test set as a benchmark for performance. I expect the ResNet-50 model to perform better than VGG16, and ResNet-101 to perform better than ResNet-50.

3 METHODOLOGY

3.1 Data Preprocessing

Preprocessing steps are identical to those described in the Fast R-CNN [2] and Faster R-CNN papers [3]:

- For PASCAL VOC datasets, each image is resized to a minimum of 600 pixels on the shorter side. Then, if the longer side ends up at more than 1000 pixels, the image is shrunk such that the longer side is 1000 pixels.
- Training sets are augmented with an additional, horizontally flipped copy of each image in the original set. Training images are shuffled at the start of training and after each pass through the full set so half of the training iterations use a flipped image.
- To be compatible with Imagenet-pretrained weights for base models, images' RGB values are preprocessed by subtracting mean values from Imagenet [1].

When using KITTI images, the resize parameters are 600 and 1500 pixels instead due to KITTI images having much wider aspect ratios and containing smaller objects.

3.2 Implementation

In this project I implement Faster R-CNN in Keras using Tensorflow as the backend engine. Keras provides VGG16 and ResNet-50 implementations in its standard library, but I implement them separately to allow for customization. Implementation details follow those described in [2] and [3], exception when described otherwise. The model is trained using the 4-step alternating procedure. Weights for layers shared by the base models are initialized to Imagenet-pretrained weights for image recognition. All other layers' weights are initialized from a zero-mean truncated Gaussian distribution with standard deviation 0.01, which only generates values within two standard deviations of the mean. To keep training simple and enable caching of training inputs, I train the Fast R-CNN module on mini-batches of 64 ROIs from one image at a time instead of using 128 ROIs from two images. Fast R-CNN was originally trained in [2] using SGD for 30k iterations with a learning rate of 0.001, followed by 10k iterations with a learning rate of 0.0001. I account for the smaller mini-batches by doubling the number of iterations in each step so that each of the 4 training steps consist of 60k iterations at a learning rate of 0.001 and 20k iterations at a learning rate of 0.0001. SGD momentum is fixed to 0.9.

Because Imagenet-pretrained models learn low level features useful for both image recognition and object detection [3], I freeze the weights in lower level layers to reduce training time. In the VGG16-based model, the conv1 and conv2 blocks are frozen. ResNet-50 and ResNet-101 are much deeper models so the conv3 block is also frozen for both.

The VGG16 model adds the RPN and Fast R-CNN layers on top of the *conv5* block, at which point the cumulative stride from all of the max pooling layers is 16 pixels of the original image. For ResNet-50 and ResNet-101, the cumulative stride at the end of *conv5* is 32 pixels, so the Faster R-CNN modules are added on top of the *conv4* block instead, where the cumulative stride is still 16 pixels [5]. The RPN uses the *conv4* output as its feature map directly. For the Fast R-CNN module, the ROI pooling layer is placed on top of *conv4*, but the fully connected layers normally following the ROI pooling output in VGG16 are replaced with the residual networks' *conv5* block. In essence, the ROI pooling layer is inserted between *conv4* and *conv5*. This architecture ends up with the RPN operating on sliding windows of stride 16 as desired, but the Fast R-CNN module has a cumulative stride of 32 pixels. I remedy this by changing the stride of the first convolutional layer in the *conv5* block to 1x1. Then the cumulative stride at the end of *conv5* becomes 16 pixels, matching the VGG16 implementation.

Though only the first 3 blocks are frozen during steps 1 and 2 of training, the batch normalization layers in all 5 blocks are frozen at all times. Due to the mini-batch strategy for Faster R-CNN training, batch normalization layers do not update their statistics correctly so they need to keep their Imagenet-trained values. Consequentially, the ResNet-50 and ResNet-101 implementations of Faster R-CNN can only be trained through transfer learning. Only the VGG16 implementation can be trained from scratch as it lacks batch normalization layers.

Ren *et al*. [3] train Faster R-CNN with a parameter decay of 0.0005. I interpret this as global L2 regularization with a regularization factor of 0.0005. Keras does not support global regularization so I add weight and bias regularizers to each individual layer. ResNet-50 and ResNet-101 are trained with 0.0001 as the regularization factor, following [5].

Girshick [2] mentions an additional step normalizing the bounding box regression parameters to have zero mean and unit variance. This does not appear to be done in the authors' source code. Instead, they used hardcoded values, multiplying the ground truth values of t_x and t_y by 10 and t_w and t_h by 5 when training. During inference, the output parameters are divided by the same values before being used. I take the hardcoded approach as well.

When training the Fast R-CNN module, only the top 12000 ROIs by RPN score are used, following the implementation used by Chen and Gupta [9]. The ROIs are filtered by NMS which selects the region with the highest score remaining, filters out other regions with an IOU overlap of more than 0.7 with that region, and repeats until up to 2000 regions are selected. During inference, NMS only includes up to 300 regions to save CPU time. Class-wise NMS for de-duplicating multiple detections of the same object uses the same 0.7 threshold.

All training and evaluation is done on a p2.xlarge EC2 instance with an Nvidia Tesla K80 GPU.

3.3 Refinement

All of the models trained with the above implementation details performed well enough on the PASCAL VOC 2007 test set, but failed to achieve a mAP higher than 0.2 on the KITTI validation set. To understand the discrepancy between the performance of the same model on PASCAL VOC and KITTI data, consider one image from the KITTI set [7].

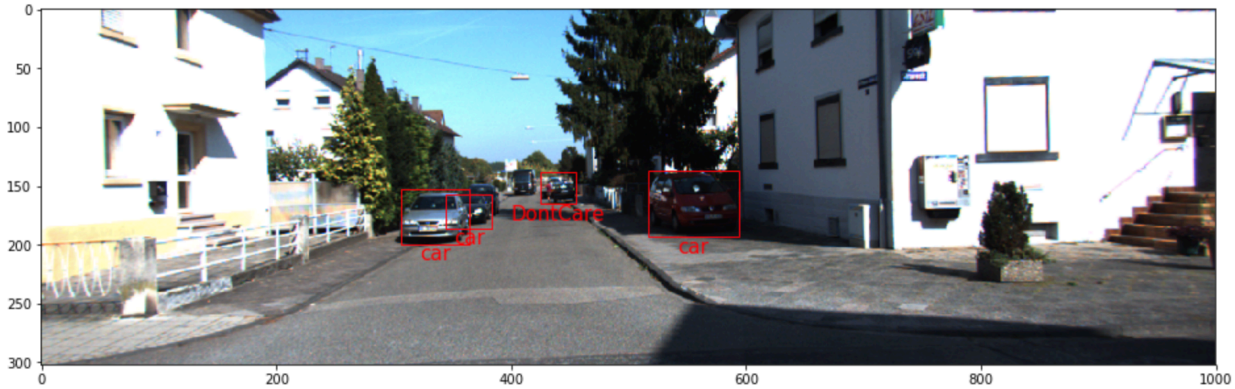

Figure 6: image 004988 from the KITTI validation set annotated with the ground truth objects.

Figure 6 shows the first image in the validation set, resized to a soft minimum dimension of 600 pixels and a hard maximum dimension of 1000 pixels. The original image is 1242x375, longer than the maximum allowed dimension so the resized version is 1000x302. In the resized image, the smallest two objects, the *DontCare* in the background and *car* in the background, have dimensions of 30x26 and 39x29 pixels respectively. When training a ResNet-50 model on the same image repeatedly in an attempt to overfit, it failed to detect these two objects despite learning the exact pixel locations of the two larger cars and outputting miniscule losses when training.

Why did the model predict the exact locations of the larger objects but not notice the smaller objects at all? With a cumulative stride of 16 pixels, the 1000x302 image produces a 63x19 convolutional feature map at the end of the *conv4* block. Using 9 anchors at each sliding window position, the model learns parameters for *63 x 19 x 9 = 10773* distinct anchors. Inspecting the internals of the Fast R-CNN module's training step showed that *none* of the 10773 regions suggested by the RPN were a good match for either of the two smaller objects' ground truth bounding boxes. The Fast R-CNN training step considers a ROI to be positive if it has an IOU overlap of at least 0.5 with at least one object's ground truth bounding box [2]. After filtering the 10773 ROIs down to 2000 with NMS, none of the remaining ROIs fit the positive criteria for either object, so the Fast R-CNN module only trained on ROIs mapping to the larger cars. This explains why the model showed a small loss function output despite missing half of the objects entirely.

The reasons for the RPN's inability to find matching ROIs for these objects are threefold:

1. The cumulative stride of 16 pixels at the end of the *conv4* block is a significant fraction of the objects' size. The convolutional feature map at that point has lost so much spatial resolution that features may not contain enough information to recognize such small objects.
2. Another consequence of the 16-pixel stride is that anchors are only centered at positions 16 pixels apart horizontally and vertically. For an extremely small object, it is possible that the best anchor position to recognize it lies between these fixed 16 pixel intervals. This effect is negligible for objects large enough to contain many anchor centers.
3. The network architecture developed for the PASCAL VOC data uses anchors at scales of 128, 256, and 512 pixels [3]. Even the smallest anchor has dimensions of 128x128, so in the best-case scenario where the anchor encompasses the 30x26 object completely, the object only occupies 4.8% of the anchor's area. With such a large proportion of irrelevant features in the convolutional filter's receptive field, the RPN has trouble distinguishing signal from noise to map this anchor to the much smaller object.

Resizing KITTI images as described reduces them to 80% of their original size, meaning the 16-pixel stride effectively becomes a 20-pixel stride at the original size. These resize parameters are effective when training on PASCAL VOC images as they are generally 500 pixels wide and 375 pixels tall. Such an image becomes 1.6 times larger after resizing to a minimum of 600 pixels per side, so the 16-pixel stride is effectively a 10-pixel stride on the original image.

PASCAL VOC training avoids the object size problem as well because the annotated objects in those images are generally much larger than those in the KITTI images. Tables 1 and 2 show the

dimensions of ground truth objects from both datasets at various percentiles when images are resized with the same parameters.

PASCAL VOC

Percentiles											
	0	10	20	30	40	50	60	70	80	90	100
Width	3	40	66	96	132	179	238	323	445	602	998
Height	5	53	84	123	168	219	278	350	435	539	998

Table 1: sizes in pixels of ground truth objects in the PASCAL VOC 2007 and 2012 "trainval" sets after resizing images.

KITTI

Percentiles											
	0	10	20	30	40	50	60	70	80	90	100
Width	1	17	23	31	38	48	61	80	110	172	571
Height	2	14	19	23	28	34	43	55	76	122	302

Table 2: sizes in pixels of ground truth objects in the KITTI "trainval" set after resizing images.

When KITTI images are resized with the same parameters as PASCAL VOC images, the difference is night and day: the median KITTI object has dimensions of 48x34 pixels, compared to 179x219 for the median PASCAL VOC object. These numbers make it clear why the Faster R-CNN authors chose scales of 128, 256, and 512 for PASCAL VOC training.

Having identified the root cause of the model's poor performance, I propose the following parameter adjustments:

1. Because KITTI objects are so much smaller than PASCAL VOC objects, the anchor scales are amended to be 16, 32, 64, 128, 256, and 512 pixels. The model now generates 18 anchors per sliding window position. With the additional smaller anchors, the RPN is far more capable of mapping anchors to ground truth bounding boxes of small objects.
2. The effective convolutional feature map stride in the resized KITTI images is 20 pixels compared to 10 pixels for PASCAL VOC. Getting a 10-pixel stride for KITTI images requires upsampling the original images by a factor of 1.6, the same as for PASCAL VOC. This can be done by using resize parameters of 600 and 2000, bringing the original ~1250x375 images to a size of 2000x600. Using images of this size increases training and inference time significantly so I compromise for a maximum dimension of 1500. These settings still result in an effective stride of 13 pixels at the end of the *conv4* block.

With these adjustments, the model detects all 4 objects when overfitting on the image from Figure 6. When trained on the entire training set, it also performs much better on the validation set as discussed in the next section.

4 RESULTS

4.1 Model Evaluation and Validation

The models used for PASCAL VOC evaluation are implemented exactly as described in section 3.2. All models are trained on the combined 2007/2012 *trainval* set and evaluated on the 2007 *test* set. Models used for KITTI evaluation are implemented using the refinements from section 3.3. KITTI models are trained on the *train* set and evaluated on the *val* set.

4.1.1 PASCAL VOC Evaluation

Table 3 shows the results of various combinations of networks and training parameters.

Base Network	Number of ROIs	Running time per image (s)	mAP
VGG16	300	0.41	0.5970
VGG16	128	0.32	0.5877
VGG16	64	0.3	0.5710
ResNet-50	300	0.53	0.6513
ResNet-50	128	0.3	0.6487
ResNet-50	64	0.22	0.6228
ResNet-101	300	0.64	0.6565
ResNet-101	128	0.39	0.6551
ResNet-101	64	0.31	0.6508

Table 3: results for various models evaluated on the PASCAL VOC 2007 test set.

The results validate the hypothesis that more successful networks for image recognition are also more successful in object detection when used as a base architecture for Faster R-CNN. The model's runtime during inference can be reduced substantially by using only 128 ROIs with minimal loss in mAP compared to using the default 300 ROIs.

Next I evaluate the impact of global regularization. Table 4 shows the performance of the ResNet-50 and ResNet-101 models with and without regularization. Using a global weight regularization of 0.0001 does not seem to improve results consistently.

Base Network	Regularization Factor	Number of ROIs	Running time per image (s)	mAP
ResNet-50	0	300	0.53	0.6565
ResNet-50	0	128	0.3	0.6526
ResNet-50	0	64	0.22	0.6164
ResNet-50	0.0001	300	0.53	0.6513
ResNet-50	0.0001	128	0.3	0.6487
ResNet-50	0.0001	64	0.22	0.6228
ResNet-101	0	300	0.64	0.6557
ResNet-101	0	128	0.39	0.6529
ResNet-101	0	64	0.31	0.6463
ResNet-101	0.0001	300	0.64	0.6565
ResNet-101	0.0001	128	0.39	0.6551
ResNet-101	0.0001	64	0.31	0.6508

Table 4: results for models trained with and without regularization evaluated on the PASCAL VOC 2007 test set.

Then I evaluate the impact of changing the optimizer. Table 5 shows the performance of the ResNet-50 model with the default SGD optimizer, as well as with an Adam optimizer where all

steps are trained at 1/10th the learning rate as with SGD. Although Adam-trained networks show lower losses throughout training, the improvement does not extend to test-time evaluation.

Base Network	Optimizer	Number of ROIs	Running time per image (s)	mAP
ResNet-50	Adam	300	0.53	0.5798
ResNet-50	Adam	128	0.3	0.5732
ResNet-50	Adam	64	0.22	0.5149
ResNet-50	SGD	300	0.53	0.6513
ResNet-50	SGD	128	0.3	0.6487
ResNet-50	SGD	64	0.22	0.6228

Table 5: results for Adam and SGD trained models evaluated on the PASCAL VOC 2007 test set.

4.1.2 KITTI Evaluation

Table 7 shows the results for KITTI evaluation. There are four notable differences from the PASCAL VOC results:

1. ResNet-101 outperforms ResNet-50. This may be because the KITTI images contain more difficult objects to detect, as discussed in section 3.3.
2. Models trained with regularization outperform those trained without. The KITTI *train* set is much smaller than the combined PASCAL VOC 2007/2012 *trainval* sets, so overfitting is more likely without regularization.
3. Using fewer ROIs at test time results in a steeper decline in mAP compared to with PASCAL VOC images. Using 128 ROIs is still feasible but using 64 ROIs incurs a large drop in mAP. This is likely because there are more objects per image in the KITTI set as shown in section 2.1, so more ROIs are needed to capture all of them.
4. Evaluation takes longer per image due to the increased number of anchors per sliding window location and larger image size.

Base Network	Regularization Factor	Number of ROIs	Running time per image (s)	mAP
ResNet-50	0	300	0.63	0.5872
ResNet-50	0	128	0.38	0.5698
ResNet-50	0	64	0.28	0.5314
ResNet-50	0.0001	300	0.63	0.6088
ResNet-50	0.0001	128	0.38	0.5818
ResNet-50	0.0001	64	0.3	0.5383
ResNet-101	0	300	0.7	0.6138
ResNet-101	0.0001	300	0.7	0.6363
ResNet-101	0.0001	128	0.45	0.5829
ResNet-101	0.0001	64	0.37	0.5727

Table 6: results for models on the KITTI val set.

4.2 Justification

In this project, I successfully adapted the Faster R-CNN architecture to achieve acceptable performance on the KITTI dataset for autonomous vehicles. Though the running time per image is longer than the 0.2s per image measured by Ren *et al*. [3], much of the additional time is due to the slow GPU used. Chu [15] measures the performance of multiple GPUs, showing that the p2.xlarge instance's GPU is 4x slower than the Nvidia GTX 1080 Ti. Assuming the performance boost from the more powerful GPU translates to this model as well, the ResNet-101 model trained with regularization using 300 ROIs at test-time would be able to produce predictions at the rate of ~6 frames per second, which may be accepted for use in an autonomous vehicle. To understand this model's performance more deeply, it is useful examine its ability to detect individual object classes. Table 7 shows the AP of the model per class.

Base Network	Regularization Factor	Number of ROIs	Object Class	AP
ResNet-101	0.0001	300	car	0.8007
ResNet-101	0.0001	300	person	0.5779
ResNet-101	0.0001	300	Cyclist	0.6733
ResNet-101	0.0001	300	DontCare	0.1645
ResNet-101	0.0001	300	Misc	0.4910
ResNet-101	0.0001	300	Person_sitting	0.4705
ResNet-101	0.0001	300	Tram	0.7569
ResNet-101	0.0001	300	Truck	0.8636
ResNet-101	0.0001	300	Van	0.7843
ResNet-101	0.0001	300	Overall	0.6363

Table 7: AP for each class when using the best-performing model.

The results show that the model does a better job of detecting larger objects. While an autonomous vehicle equipped with this model may avoid vehicle collisions successfully, it could need a separate model to detect pedestrians for extra safety.

5 CONCLUSION

5.1 Free-Form Visualization

Figures 7, 8, and 9 show examples of the regularized ResNet-101 model's outputs on KITTI *val* images.

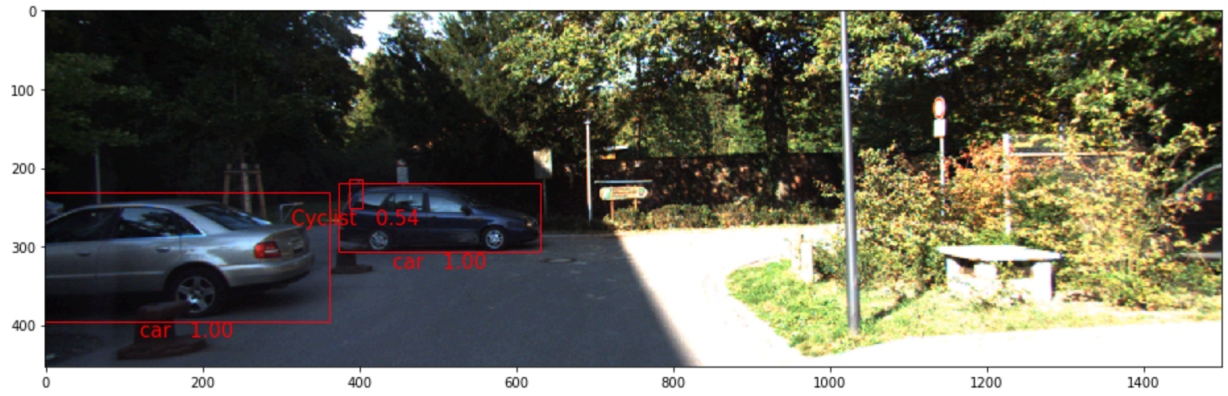

Figure 7: output for image 006600 of the KITTI validation set.

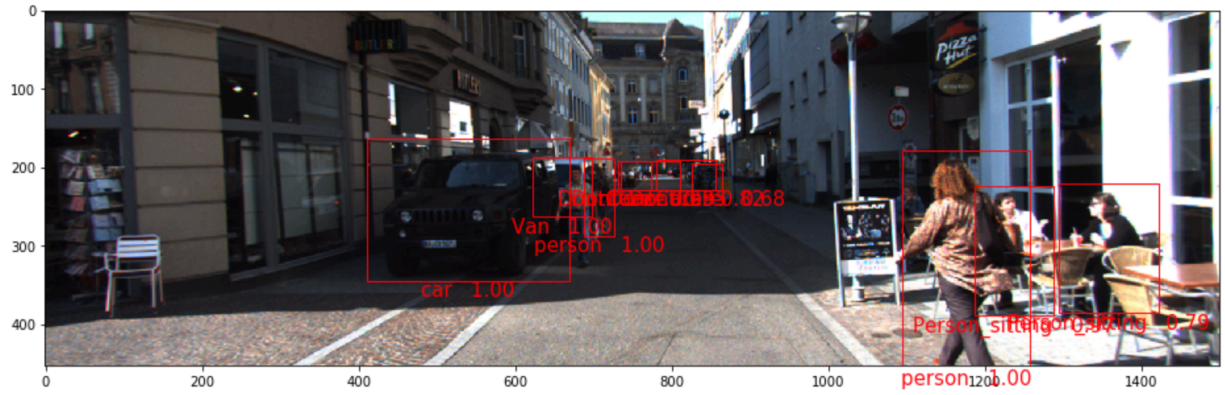

Figure 8: output for image 006800 of the KITTI validation set.

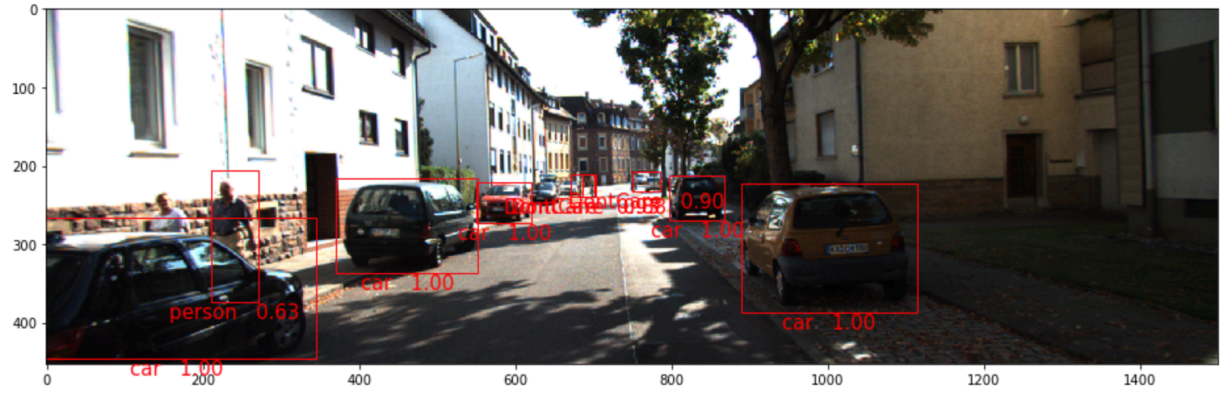

Figure 9: output for image 007000 from the KITTI validation set.

Figures 7, 8, and 9 show that the model is very effective at detecting vehicles, but has some trouble with pedestrians. The most likely reason is that pedestrians are very thin, so we again run into the problem of the convolutional feature map lacking the resolution needed for downstream modules to distinguish pedestrians.

5.2 Reflection

The entire process of solving the problem went through these steps:

1. Implement the model architecture following the details of the Faster R-CNN paper as closely as possible. With this done I can evaluate performance with a VGG16-based model on PASCAL VOC 2007 and use the result as a benchmark.
2. Write the software needed to use the trained model as an object detector to evaluate its performance on the test set.
3. Extend the model to support using ResNet-50 as the base architecture.
4. Rewrite most of the codebase to optimize performance during the non-GPU portions of training.
5. Extend the model to support using ResNet-101 as the base architecture.
6. Add an option to use Adam as the optimizer.
7. Add an option to train on both the 2007 and 2012 datasets for PASCAL VOC.
8. Add the ability to train the model on KITTI datasets.
9. After investigating the model's poor performance on the KITTI validation set, add options to customize the image resize parameters and anchor scales.
10. Create a script to annotate frames from a video to make an easy to understand demo.

Step 1 was extremely difficult, as implementing the full architecture required a lot of code. Keras does not provide an easy way of implementing the custom loss function behavior described in the Faster R-CNN and Fast R-CNN papers, nor does it provide a way to use a single image as the input for 64 different ROIs and propagate the loss for each ROI, so I needed to find hacks around those problems.

Step 2 presented a problem as well because I do not have Matlab, in which the official PASCAL VOC evaluation code is written. Attempting to run the same evaluation scripts in Octave did not work. The definition of mAP is not explained in detail anywhere so I had to copy most of the mAP-calculating code from the official *py-faster-rcnn* repository.

Step 3 had me stumped for several weeks, as I was not familiar with batch normalization. Trying to make the model overfit on one training image eventually revealed that the RPN worked well but the Fast R-CNN module produced wildly inaccurate results even though the training loss decreased. I investigated the issue by comparing the training loss at each iteration with the evaluation loss on the same image, which should be similar in magnitude. Instead, the training loss steadily decreased in each iteration while the evaluation loss was orders of magnitude higher. On the other hand, the ResNet-50 RPN worked well when using a VGG16 based Fast R-CNN. It was also worked well with a hybrid model, where a VGG16-based Fast R-CNN module is placed of a ResNet-50 base network. These observations indicated the problem was somewhere in the Fast R-CNN layers of the ResNet-50 model. After I took out the batch normalization layers, the evaluation loss ended up matching the training loss in magnitude. Reading the batch normalization paper made it clear why my implementation would not collect batch statistics correctly during training so the Fast R-CNN module ended up outputting garbage values. Freezing all layers' batch normalization parameters and statistics to Imagenet-pretrained values fixed the issue.

Step 4 was one of the most fun parts of this project. The training process was extremely slow due to poorly optimized code for generating the ground truth tensors for training both the RPN and the Fast R-CNN module. My original implementation used a lot of the abstraction techniques I often use when writing Java at my day job, but this style of programming is inefficient in Python. I wrote a profiling decorator to measure the performance of individual functions and found that filling in the ground truth values for all anchors in an image took 0.4s on its own, whereas the GPU-optimized backpropagation step for RPN training ran in just 0.2s. I found similar issues in the ground truth value calculator for Fast R-CNN training. I optimized away the slowness of the CPU-run parts of RPN training by throwing away my abstractions in favor of vectorized numpy operations, experimenting with tricks to make numpy perform calculations in place instead of allocating new memory, caching results of expensive calculations to reuse cached results when encountering the same image again, and using smaller data types (32-bit floating point numbers and 16-bit integers instead of numpy's default 64-bit numbers). In the end, most of the CPU-executed operations ran in negligible time compared to the time spent during GPU forward and back propagation. The exception is the NMS function during Fast R-CNN training. NMS appears to be an $O(n^2)$ algorithm in the worst case where n is the number of ROIs used. The dramatic increase in the number of ROIs after adding extra anchor scales is the main reason why the KITTI model trains and inferences so more slowly than an equivalent PASCAL VOC model. I tried several ways to make NMS run faster, but the only trick that worked was casting its inputs to 16-bit integers, which shaved off 25% of its execution time.

During all the refactoring needed for performance improvements, maintaining the code's correctness was a challenge because it was infeasible to retrain a model and check its performance every time I renamed a variable or similar. For example, I occasionally wasted several days training a model only to find out that I had heights and widths switched. I solved this problem by writing very crude "integration" tests that set random seeds to fixed values, then run one training iteration on an image and check whether the weights of an arbitrary model layer match the same layer from a file containing reference weights. These tests do not confirm correctness, only whether a network does exactly what it did before. Reference weights were obtained using a model known to work from manual testing. Reference weights had to be re-generated every time I fixed a bug, such as during the batch normalization investigation. Though crude, the tests let me confidently refactor code by catching errors such as mixing up dimensions, confusing the coordinates of an anchor's top left corner with the coordinates of its center, confusing a box's dimensions with its coordinates, and various typos. It was thanks to these tests that I could experiment with various ways of removing performance bottlenecks.

Step 9 was challenging as well, but by then I had learned how to debug neural network issues from dealing with the batch normalization problem so a day-long deep dive into training internals was all it took to uncover the problem with small object sizes. During the investigation, I wrote a script to show statistics of ground truth object sizes in a dataset (see Tables 1 and 2), as well as a Jupyter notebook to visualize ground truth object locations and compare them to model-predicted ones (see Figures 1 and 6).

Apart from those challenges, the remaining steps were straightforward coding. Implementing Faster R-CNN gave me some insights that I could not have learned just from reading papers, such as why the authors chose scales of 128, 256, and 512 only and why research papers on object detection always reported better results on PASCAL VOC than on other benchmarks. The significantly longer run time of the final model after adjusting for small object sizes made it clear why I have not heard of Faster R-CNN being used for autonomous vehicles, despite its performance on the KITTI benchmark.

5.3 Improvement

5.3.1 Improvements in Science

I implemented two adjustments to my network to better detect small objects, but there are other ways I did not explore. As the problem came from a large cumulative stride relative to object sizes, I could have also reduced the stride in the conv4 block to 1 pixel so that the cumulative stride at the end is only 8 pixels. This would increase the resolution of the convolutional feature map and perhaps make it easier to recognize small objects, but it would also increase the training and inference time dramatically. With each sliding window location reduced to half its diameter, the RPN would examine 4 times as many anchors as before, likely making the NMS step prohibitively slow.

The residual networks I used in this project perform better than VGG16, but there are other, more recent architectures I could explore with even better performance. Newer image recognition architectures include Inception-ResNet [11], Wide ResNet [12], and DenseNet [13], among others. The Faster R-CNN architecture can be built on top of newer image recognition networks to benefit from their higher quality convolutional feature maps.

In addition to using newer image recognition architectures, I could also implement some object-detection-specific improvements. He *et al*. describe several Faster R-CNN refinements to increase mAP in [5]. Redmon and Farhadi propose the Yolo9000 architecture for a much faster object detector in [14].

5.3.2 Improvements in Engineering

I could optimize the runtime of network training further by exploring ways to run NMS as compiled C code or running it on the GPU. If I were willing to rewrite much of the codebase, I could use a multithreaded training algorithm where the GPU trains the model on one image while the CPU simultaneously runs the NMS step of the next image. The code would be far more complicated, as I would have to first forward propagate the RPN for the next image on the GPU, then start training the Fast R-CNN using the current image on the GPU while simultaneously running NMS for the next image on the CPU. This optimization would minimize the time the GPU spends idle waiting for CPU operations to complete, but the training code would be difficult to maintain and diverge significantly from the inference code.

At one point, I had much faster training time during step 4 of training because I reused cached data from intermediate steps when encountering the same image twice, which allowed me to skip running the RPN and the NMS step. This only worked when I trained on small datasets. Once

I switched to the larger *trainval* sets and added horizontal flipping, my EC2 instance ran out of memory to store all the images' convolutional feature maps and TensorFlow crashed after around 4000 images. The caching optimization still works in step 2 of training because the model input in that step is the original image, not the convolutional feature map, so I could lazily load the image from disk. As such, step 2 ends up training faster than step 4 despite performing an extra forward pass of the image through the base model's layers. If I stored the previously cached convolutional feature maps on disk instead to save memory, I might be able to speed up step 4 training again.

Finally, I could improve training and possibly inference time significantly by training the RPN and Fast-RCNN simultaneously instead of using the 4-step alternating procedure. To do so would require operations that Keras lacks, such as training the RPN on an image and then using the output to pick regions for the Fast R-CNN part of the network. I might be able to do this using lower level TensorFlow primitives to implement the current Python-based operations as a network layer. As before, this would require rewriting most of the codebase.

REFERENCES

[1] R. Girshick, J. Donahue, T. Darrell, and J. Malik. "Rich feature hierarchies for accurate object detection and semantic segmentation", in CVPR, 2014.

[2] R. Girshick. "Fast R-CNN", in ICCV, 2015.

[3] S. Ren, K. He, R. Girshick, and J. Sun. "Faster R-CNN: Towards real-time object detection with region proposal networks", in NIPS, 2015.

[4] K. Simonyan and A. Zisserman. "Very deep convolutional networks for large-scale image recognition", in CVPR, 2014.

[5] K. He, X. Zhang, S. Ren, and J. Sun. "Deep residual learning for image recognition", in CVPR, 2015.

[6] M. Everingham, L. Pool, C. Williams, J. Winn, and A. Zisserman. "The PASCAL visual object classes (VOC) challenge", in IJCV, 2010.

[7] A. Geiger, P. Lenz, C. Stiller, and R. Urtasun. "Vision meets robotics: the KITTI dataset", in IJRR, 2013.

[8] J. Huang, V. Rathod, C. Sun, M. Zhu, A. Korattikara, A. Fathi, I. Fischer, Z. Wojna, Y. Song, S. Guadarrama, K. Murphy. "Speed/accuracy trade-offs for modern convolutional object detectors", in CVPR, 2017.

[9] X. Chen and A. Gupta. “An Implementation of Faster RCNN with Study for Region Sampling”. arXiv:1702.02138 (2017).

[10] S. Ioffe and C. Szegedy. “Batch normalization: Accelerating deep network training by reducing internal covariate shift”, in ICML, 2015.

[11] C. Szegedy, S. Ioffe, V. Vanhoucke. “Inception-v4, inception-resnet and the impact of residual connections on learning”. arXiv:1602.07261 (2016).

[12] S. Zagoruyko and N. Komodakis. “Wide residual networks”. arXiv:1605.07146 (2016).

[13] G. Huang, Z. Liu, and K. Q. Weinberger. “Densely connected convolutional networks”. arXiv:1608.06993 (2016).

[14] J. Redmon and A. Farhadi. “Yolo9000: Better, faster, stronger”. arXiv:1612.08242 (2016).

[15] V. Chu. 2017 Apr 20. “Benchmarking Tensorflow Performance and Cost Across Different GPU Options”. https://medium.com/initialized-capital/benchmarking-tensorflow-performance-and-cost-across-different-gpu-options-69bd85fe5d58