Object Detection of Cars in a Parking Lot with Deep Learning

By: Samuel Ordonia

Advisor: Professor Chris Pollett

Abstract

Deep learning has reemerged onto the scene of research because of advances in computational power and enormous data sets. Specifically, in the field of computer vision, there have been several advances following the renewed research efforts into deep learning. Since the introduction of the Convolutional Neural Net (CNN) in the late 1990s, several iterations and improvements on the CNN have been made in computer vision. However, these improvements have mostly been in iconic images--those with one or two objects in them. This master's project focuses on classifying and locating multiple objects--namely, cars--using a custom CNN based on the darknet "You Only Look Once" (YOLO) fully convolutional net.

Background

Machine learning involves the consumption of data to "train" a model to make predictions on future inputs. One of the most prominent applications for machine learning is the training and analysis of images. Some research questions that arise as a result include, "Can a machine learn to track movement within a set of images like a human can or better?" and "Can a machine identify objects in an image as accurately as a human can?" The former question is referred to as image localization, while the latter question is also known as image classification. Together, they form the field of image detection.

Deep learning is a branch of machine learning that has existed for several decades alongside other "classical" approaches, such as image segmentation and multi-linear SVMs. However, deep learning was not explored as extensively until recently, when modern computing power and much larger datasets have become available. The field known as deep learning started as a means of simulating how the human brain learns, and so the term artificial neural net (ANN), a superset of DNNs, was coined [3]. In [3], Goodfellow et al review the history of ANNs across the decades since the 1940s. Indeed, the basic fundamentals of ANNs have been established since the 1950s.

The unit of a DNN is the neuron, and as the name implies, it simulates what occurs in the human brain's neuron. The sensory input of the human neuron is mirrored with a data input into the computer neuron. And in the human brain, certain neurons in the brain are fired depending on the input, which in turn results in learning or

decision-making. Likewise, in an artificial neuron, there is an activation function that "fires" depending on the input, model weights, and hyperparameters.

An early approach of computer vision involving neural nets is found in [4]. The application is the detection of automobiles crossing toll points. By modern standards, the architecture is quite archaic and simplistic, but the results show effective detection of automobiles within the designated interest regions. The grayscale image is fed into a set of fully-connected neuron layers, and a tanh activation function filters the output. The model only signals if an object has passed through the observation region and does not identify what the object is.

A noteworthy breakthrough in deep learning architecture as well as computer vision occurred when LeCun et al developed the convolutional neural net (CNN) for image classification [4]. His premise was to minimize the preprocessing of the input image and hand over the feature detections entirely to the model to find and learn from. And so the CNN was developed to take the image input itself and output what the image is. Essentially, it breaks up an image into smaller subsections each with its own feature maps spanning multiple planes. Each set of units within the feature map is taken as a subsample of the input image, and the states of each stored via a transformation known in mathematics as convolution. The process is followed by pooling the results for analysis in subsequent layers of the neural net. LeCun et al shows in their paper the effectiveness of computer recognition of handwritten numbers with their neural net architecture.

The LeCun et al CNN boasts the freedom of error if the image is translated, since the architecture is configured in such a way as to determine the regions of interest itself [3]. It also shows promise in distinguishing different objects from each other within the same image. Finally, the CNN breaks up the image into feature maps in parallel and performs the same convolution operation within the same layer. This means that the image can be fed into the model at once, instead of being broken up into regions and scanned slowly. Unfortunately, there remain multiple limitations regarding accuracy that more recent research has attempted to address.

Some modern CNNs, such as R-CNN, return to a region-based approach, which trades runtime for a boost in accuracy. Unfortunately, in many applications of computer vision, detection time needs to be quick, especially when cars are involved. But with the "you look only once" (YOLO) model, object detection can be done once without doing region analysis [5]. The YOLO model is able to classify over nine thousand different objects. It takes advantage of the fact that an image is fed only once into a CNN to perform

localization of significantly distinct objects within an image and form bounding boxes around them. And because it is trained as a classifier as well, it is able to label what is inside the bounding box with relatively higher accuracy compared to R-CNN and with lower compute time. This approach YOLO takes to object detection is also known as “single-shot detection” (SSD).

Project Deliverables

Pytorch Binary Classifier

The first deliverable focuses on familiarization of the deep learning framework Pytorch by building a binary digit classifier that would differentiate 0-4 and 5-9 digits from the MNIST dataset. Accuracy is approximately 97%.

Constructing a CNN with additional functionalities such as activation functions and backpropagation from scratch is not feasible, and there are several deep learning frameworks from which to choose. Tensorflow, Caffe, Keras, Theano, and Pytorch are but a handful.

In recent years, Tensorflow has become the most popular deep learning framework. Developed by Google, it claims to make deep learning model construction straightforward yet powerful. At first glance, it does offer the same functionality as a lower-level fraemwork such as Caffe offers, but with a fraction of the code required to express it by the developer. However, when experimenting with it, there are too many levels of abstraction that make any modifications to existing modules or functions extremely difficult.

Further, Tensorflow is inherently declarative; it compiles the entire neural net and training/testing binary altogether, then executes it all at once. This makes it difficult to perform any logging outside of their semi-flexible logging tool Tensorboard. Debugging tools cannot be used either.

Another popular framework is Keras. It is an API to frameworks such as Tensorflow or Theano. Although Keras is excellent at basic prototyping, it is not as flexible as a lower-level framework when modifying certain parameters of the DNN.

Pytorch is relatively new to the scene of deep learning frameworks. Its 1.0 version was released this year 2018. Because it is newer, it has the benefit of hindsight against other frameworks and takes a different approach than frameworks such as Tensorflow.

Pytorch is chosen for its inherently imperative approach to development, making its dynamic paradigm conducive to research. Like normal Python, it is compiled at runtime, so printing, logging, or debugging can happen at runtime, anytime. This makes it easy to iterate the Pytorch deep learning model.

Syntactically, Pytorch provides a nice balance of flexibility and brevity in its syntax. The deliverable itself used a simplified version of the AlexNet CNN--the SamNet. SamNet comprises a few convolutional and pooling layers, followed by a dense output layer to classify. Expression of these layers is terse and straightforward.

```
class SamNet(nn.Module):
    def __init__(self):
        super(SamNet, self).__init__()
        # 1 input image, 10 output channels, 5x5 square convolution kernel
        self.conv1 = nn.Conv2d(1, 10, kernel_size=5)
        # 10 input channels, 10 output channels, 5x5 square convolution kernel
        self.conv2 = nn.Conv2d(10, 10, kernel_size=5)
        self.conv2_drop = nn.Dropout2d()
        # Only need 2 neurons for output
        self.fc1 = nn.Linear(160, 2)
```

Further, Pytorch has straightforward integration with the GPU for computation and training times that are orders of magnitude faster than on a CPU. The GPU has thousands of cores, and a well-written deep learning framework will take advantage of parallelization to maximize efficiency and linear speedup. Tensors are defined on the GPU. And forward feeds along with back-propagation also happen on the GPU.

Using the CUDA API with Pytorch, runtime to train two epochs and test the SamNet requires less than thirty seconds. To use the GPU, assign it to the “device” attribute during setup.

```
use_cuda = torch.cuda.is_available()
device = torch.device("cuda" if use_cuda else "cpu")

train_loader, test_loader = load_data(args, use_cuda)

torch.manual_seed(args.seed)
model = SamNet().to(device)
```

Image Processing and Deep Learning Environment

The second deliverable is to build a comprehensive but lightweight research environment to conduct further experiments. Pytorch has been chosen as the deep learning framework.

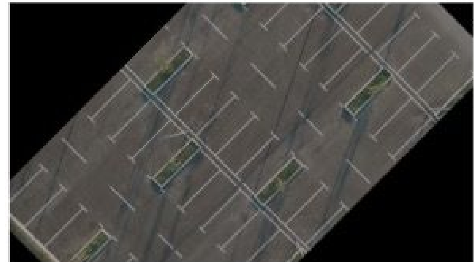
Open CV is chosen as the tool for image manipulation, transformation, or processing. In the field of computer vision, it is the most popular framework and has APIs for different languages, from C++ to Python. It is also known to be faster in Python than other image libraries or frameworks such as Pillow, since Open CV has been heavily optimized in the C code.

Some examples of helpful Open CV features are image rotation to generate more training data and conversion of videos into frame-by-frame image files. The video to images conversion is included in this deliverable and is tested on the big bunny video.

It also has other powerful processing tools such as affine transformations, resizing to construct image pyramids, and pattern matching. These functions are included in this deliverable and are exemplified in the following.

Examples

(Left) Original.
(Bottom-left) Affine
(Right) Concatenated
(Bottom-right) 45 degree rotation



The Open CV functions and code to perform these transformations are concise and self-explanatory.

```python
def affine_transform(img):
    rows, cols, ch = img.shape
    pts1 = np.float32([[50, 50], [200, 50], [50, 200]])
    pts2 = np.float32([[10, 100], [200, 50], [100, 250]])

    M = cv2.getAffineTransform(pts1, pts2)
    return cv2.warpAffine(img, M, (cols, rows))


def concatenate(img1, img2=None, axis=0):
    """
    axis = 0 will stack the images vertically.
    axis = 1 will stack the images horizontally.
    """
    if not img2:
        img2 = img1
    return np.concatenate((img1, img2), axis=axis)


def resize(img, factor):
    """
    Specify the image size manually with fx & fy.
    This allows the use of floats for scaling.
    """
    height, width = img.shape[:2]
    return cv2.resize(img, None, fx=factor, fy=factor)


def rotate(img, degree):
    rows, cols = img.shape[:2]
    rotation_matrix = cv2.getRotationMatrix2D((cols/2, rows/2), degree, 1)
    return cv2.warpAffine(img, rotation_matrix, (cols, rows))
```

However, Open CV wreaks havoc on the host machine when installed onto it. The install paths for entire packages are overridden, and not all the functionality works well, since some of Open CV's dependencies became out of sync with the host's pre-installed versions of those same dependencies. Fortunately, the Open CV binaries all resided in one place on the host machine, so the host machine was recovered quickly once Open CV was removed. As a result of this mishap, Open CV is contained within a Docker container to protect the host machine's environment.

Docker takes advantage of the same Linux kernel across containers and the host's other executions, so the container running Open CV appears to be another executable to the host. In contrast, a virtual machine runs a completely separate "guest" operating system on the host and proves too compartmentalized and costly to maintain for isolating Open CV. As a result, Docker proves to be the correct balance of containerization and does not require a hypervisor and separate linux kernel like a virtual machine would.

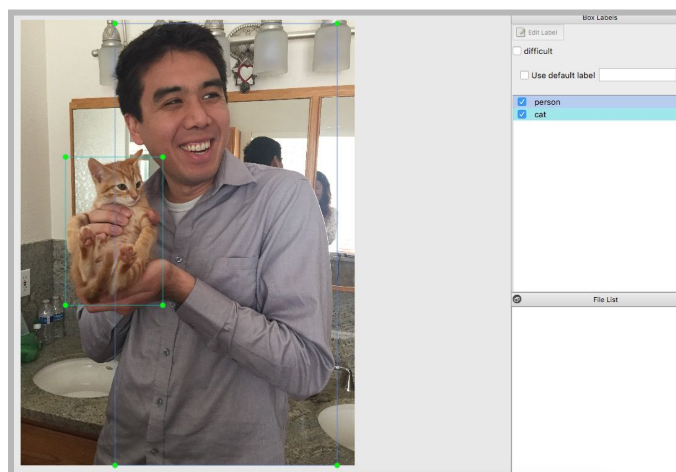
Linux containers have existed long before Docker, but Docker provides a powerful, concise API to define these containers. There are also prebuilt containers on the remote registry with basic tools and services such as Python, Redis, or Postgres already built into them.

Custom Data Annotation for Images

The third deliverable focuses on the data aspect of deep learning. A machine learning model is only as good as the data on which it trains. This deliverable is to find an image annotation tool and annotate a hundred images of cars in a parking lot. Once a dataset is formed, it can be used to train, validate, and test the deep learning model.

The image annotation tool needs to track which classes correspond to which bounding box. It also needs to output the bounding box locations into a consistent, sensical format. When annotating images, the directory and file structure need to be consistent as well; otherwise, the labels may become out of sync with the raw images. The images themselves cannot be annotated directly, since their raw formats will be using for training and validation. These annotations are considered the “ground truth” of these images.

The image annotation tool is straightforward to use and requires only a few steps to build onto the local machine as an executable application.



It outputs the locations and classes of each object in an image into a separate text file or set of markups, depending on the settings chosen for exporting the ground truth.

Kitten.xml

```
<annotation>
	<folder>My Pictures</folder>
	<filename>Kitten.jpg</filename>
	<path>/Users/Sam0/Documents/My Pictures/Kitten.jpg</path>
	<source>
		<database>Unknown</database>
	</source>
	<size>
		<width>2448</width>
		<height>3264</height>
		<depth>3</depth>
	</size>
	<segmented>0</segmented>
	<object>
		<name>person</name>
		<pose>Unspecified</pose>
		<truncated>1</truncated>
		<difficult>0</difficult>
		<bndbox>
			<xmin>691</xmin>
			<ymin>27</ymin>
			<xmax>2319</xmax>
			<ymax>3264</ymax>
		</bndbox>
	</object>
	<object>
		<name>cat</name>
		<pose>Unspecified</pose>
		<truncated>0</truncated>
		<difficult>0</difficult>
		<bndbox>
			<xmin>328</xmin>
			<ymin>1005</ymin>
			<xmax>1041</xmax>
			<ymax>2091</ymax>
		</bndbox>
	</object>
</annotation>
```


However, annotating images manually quickly became extremely time consuming, especially for object-rich images such as parking lots. The more objects to annotate per image also increases the risk of errors and retries. At an annotation rate of ten minutes per image, the workload to annotate a hundred of these images would be almost seventeen hours.

The VOC and COCO datasets are two popular collections of annotated images spanning over ten thousand images of non-iconic shots. Annotation of ten thousand parking lot images would require nearly seventy days of working twenty-four hours a day. There is also the difficulty of finding all these parking lot images as well. Because of the rich feature density per image, it is difficult to replicate training examples computationally, so these images would have to be taken in the real world.

The deliverable had to extend in scope to include the search for parking lot datasets already annotated by teams or entire workforces.

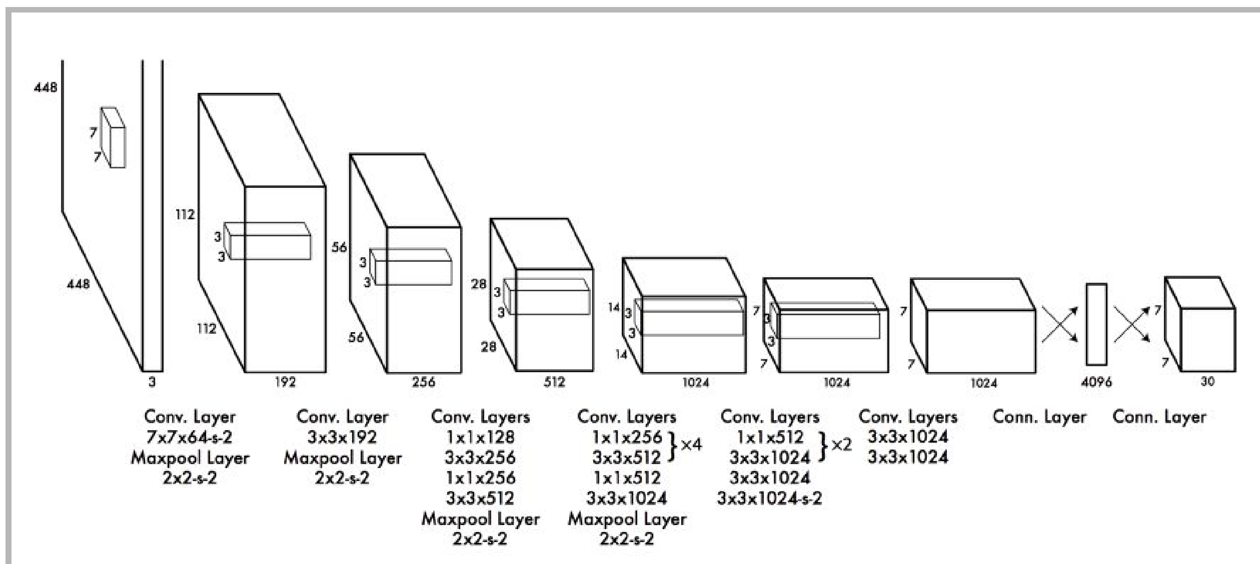
So, separate data sets were obtained from other researchers who have already done the arduous work of annotating the parking lots. In total, there are several thousand annotated parking lot images. There is additional work needed to compile their

annotation formats into one the model can consume during training, validation, and testing. But the hundreds or thousands of hours required to annotate these images have been spared for research efforts instead of manual annotation.

YOLO Detection

The fourth and most challenging deliverable is to find or build a working implementation of the YOLO CNN and benchmark its performance at object detection. When surveying different contemporary approaches to computer vision deep learning, it became clear that the winning approach in the long run would need to prioritize speed. The YOLO deep learning model has proven to retain accuracy compared to other approaches while also boasting fast detection speeds.

It is currently on version three, which has the general shape of the following. It is a “fully convolutional” network (FCN). The final output layer is depth of twenty indicating it is trained on twenty classes.



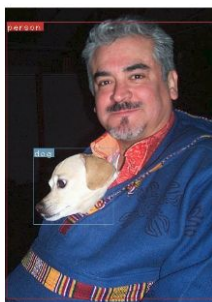
The original author of YOLO, PJ Reddie, open-sourced his work. Unfortunately, the program proved faulty. Most notably, it did not run on the GPU, most likely because of the lack of continued support for the original open-sourced work to newer version of CUDA. As a result, a custom Pytorch implementation is built to benchmark the detection accuracy.

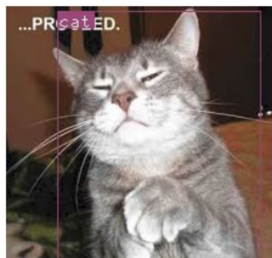
The open sourced version of YOLO is written in C. However, in recent years, deep learning communities have exchanged architectures using text files. These files contain

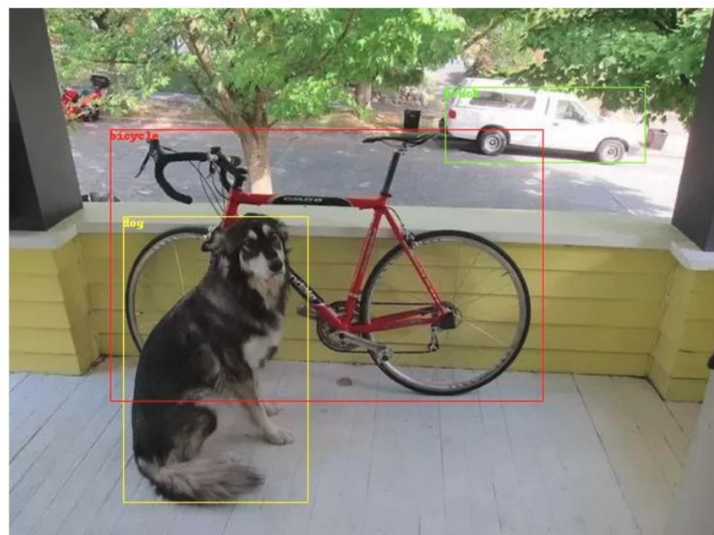
the information to create every layer. They also contain hyperparameters used across the DNN. Sharing neural nets through config files makes the network framework-agnostic; Pytorch, Tensorflow, or any other framework could parse it and run the network.

The Pytorch implementation of YOLO leverages all the functionalities of the framework. It parses the config file, then programmatically generates the neural net. Because the Pytorch is much clearer than the C implementation, it is much easier to prototype and iterate. Also, Pytorch's ease of CUDA integration makes it relatively straightforward to instantiate the tensors and run the neural net on the GPU. Pytorch also supports loading the weights of the latest version of YOLO, version three.

Detection proves accurate on iconic images. These are images with the main object of interest front and center, and there tend not more than two or three other objects in the frame. Even with objects in the background, it still has the ability to detect them.







The YOLO model shows difficulty detecting several objects in the background. Interestingly, it detects a bounding box for the entire parking lot and correctly labels its contents.

YOLO Training

This deliverable involves training on the latest version of the YOLO CNN architecture. It runs on a Pytorch implementation of the model and interfaces with the GPU via the CUDA framework API. It builds off the progress of the previous deliverable, Deliverable 4, and it also adds insight into how to format the annotated images from Deliverable 3.

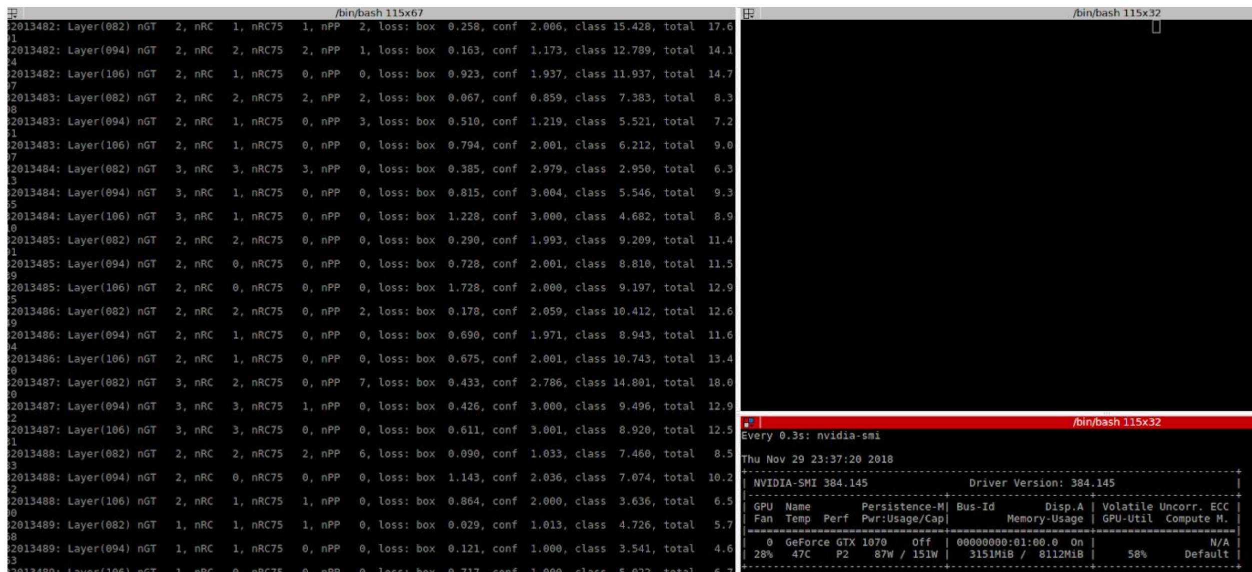
A model is only as good as the data it is trained on. And because the model architecture will update during the research development process, it will need to run through training data whenever it changes. Therefore, it is crucial to the continued progress of this research topic to set up training for this model.

The neural net expression for YOLO is in the same application. Training successfully occurs on the GPU because of the relative ease of using CUDA with Pytorch. In fact, training on the GPU needs to happen. Otherwise, there will not be much progress towards improving the model given time constraints and the necessity to see results sooner rather than later during the research development process. In this Deliverable as well as the previous ones involving deep learning, an NVIDIA 1070 and CUDA 9 are used.

However, much more additional work is completed to compile training data into a format the neural net can consume. The data sets used during training are the VOC ones from 2007 and 2012. It contains a mixture of iconic and non-iconic images. The non-iconic images do not contain nearly the same density of objects as a parking lot full of cars, however. There are thousands of annotated images to compile and run on YOLO. Fortunately, the creation of Pytorch tensors can also happen on the GPU with ease.

The VOC website provides scripts to compile their ground truth annotations in Python. And with the custom integration into the input for the YOLO CNN, future data set directories can be modeled after VOC's. That way, the code for inputting training data can be reused.

A separate Pytorch training module is created for training and validation. The epochs of training are run on top of the existing YOLO v3 weights. The following shows the training occurring on the GPU. Because of Pytorch's imperative paradigm, it is straightforward to print out the progress during execution.



Although training is quite performant on the GPU, it will still require hours of uptime to train across several epochs of the VOC data sets. During that time, it is crucial to maintain steady control of the GPU temperature, because it rises forty degrees Celsius above room temperature during training. VRAM and power usage both fluctuate between approximately forty to fifty percent.

The Pytorch training uses stochastic gradient descent (SGD) with learning rate and momentum inspired by Redmon's work. The loss function is also inspired by his research and represents a concise snapshot of all the main factors behind classification and bounding box regression.

Next Steps

The YOLO implementation is complete. It detects objects in images well, unless there are many small ones in the background. The model also runs on the GPU, making it feasible to iterate exponentially faster than if training, validation, and testing occurred on the CPU.

There are already several thousand images collected of cars in parking lots collected in Deliverable 3. The natural progression is to incorporate these images into the training and validation. Each dataset will need some conversion script to fit the input format YOLO expects.

The hypothesis is that additional training examples on objects in the background or from different viewpoints and pose will improve accuracy. However, the hypothesis further extends to predict that the additional training data will not improve accuracy that much. Therefore, once the model trains across several epochs of the data, further improvements will need to be made to the CNN.

YOLO has a particular architecture that it follows. It boasts faster runtimes than other contemporary neural net architectures because it performs classification and bounding box regression in the same pass. YOLO does not crop images into regions and run each one through the CNN like R-CNN does.

It is important to note the non-negotiables of YOLO before determining next steps with the CNN itself. The "single-shot detection" needs to run the entire image once through the CNN. It also should not do a regions-based approach and split the model between a classifier and regressor.

Certain modifications, such as image segmentation, do not make much sense and simply collapse into a different, already explored architecture. However, some preprocessing techniques can be used, such as rotation, resizing, or affine transformations; however, the intention here is more to generate additional training data instead of augmenting the detection itself. It is possible that resizing to make the

parking lot images larger will help with detection, but there is a certain threshold that is hit when resolution becomes too poor upon over-sizing.

YOLO’s loss function is one of the most compelling innovations of the CNN. It captures all in one calculation the anchor of the image, the bounding box size, a confidence level for the object, and the classification score for the object. It is worth exploring how to improve its effectiveness further.

Additional research will include the addition and modification of the convolution and pooling layers to improve detection of small features. Even removing layers may indicate what direction to take next regarding accuracy improvements for cars in the background.

Another much more ambitious modifcation will be switching out the entire feature extractor or regressor blocks of the CNN for a different one, perhaps ResNet or VGGNet. However, such changes will be more involving and may also require a rework for how the data is inputted.

In conclusion, there is no clear path in terms of updating the CNN. Once all the additional parking lot data is incorporated into the training, validation, and testing workflow, the most challenging and also innovative aspects of the project will involve improvements to accuracy with changes to the CNN.

Literature References

[1] W. Lingxia, J. Dalin, “A method of parking space detection based on image segmentation and LBP,” Multimedia Information Networking and Security Conference, 2012. [Online]. Available: http://ieeexplore.ieee.org/document/6405668.

[2] W. Zhu et al, “Tracking of object with SVM regression,” Conf. Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, 2001. [Online]. Available: https://www.cse.sc.edu/~songwang/document/cvpr01.pdf.

[3] I. Goodfellow et al, Deep Learning, MIT Press, 2016, pp. 3. [Online]. Available: http://www.deeplearningbook.org.

[4] Y. LeCun et al, “Object recognition with gradient-based learning,” Proc. IEEE, 1998. [Online]. Available: http://yann.lecun.com/exdb/publis/index.html#lecun-98.

[5] J. Redmon, A. Farhadi, “YOLO9000: better, faster, strong,” in Conf. on Computer Vision and Pattern Recognition (CVPR), Honolulu, HI, USA, 2017. [Online]. Available: https://arxiv.org/pdf/1612.08242.pdf.

[6] Y. Xiang et al, “Subcategory-aware convolutional neural networks for object proposals and detection,” in IEEE Winter Conf. on Applications of Computer Vision (WACV), Santa Rosa, CA, USA, 2017. [Online]. Available: https://yuxng.github.io/xiang_wacv17.pdf.