Department of Computing

Imperial College of Science, Technology and Medicine

H-PERL: The Hybrid Property, Event and Relation Learner

Author:
Ross Irwin

Supervisors:
Prof. Alessandra Russo
Dr. Krysia Broda
Dr. Jorge Lobo

June 11, 2020

Chapter 1

Background

This chapter introduces some technical background relevent to this project. It includes an introduction to neural networks and CNNs, a comparison of some existing object detection algorithms and a discussion on Knowledge Representation and Reasoning and symbolic rule learning.

1.1 Deep Learning

Deep neural networks (DNNs) have emerged as a very successful algorithm for machine learning; deep learning has been used to beat records in tasks such as image recognition, speech recognition and language translation [12]. Many different architectures have been proposed to solve various tasks, these architectures include convolutional neural networks (CNNs), which are designed to process data that come in the form of multiple arrays [12], and recurrent neural networks (RNNs), which are designed to process sequences of arbitrary length [13]. The following section gives a brief introduction to CNNs and describes some of their use cases.

1.1.1 Convolutional Neural Networks

CNNs contain three types of layers: convolution, pooling and fully connected. Units (artificial neurons) in a convolution layer are organised into feature maps. The inputs to each unit in a feature map come from the outputs of the units in a small region of the previous layer, the output of the unit is then calculated by passing the weighted sum of its inputs through an activation function such as ReLU. The set of weights, also known as a filter or kernel, is the part of the layer which is learned through backpropagation. The value of each unit in a feature map is calculated using the same kernel. Each feature map in a layer has its own kernel. Pooling layers reduce the size of the input by merging multiple units into one. A typical pooling operation is max-pooling, which computes the maximum of a local patch of units. Finally, in fully-connected layers (which are typically placed at the output of the CNN) every unit in a layer is connected to every unit in the previous layer. An example CNN architecture is shown in Figure 1.1.

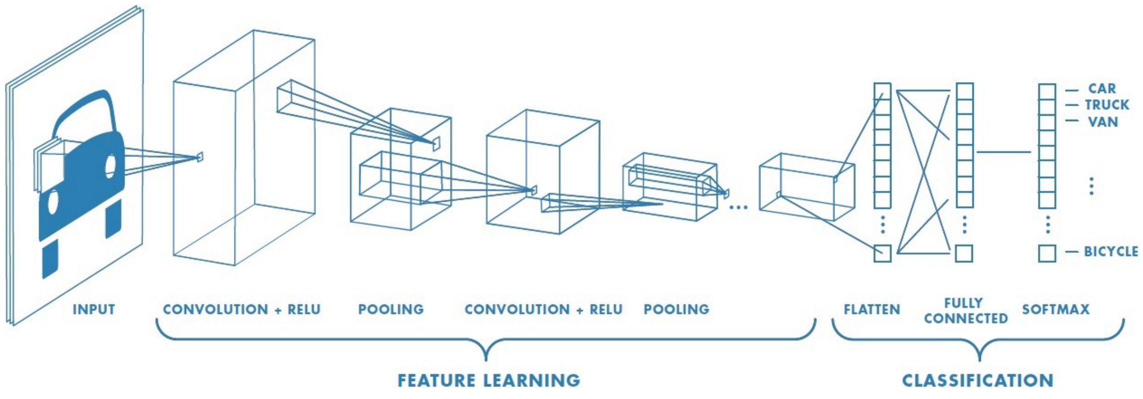


Figure 1.1: An example of a CNN architecture. The input image is passed through a series of convolution and pooling layers before being flattened into a one-dimensional layer and passed through one final fully connected layer. The softmax classification function is then applied at the output.

1.2 Image Processing

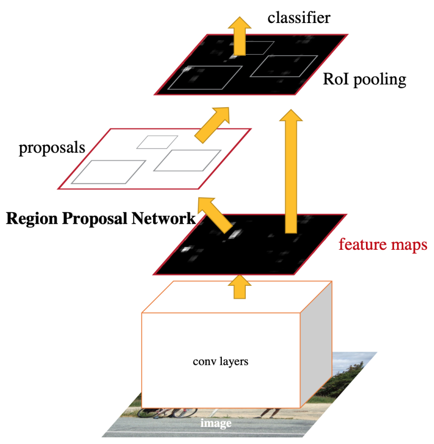
Convolutional networks have proven to be adept at a number of tasks involving images, including image classification [6] and object detection [15, 17]. This section explores the object detection task, as well as metrics for both tasks, further. Object detection forms an important component of the Hybrid VideoQA approach which we outline in Chapter **??**.

1.2.1 Object Detection

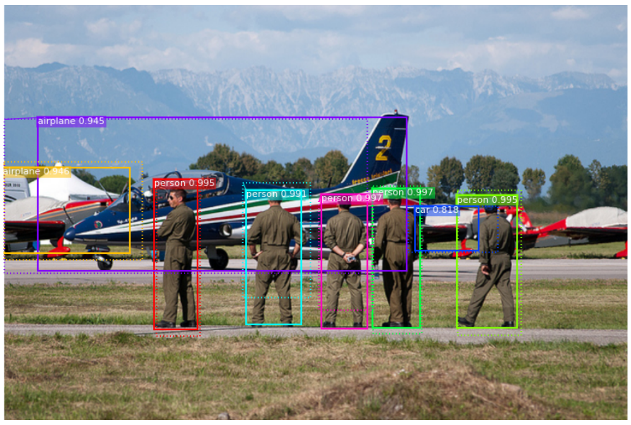
The object detection task could formally be defined as designing a model which, when given an image, can produce a rough localisation of objects of interest in the image (in the form of a bounding box) and classify each of these objects into a set of predefined classes. In this section we introduce two well known object detection algorithms, *Faster R-CNN* [17] and *You Only Look Once* (YOLO) [15].

Faster R-CNN is an evolution of previous object detection algorithms, R-CNN [5] and Fast R-CNN [4]. Faster R-CNN builds on its predecessors by adding a region proposal network (RPN) - a neural network which takes an image and produces a set of region of interest (RoI) proposals. This method of region proposal is much faster than previous algorithms (such as those used in [5] and [4]) since it is able to make use of the GPU, as opposed to requiring the CPU. Faster R-CNN then uses a similar classifier and bounding box regressor as Fast R-CNN at the output; this section of the network also receives the feature maps from the final layer of the RPN, in this sense the initial layers of the network are shared between the region proposal section and the classifier/regressor section. A diagram of the Faster R-CNN architecture is shown in Figure 1.2a.

The three object detection algorithms mentioned above all work by first producing region proposals, then producing a more accurate localisation and a class score



(a) Diagram of the Faster R-CNN architecture. Figure from [17].



(b) An example of the bounding boxes and confidence scores produced by an object detection algorithm. Image from [1].

Figure 1.2

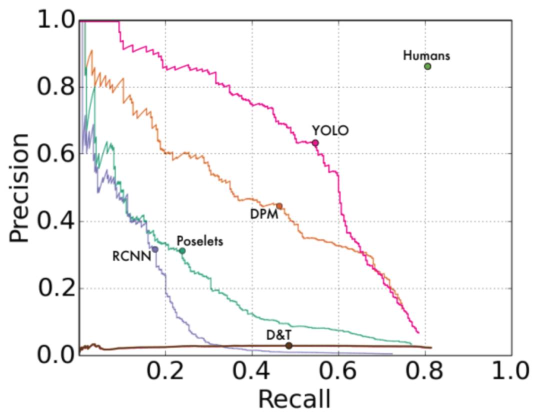
for each region and finally removing any low-scoring or redundant regions. This requires the algorithm to 'look' at the image multiple times (around 2000 times for R-CNN). You Only Look Once (YOLO) is a significantly more efficient algorithm which, as the name suggests, takes a single look at the image. A convolutional neural network is used to simultaneously predict multiple bounding boxes and the class probabilities for each box. As well as being very fast, YOLO makes fewer than half the number of background errors (where the algorithm mistakes background patches for objects) as Fast R-CNN [16]. YOLO is, however, slightly less accurate than some of the slower methods for object detection [15].

1.2.2 Commonly Used Metrics

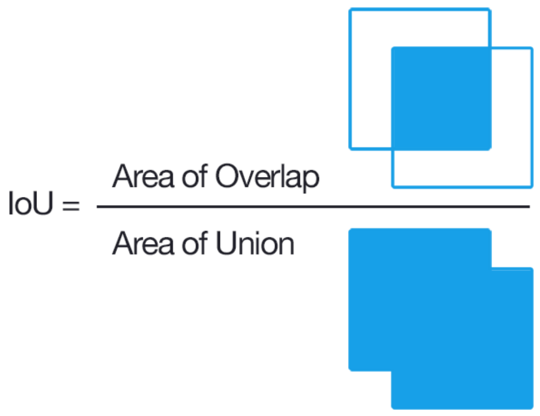
In this section we present some commonly used metrics for classification and object detection tasks. We use TP, TN, FP and FN to mean True Positive, True Negative, False Positive and False Negative, respectively.

Firstly, for classification tasks the following terminology is commonly used:

- $Accuracy = \frac{TP+TN}{TP+TN+FP+FN}$. The accuracy is the ratio of correct predictions to the total number of predictions.
- $Precision = \frac{TP}{TP+FP}$. The precision is the ability of a classifier to not label the negative data as positive.
- $Recall = \frac{TP}{TP+FN}$. The recall is the ability of the classifier to find the positively-labelled data.
- $F_1 = 2 * \frac{precision*recall}{precision+recall}$. The F_1 score is a way of combining the precision and recall scores.



(a) Example precision-recall curves for various object detection models. Image from [16].



(b) Visual explanation of the intersection over union metric. Image from [18].

Figure 1.3: Precision-recall curves and definition of intersection over union

Each object detector model will output a confidence score for each object classification it makes. We can then set a threshold value such that confidence scores below the threshold are not counted as a classification of an object. Altering this threshold value will give different precision and recall values for the model, which can then be plotted on a precision-recall graph. Example precision-recall curves are shown in Figure 1.3a.

For object detection tasks, where a bounding box is produced to estimate an object's position, metrics which measure the accuracy of the detection are required. One very common metric is the Average Precision (AP), which is roughly defined as the area under the precision-recall curve (although estimates of this value are usually used and there a number of slightly different definitions). In order to assess how well a model localises an object in an image, the *Intersection over Union* (IoU) is calculated between the ground-truth bounding box and the box produced by the model. The IoU between object A and object B is defined as the area of intersection between A and B divided by the area of union between A and B. Figure 1.3b gives a visualisation of IoU. Each IoU threshold will produce its own precision-recall curve. We use Average Precision with a number of IoU thresholds to evaluate our trained object detector in Chapter 2.

1.3 Knowledge Representation and Reasoning

Knowledge Representation and Reasoning (KRR) is concerned with how intelligent agents store and manipulate their knowledge. In this section we discuss: Answer Set Programming, a logic programming framework; the action language $\mathcal{AL}$, which can be used to model the behaviour of a dynamic system; and methods for learning logical rules.

1.3.1 Answer Set Programming

Answer Set Programming (ASP) is a form of declarative logic programming that can be used to solve difficult search problems. Whereas imperative programs define an algorithm for finding a solution to a problem, logic programs simply define a problem, it is then the job of logic program solvers to find the solution. ASP also differs from Prolog, also used for logic programming, in that ASP programs are purely declarative. This means that reordering rules, or atoms within rules, has no effect on the output of the solver [2]. ASP solvers work by finding the answer sets of the program, where each rule in the program imposes restrictions on possible answer sets. An answer set can be thought of as a set of ground atoms which satisfies every rule of the program (although the full definition of an answer set is too in-depth for this discussion).

The following templates are some of the possible forms of rules in an ASP program:

$$a \text{ :- } b_1, ..., b_k. \tag{1.1}$$

$$l\{c_1; ...; c_n\}u \text{ :- } b_1, ..., b_k. \tag{1.2}$$

$$\text{:- } b_1, ..., b_k. \tag{1.3}$$

Each b_i and c_i, as well as a, in the above equations is known as a *literal*. Literals are defined as being either a first-order logic atom, a, or its negation, $not\ a$. The not in the rule body stands for negation-as-failure, and means that $not\ b_i$ will be satisfied when b_i cannot be proved to be true in an answer set.

The left hand side of an ASP rule is known as the head, and the right hand side is known as the body. Each rule requires that when the body is satisfied - that is, when every member of the body is in an answer set - the head must be in the answer set.

Rule 1.2 is known as a *choice rule*. The bounds of a choice rule (l and u in Rule 1.2) must be integers. The head of a choice rule can be satisfied by any subset, S, of the atoms inside the brackets, provided $l \leq |S| \leq u$; in effect, a choice rule creates possible answer sets. The bounds can be left off, in which case they take default values of 0 and the size of the choice set, respectively.

Finally, rule 1.3 is known as a *constraint*. The body of a constraint must not be satisfied; intuitively, a constraint rules out answer sets that satisfy its body.

As well as negation as failure, ASP also has a notion of 'strong negation'. The strong negation of an atom p is written $\neg p$. Strong negation can be thought of as classical negation, although it does not always have the same properties. In practice, ASP solvers implement strong negation by treating $\neg p$ as an additional atom, and enforce that no answer set can contain both p and $\neg p$.

$$P = \left\{ \begin{array}{l} p \text{ :- } a. \\ \{a; b\} \text{ :- } f. \\ \text{:- } b. \\ f. \end{array} \right\} \tag{1.4}$$

As an example, the two answer sets of the program, P, outlined in 1.4, are $\{f, a, p\}$ and $\{f\}$. The final rule of the program is known as a *fact*. Facts must be true in all answer sets.

As well as finding the set of answer sets of a logic program, the ASP language can also be used to create a preference ordering between answer sets. The ASP solver is then capable of finding the most preferred answer set. Preferences are assigned to answer sets using weak constaints. The format of a weak constraint in ASP is as follows:

$$:\sim b_1, ..., b_n.[w@p, t_1, ..., t_m] \tag{1.5}$$

In Rule 1.5 each b_i is a literal, each t_i is a term (combinations of function symbols, variables and constants), w is an integer weight and p is an integer priority level. When ASP optimises a weak constraint it firstly assigns each answer set with one cost accumulator for each priority level. When the body of a weak constraint is satisfied by an answer set, the constraint's term tuple $(t_1, ..., t_m)$ is added to a set, and, if the term tuple was not already in the set, the weight is added to the cost accumulator for the given priority. ASP chooses the optimal answer set by minimising the cost at the highest priority level. If multiple answer sets are considered optimal ASP optimises the next highest priority level. It continues to optimise until either only one answer set remains or there are no further priority levels to optimise, at this point the optimal answer sets are returned.

1.3.2 The Action Language $\mathcal{AL}$

Action languages are formal models for describing the behaviour of dynamic systems. In this section we present the version of $\mathcal{AL}$ given in [3]. $\mathcal{AL}$'s signature contains three special sorts: *statics*, *fluents* and *actions*. Fluents are partitioned into two sorts: *inertial* and *defined*. Inertial fluents are subject to the law of inertia, which means their value stays the same unless it is changed (directly or indirectly) by an action. Defined fluents are not subject to the inertia axiom and cannot be caused by an action; instead they must be defined in terms of other fluents. Statics and fluents are both referred to as 'domain properties'. A 'domain literal' is a domain property or its negation.

$\mathcal{AL}$ models a system by a *transition diagram* - a directed graph whose nodes correspond to possible states of the system and whose arcs, which are labelled by actions, correspond to possible transitions between states. The following three types of $\mathcal{AL}$ statements are used to describe the transitions of an $\mathcal{AL}$ model:

- *Causal Laws*, which say that an action a, executed in a state satisfying domain literals $p_1, ..., p_m$, causes fluent f to become true in the next state of the system, take the following form:

$$a \textbf{ causes } l_{in} \textbf{ if } p_1, ..., p_m \tag{1.6}$$

- *State Constraints* say that every state which satisfies domain literals $p_1, ..., p_m$ must also satisfy f, and take the following form:

$$l \textbf{ if } p_1, ..., p_m \tag{1.7}$$

- Finally, *Executability Conditions* make it impossible to execute actions $a_1, ..., a_k$ simultaneously in a state which satisfies the domain literals $p_1, ..., p_m$. These conditions take the following form:

$$\textbf{impossible } a_0, ..., a_k \textbf{ if } p_0, ..., p_m \tag{1.8}$$

A collection of $\mathcal{AL}$ statements is known as a *system decription*. An $\mathcal{AL}$ system description can be used to model the behaviour of dynamic systems with discrete states; each state can be seen as the set of fluents which are true and transitions between states are caused by actions. It is possible to encode a given $\mathcal{AL}$ system description, along with a number of domain independent axioms, in ASP. The method for creating this, along with an example encoding, is given in Appendix **??**. In Chapter **??** we outline an $\mathcal{AL}$ model and use it to find the set of events which are most likely to have occured in

1.3.3 Symbolic Rule Learning

Inductive Logic Programming [14] (ILP) is a field of symbolic AI research concerned with learning symbolic rules which, when combined with background knowledge, entail a set of positive examples and do not entail any negative examples. ILASP [11] (Inductive Learning of Answer Set Programs) is an ILP framework for learning ASP programs.

The authors of [7] define the *Learning from Answer Sets* (ILP_{LAS}) task (which is the task solved by the original version of ILASP), by first defining a *partial interpretation*. A partial interpretation E is a pair of sets of atoms E^{inc} and E^{exc}, known as the *inclusions* and *exclusions* of E. An answer set A *extends* E if it contains all of the inclusions ($E^{inc} \subseteq A$) and none of the exclusions ($E^{exc} \cap A = \emptyset$). An ILP_{LAS} task is then defined as the tuple $T = \langle B, S_M, E^+, E^- \rangle$, where B is the background knowledge, S_M is the search space, E^+ and E^- are the partial interpretations for the positive and negative examples, respectively. ILASP is able to construct the search space from a *language bias* specified by *mode declarations*. Full details, and examples, on how to write a language bias for ILASP can be found in the ILASP manual [11].

Given an ILP_{LAS} task, T, ILASP finds an hypothesis (an ASP program), H, which is known as an inductive solution of T, such that all of the following are true:

1. $H \subseteq S_M$
2. $\forall e^+ \in E^+ \; \exists A \in AS(B \cup H)$ *such that* A *extends* e^+

3. $\forall e^- \in E^- \nexists A \in AS(B \cup H)$ *such that* A *extends* e^-

Where $AS(P)$ *refers to the answer sets of a program* P.

Later versions of ILASP are capable of solving more complex tasks, including learning weak constraints [10] (a method for specifying preferences in ASP), learning from context dependent examples [9] and learning from noisy examples [8].

Chapter 2

Evaluation

This chapter describes the performance details of the components and models outlined in Chapters **??** and **??**. Section 2.1 compares the performance of a selection of counterpart components used in the hardcoded and trained models. Section 2.2 compares the overall performance of the two H-PERL models, and includes details of the models' performance under noisy conditions.

Before presenting the evaluation, we outline the following three key criteria that we want to evaluate the models against:

1. **Accuracy** :- What proportion of questions does the model answer correctly?
2. **Speed** :- How long does the model take to complete the evaluation?
3. **Adaptability** :- How simple is it to transfer the model to a new environment?

While we would have preferred to have compared our models to state-of-the-art neural network implementations for VideoQA, training end-to-end VideoQA networks is very resource intensive and takes a long time. If these models were included in the evaluation, we would add explainability as an additional criterion. However, since our models are both hybrid, their implementations have roughly the same explainability. We do, however, keep in mind that hybrid models are often easier to understand than fully-neural counterparts, as well as being significantly faster to train.

2.1 Components

The first part of the evaluation concerns the individual model components. In this section we compare the performance of the properties, relations and events components from both the hardcoded and trained models. These components are evaluated using perfect input data - where we assume no mistakes have been made in previous parts of the pipeline. We also outline the performance of the object detection component, which is common to both H-PERL models. All of these components are evaluated using the 200 validation videos, rather than the testing videos, although both sets of videos are generated in the same way.

This section does not include a dedicated discussion on the object tracker's performance, since there are no QA pairs with which it can be directly evaluated. However, after using some intuitive heuristics to gauge the component's object tracking abilities, we see that the tracker assigns identifiers in exactly the way we would expect for the OceanQA environment. In the rest of this chapter, therefore, it should be assumed that the tracker operates with perfect accuracy on the OceanQA environment, but we cannot provide an official evaluation for this.

2.1.1 Object Detector

In Chapter 1 we mentioned that *Intersection over Union* (IoU), which is calculated between a ground truth bounding box and the bounding box produced by the detector, measures how well a detector localises an image. As a reminder, the IoU between an object A and object B is defined as the area of intersection between A and B, divided by the area of union between A and B. Chapter 1 also mentioned that a threshold t can be applied to a set of detections so that only the detections with confidence $> t$ are used. Each IoU threshold that is applied to the set of detections produces a different precision-recall curve for the detector. The average precision (AP) is then defined as the area under the (smoothed) precision-recall curve, although estimates of this value are used. Average precision, however, is only defined for detection of a single class. The mean average precision (mAP) metric is therefore used to find the mean AP across k classes, as follows:

$$mAP = \frac{\sum_{i=1}^{k} AP_i}{k} \tag{2.1}$$

In order to conduct the evaluation, we used the *PyCocoTools*[1] library, which implements the metrics used in the COCO object detection challenge. Table **??** presents the average precision values, for a range of IoU thresolds, achieved by our detector on the OceanQA validation dataset.

2.1.2 Properties

2.1.3 Relations

2.1.4 Events

2.2 Models

[1]Available at: https://github.com/cocodataset/cocoapi

Bibliography

[1] W. Abdulla. *Mask R-CNN for object detection and instance segmentation on Keras and TensorFlow*. URL: `https://github.com/matterport/Mask_RCNN` (visited on 01/15/2020).

[2] Thomas Eiter, Giovambattista Ianni, and Thomas Krennwallner. "Answer Set Programming: A Primer". In: *Reasoning Web. Semantic Technologies for Information Systems: 5th International Summer School 2009*. Ed. by Sergio Tessaris et al. Springer Berlin Heidelberg, 2009, pp. 40–110.

[3] Michael Gelfond and Yulia Kahl. *Knowledge representation, reasoning, and the design of intelligent agents: The answer-set programming approach*. Cambridge University Press, 2014.

[4] Ross Girshick. "Fast r-cnn". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1440–1448.

[5] Ross Girshick et al. "Rich feature hierarchies for accurate object detection and semantic segmentation". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2014, pp. 580–587.

[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: *Advances in Neural Information Processing Systems 25*. Ed. by F. Pereira et al. Curran Associates, Inc., 2012, pp. 1097–1105.

[7] Mark Law, Alessandra Russo, and Krysia Broda. "Inductive Learning of Answer Set Programs". In: *Logics in Artificial Intelligence*. Springer International Publishing, 2014, pp. 311–325.

[8] Mark Law, Alessandra Russo, and Krysia Broda. "Inductive learning of answer set programs from noisy examples". In: *arXiv preprint arXiv:1808.08441* (2018).

[9] Mark Law, Alessandra Russo, and Krysia Broda. "Iterative learning of answer set programs from context dependent examples". In: *Theory and Practice of Logic Programming* 16.5-6 (2016), pp. 834–848.

[10] Mark Law, Alessandra Russo, and Krysia Broda. "Learning weak constraints in answer set programming". In: *Theory and Practice of Logic Programming* 15.4-5 (2015), pp. 511–525.

[11] Mark Law, Alessandra Russo, and Krysia Broda. *The ILASP system for learning Answer Set Programs*. `www.ilasp.com`. 2015.

[12] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. “Deep learning”. In: *Nature* 521 (2015), pp. 436–444.

[13] Pengfei Liu, Xipeng Qiu, and Xuanjing Huang. “Recurrent neural network for text classification with multi-task learning”. In: *arXiv preprint arXiv:1605.05101* (2016).

[14] Stephen Muggleton. “Inductive logic programming”. In: *New generation computing* 8.4 (1991), pp. 295–318.

[15] Joseph Redmon and Ali Farhadi. “Yolov3: An incremental improvement”. In: *arXiv preprint arXiv:1804.02767* (2018).

[16] Joseph Redmon et al. “You only look once: Unified, real-time object detection”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 779–788.

[17] Shaoqing Ren et al. “Faster r-cnn: Towards real-time object detection with region proposal networks”. In: *Advances in neural information processing systems*. 2015, pp. 91–99.

[18] Adrian Rosebrock. *Intersection over Union (IoU) for object detection*. Nov. 7, 2016. URL: `https://www.pyimagesearch.com/2016/11/07/intersection-over-union-iou-for-object-detection` (visited on 01/15/2020).