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 8, 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 [13]. 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 [13], and recurrent neural networks (RNNs), which are designed to process sequences of arbitrary length [14]. 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.

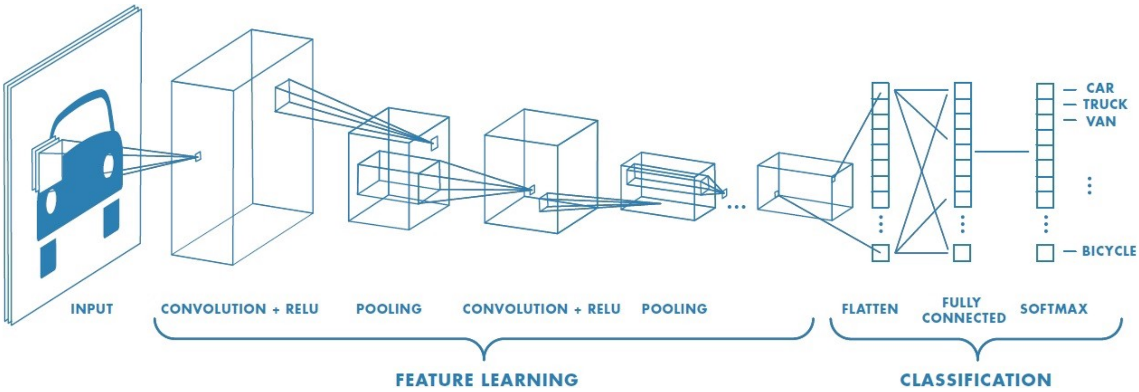


Figure 1.1: An example of a CNN architecture. The input image is passed through a series of convolution and pooling layers before being flattened into a one-dimensional layer and passed through one final fully connected layer. The softmax classification function is then applied at the output.

1.2 Image Processing

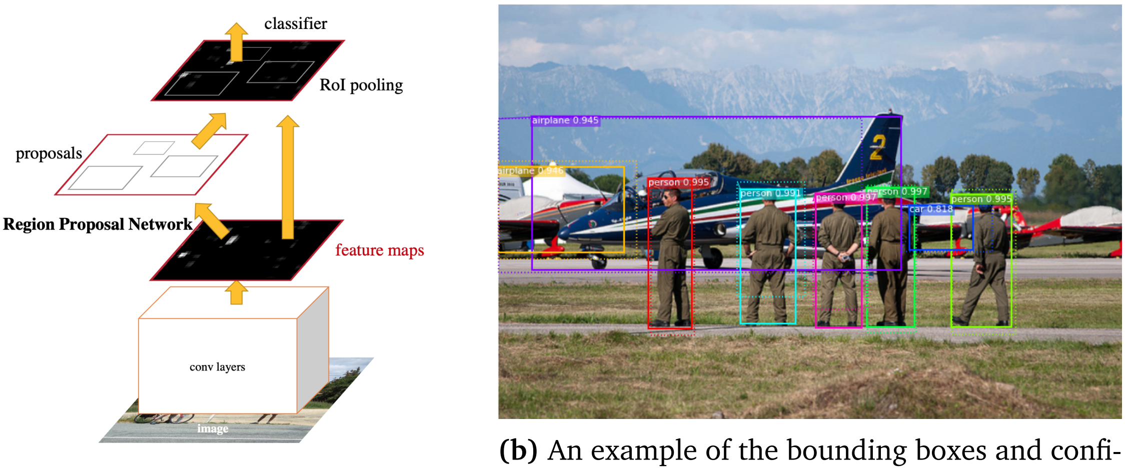
Convolutional networks have proven to be adept at a number of tasks involving images, including image classification [7] and object detection [16, 18]. This section explores the object detection task, as well as metrics for both tasks, further. Object detection forms an important component of the Hybrid 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* [18] and *You Only Look Once* (YOLO) [16].

Faster R-CNN is an evolution of previous object detection algorithms, R-CNN [6] and Fast R-CNN [5]. 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 [6] and [5]) 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 [18].

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

Figure 1.2

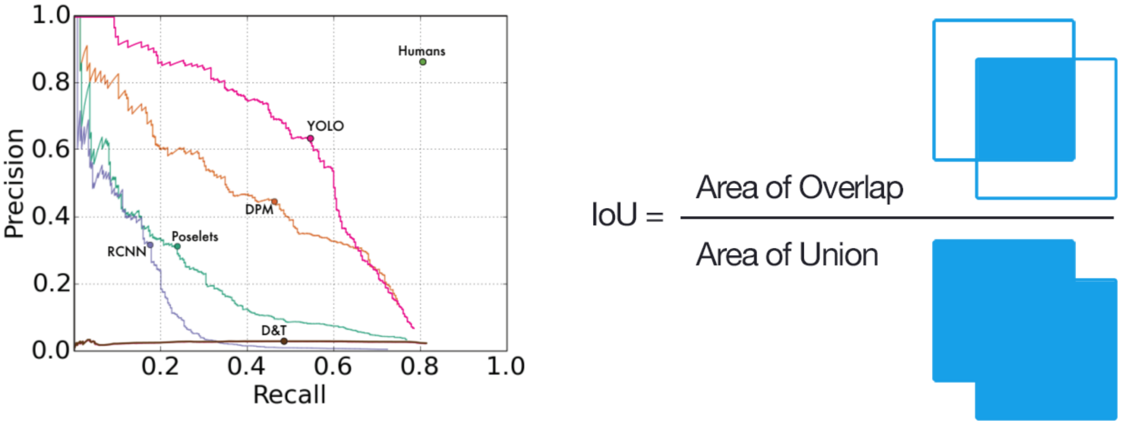
for each region and finally removing any low-scoring or redundant regions. This requires the algorithm to 'look' at the image multiple times (around 2000 times for R-CNN). You Only Look Once (YOLO) is a significantly more efficient algorithm which, as the name suggests, takes a single look at the image. 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 [17]. YOLO is, however, slightly less accurate than some of the slower methods for object detection [16].

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 [17].

(b) Visual explanation of the intersection over union metric. Image from [19].

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 **??**.

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 [4]. $\mathcal{AL}$'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 2 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 [15] (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 [12] (Inductive Learning of Answer Set Programs) is an ILP framework for learning ASP programs.

The authors of [8] 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 [12].

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 [11] (a method for specifying preferences in ASP), learning from context dependent examples [10] and learning from noisy examples [9].

Chapter 2

Hardcoded Model

Our first H-PERL implementation is the ‘hardcoded’ model. The hardcoded model makes use of a mixture of manually engineered components and components which are trained on the full-data version of the OceanQA dataset. This model should not be taken as a solution to the general VideoQA problem, since it would be labourious to rewrite components for each new dataset environment. Instead we intend this model to be used as a benchmark for the OceanQA dataset, against which other VideoQA implementations can be evaluated.

Full details on the performance of the model, as well as the performance of some of the individual components is given in Chapter **??**.

2.1 Properties

As mentioned in Chapter **??**, the role of the properties component is to take an image of an object (as produced by the object detector) and, in the case of the OceanQA dataset, to return the colour and rotation of the object.

Since the object image is a 3D tensor[1] of raw pixel values, a convolutional neural network is an excellent candidate for the implementation of the properties component. Other computer vision techniques for machine learning such as decision trees, random forests and SVMs require thousands of manually engineered filters to be applied to the image, whereas convolutional networks can learn a much smaller set of filters based on the training data.

Figure 2.1 shows the architecture of the properties network. The first part of this network encodes the object image into a 1024 dimenional latent vector using a series of convolutional and fully-connected layers. A set of fully-connected layers, one for each property, each take the vector encoding of the object and produce a vector of real numbers. The size of each vector is equal to the number of possible values that property can take.

[1] Height, width and RGB colour channels make up the three dimensions

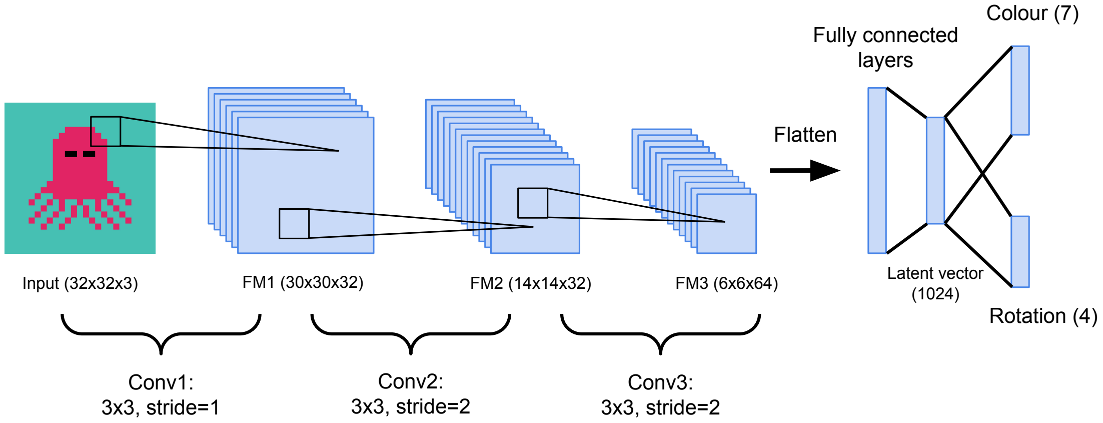


Figure 2.1: Architecture of the properties component neural network. An object is encoded into a latent vector, before a set of fully-connected layers produce a set of probability distributions over the property values. Batch normalisation is applied between convolutional layers. FM is a set of feature maps, the size of each feature map is given as the tuple (*height*, *width*, *number of feature maps*).

The final output of the network is a set of probability distributions, one for each property, over the set of possible values that property can take. As is standard in a multiclass classification problem, the softmax function is applied to each set of property values in order to construct the probability distribution. The softmax function guarantees that the elements of a vector sum to one and that each element is greater than or equal to zero, hence softmax creates a discrete probability distribution. The softmax function for a vector $\mathbf{z} \in \mathbb{R}^K$ is as follows:

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \text{ for } i = 1, ..., K \tag{2.1}$$

The full-data version of the OceanQA dataset labels both the colour and rotation of every object in every frame. This makes training a neural network relatively straightforward; we collate all of the objects and their properties in the dataset, resize each object to 32x32 pixels, convert each property into a one-hot encoded vector and train the network using batches of 256 objects with a learning rate of 0.001 for 2 epochs. The cross-entropy loss is calculated for each property and these are summed to give an overall loss. The cross-entropy loss between a predicted probability distribution vector $\mathbf{p} \in \mathbb{R}^K$ and a one-hot encoded classification vector $\mathbf{y} \in \mathbb{R}^K$, where K is the number of classes, is as follows:

$$H(\mathbf{p}, \mathbf{y}) = -\sum_{c=1}^{K} y_c \log(p_c) \tag{2.2}$$

When the H-PERL model is being evaluated each object in the video is applied to the network (batched together for efficiency) and a set of probability distributions is produced. For each object, the property value for a particular property is given by the most probable element of the vector.

2.2 Relations

The job of the relations component is to list all instances of relevant binary relations between objects in each frame of the video. Since there is only one relevant binary relation in the OceanQA dataset, the job of this component is simply to list the instances of the *close* relation.

The relations component of the hardcoded model contains a hand-written binary classification algorithm for determining whether two objects are close or not. For a given video, this algorithm is applied to every pair of objects in every frame of the video in order to list all instances of the relation. The object arguments of the closeness algorithm are given in symbolic form, rather than as raw pixel matrices. This means the algorithm is heavily reliant on accurate information from the object detector - the position tuple in particular. Although the closeness algorithm doesn't make use of the property component's extracted features, other algorithms for determining binary relations may require these.

The algorithm for determining the closeness of two objects is, in fact, identical to the algorithm used when constructing the dataset. The algorithm uses the idea of an expanded box around each object; the objects are close if their boxes overlap, as can be seen in Figure 2.2. The algorithm was discussed alongside techniques used to create the dataset in Chapter **??** and is shown in Algorithm **??**.

Clearly, since the algorithm used to construct the dataset and the algorithm used to find the relations between objects are exactly the same, the relations component achieves perfect accuracy provided the object detection is exactly correct. Although this may seem like cheating, since in general it is not possible to know the underlying rules of the dataset, we reiterate that the hardcoded model is not a solution to VideoQA tasks in general, but rather is specific to the OceanQA dataset, and can be used for comparisons with other models.

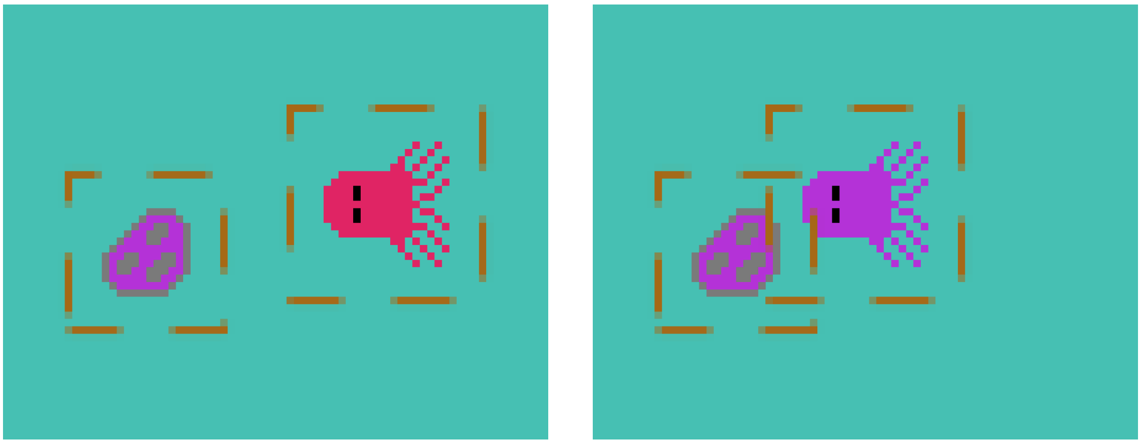


Figure 2.2: Diagrams showing the octopus before and after moving close to a purple rock, and therefore turning purple itself. The brown dashed lines show the bounding boxes expanded by 5 pixels on each side around the objects. If these boxes overlap the objects are deemed to be close to one another.

Another consideration for this approach to relation classification is speed; each relation classification algorithm (of which there is only one in the OceanQA dataset, but, in general there may be many) looks at every possible pair of objects in every frame of the video. Assuming that each relation classifier operates in constant time, this creates an overall algorithmic complexity of $\mathcal{O}(kmn^2)$, where k is the number of frames per video, m is the number of relations to be classified and n is the number of objects in each frame. Despite this we found that the time taken by the relation component was small relative to other components in the H-PERL model. Chapter **??** outlined the full details of the component's performance, including details on the time taken during evaluation.

2.3 Events

As mentioned in Chapter **??**, the role of the event detection component is to list all of the actions and effects which take place between each pair of consecutive frames of a given video. Since preceeding components have extracted a number of object and frame-level symbolic features already, the event detector in the hardcoded model works only with these features, as opposed to viewing the raw pixels in each frame.

For the hardcoded model, the event detector is implemented by, firstly, searching through all possible combinations of actions. The component assumes that there is only one action per frame and that only non-static objects can be the cause of an action. Each combination of actions is then applied to a hand-written $\mathcal{AL}$ model of the environment to generate the set of symbolic features which would be expected to be observed should the set of actions have occurred. This set of features is then compared to the set of features which were actually observed - the set of features extracted by preceeding components. Finally, the combination of actions which lead to the best fit with the observed data are selected.

For each set of possible actions and their associated features generated by the $\mathcal{AL}$ model, a set of manually engineered ASP rules is used to find the corresponding set of effects. For example, if the octopus moves[2] during frame i and is close to a blue rock during frame $i+1$, then we can infer that the *change colour* effect occurs during frame i.

2.3.1 The $\mathcal{AL}$ Model

As described in Chapter 1, $\mathcal{AL}$ is a formal model for describing the behaviour of a dynamic system. An $\mathcal{AL}$ system consists of a set of states and some way of transitioning between states using actions. We model our video as an $\mathcal{AL}$ system by considering each frame as a state and object actions between frames as $\mathcal{AL}$ actions. In $\mathcal{AL}$ a state is described by a set of fluents. Each fluent, <*f*>, can be

[2] Moving during frame i refers to an object moving between frame i and frame $i+1$.

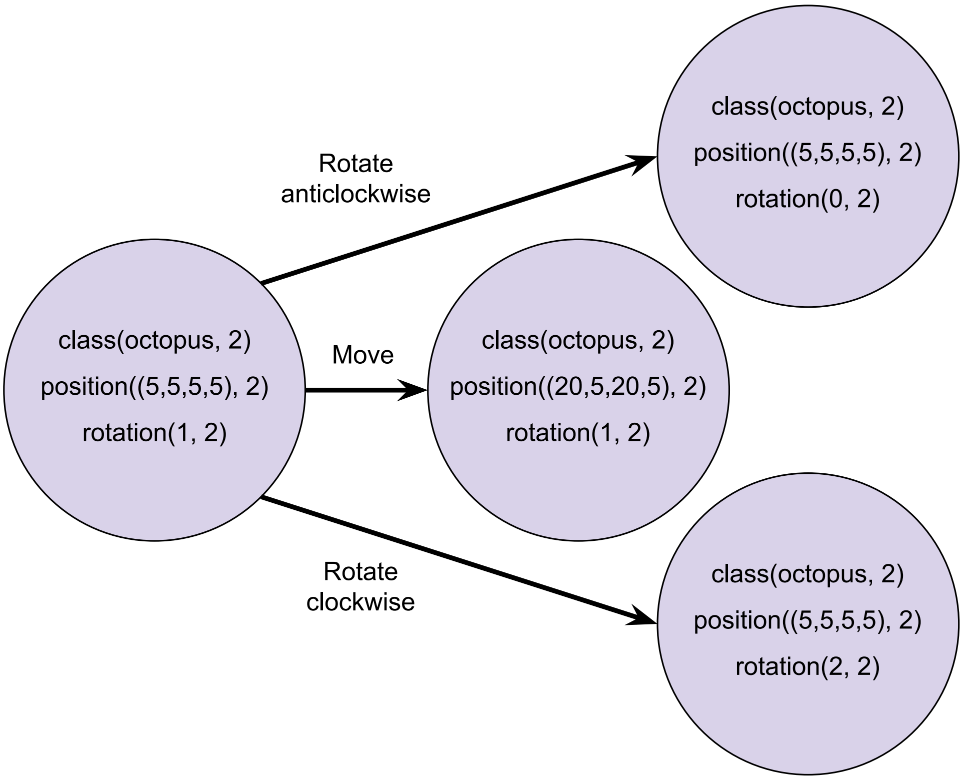


Figure 2.3: A simplified example set of $\mathcal{AL}$ states and transitions. States which model the OceanQA environment contain many more fluents than are shown here.

wrapped up in a predicate, $holds(<f>, <i>)$, where <*i*> is a timestep in the video, which means that fluent <*f*> is true at step <*i*>. Conversely, $\neg holds(<f>, <i>)$ means that <*f*> is not true at step <*i*>. An OceanQA-specific example of a set of $\mathcal{AL}$ states and transitions between them is shown in Figure 2.3.

As mentioned above, the optimal set of actions is found by comparing the observed features of the environment with an internal $\mathcal{AL}$ model of the environment. In the rest of this section we outline the $\mathcal{AL}$ system description for the OceanQA environment. This system description can then be written in ASP using the encoding provided in Appendix **??**.

Firstly, we outline the types involved in the $\mathcal{AL}$ system description, which are as follows:

- Properties, including class and position, are modelled as *inertial fluents* - their values can change as a direct result of the actions taken. Notice, however, that the number of possible position values is very large - 256^4, hence, rather than allowing all possible values, we only include values that have been observed in the video, all other values are not modelled. This applies to all properties, class and position fluents.

- An additional inertial fluent, $exists(<id>)$, is also required. Intuitively, it means that an object with identifier <*id*> is present in the current timestep.

- The close relation, which is written in ASP as $close(Id1, Id2)$ when an object with identifier $Id1$ is close to an object with identifier $Id2$, is considered a *defined fluent*. Although they are modelled as defined fluents, their thruthiness cannot be altered by the events component, since relation classification is handled by the relations component. Therefore, these fluents are copied directly across from the observed data into the $\mathcal{AL}$ model.

- We add a further defined fluent, which also comes directly from the observed information. This fluent is called $disappear($<*id*>$)$, and, naturally, means that an object with identifier <*id*> disappears immediately after the current timestep.

- Actions are, of course, modelled by the *action* type. The action itself takes a single argument - the object's identifier. In ASP each action is written as <*action*>(<*id*>).

The domain independent rules for the OceanQA environment are as outlined in Appendix **??**, with the following addition:

$$\neg holds(F, 0) \text{ :- } fluent(inertial, F), not\ holds(F, 0). \tag{2.3}$$

This rule ensures that the initial state of the system is complete for inertial fluents - all inertial fluents are either true or false. While this isn't a requirement for $\mathcal{AL}$ systems, it simplifies the rest of the model since there is no uncertainty about the system's state.

Finally, we outline the domain dependent $\mathcal{AL}$ statements for the OceanQA environment:

- The *move* action causes the object to move 15 pixels in the direction of rotation. This means that, in frame $i + 1$, the object is no longer in the position it was in during frame i. We use a Python function, new_pos, in the ASP program to calculate the object's next position. This function is called using the @ symbol. The $\mathcal{AL}$ *causal laws* for the move action are the following:

  $$move(Id) \textbf{ causes } position(@new\_pos(P, R), Id) \textbf{ if } position(P, Id), rotation(R, Id)$$
  $$move(Id) \textbf{ causes } \neg position(position(P, Id)) \textbf{ if } position(P, Id)$$

- $\mathcal{AL}$ causal laws follow the same pattern for both rotation actions. The Python function, new_rot, is used to calculate the new rotation for the object, it takes the previous rotation and the type of rotation as arguments. In the following $\mathcal{AL}$ causal laws <d> is used to refer to the direction of rotation:

  $$rotate\_\text{<d>}(Id) \textbf{ causes } rotation(@new\_rot(R, rotate\_\text{<d>}) \textbf{ if } rotation(R, Id)$$
  $$rotate\_\text{<d>}(Id) \textbf{ causes } \neg rotation(R, Id) \textbf{ if } rotation(R, Id)$$

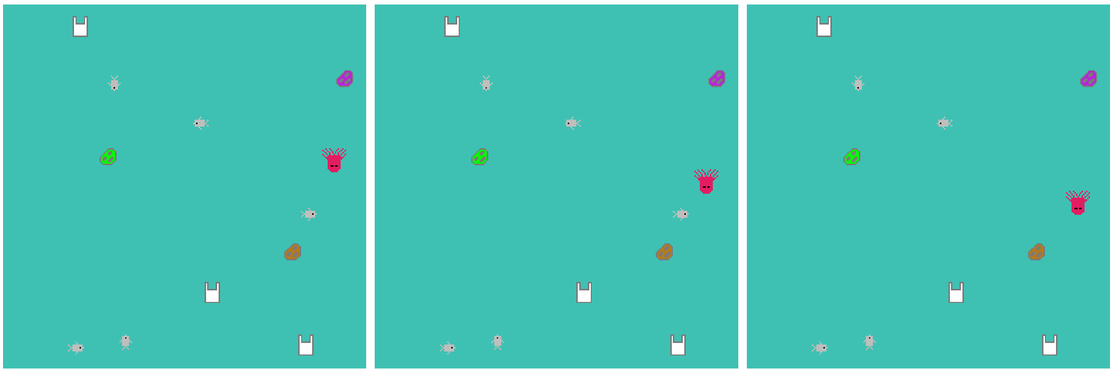


Figure 2.4: An example of the *eat a fish* effect.

- In order to ensure that objects are modelled as non-existent after they have disappeared, the following causal law is added:

$$move(Id)\ \textbf{causes}\ \neg exists(Id)\ \textbf{if}\ exists(Id), disappear(Id)$$

- Finally, the following *executability conditions* are added to reduce the size of the search space for the optimiser:

$$\begin{gathered}\textbf{impossible}\ move(Id)\ \textbf{if}\ \neg exists(Id)\\ \textbf{impossible}\ rotate\_clockwise(Id)\ \textbf{if}\ \neg exists(Id)\\ \textbf{impossible}\ rotate\_anticlockwise(Id)\ \textbf{if}\ \neg exists(Id)\end{gathered}$$

The use of the $exists(<\!id\!>)$ fluent is an intuitive way of encoding the knowledge that when an object disappears it no longer exists, and, if an object does not exist, it cannot participate in any actions. Without this predicate, encoding knowledge like this would be tricky because the $disappear(<\!id\!>)$ fluent appears (at most) once during the video.

In the remainder of this section we diverge from $\mathcal{AL}$ orthodoxy in order to model effects. Effects are defined by ASP rules with a combination of actions and $\mathcal{AL}$ fluents in the body. These rules can mostly be considered to observe the $\mathcal{AL}$ model, rather than affect it. However, an exception is made for the rules which update the octopus' colour.

Firstly, for the *eat a fish* (an example of which is shown in Figure 2.4) and *eat a bag* effects, the following ASP rules are used:

$$\begin{aligned}occurs\_effect(eat\_a\_fish(Octo), I)\text{ :- }& occurs\_action(move(Octo), I),\\ & holds(class(fish, Fish), I),\\ & holds(disappear(Fish), I).\end{aligned} \tag{2.4}$$

$$\begin{aligned}occurs\_effect(eat\_a\_bag(Octo), I)\text{ :- }& occurs\_action(move(Octo), I),\\ & holds(class(bag, Bag), I),\\ & holds(disappear(Bag), I),\\ & holds(disappear(Octo), I).\end{aligned} \tag{2.5}$$

We also need to ensure the octopus' colour is updated when it comes close to rock, and that we keep track of any *change colour* effects that occur. To do this we, firstly, use the following helper predicate:

$$\begin{aligned} \mathit{change\_colour}(\mathit{Old}, \mathit{New}, \mathit{Id}, I-1) \text{ :- } & \mathit{holds}(\mathit{class}(\mathit{octopus}, \mathit{Id}), I), \\ & \mathit{holds}(\mathit{close}(\mathit{Id}, \mathit{IdRock}), I), \\ & \mathit{holds}(\mathit{colour}(\mathit{Old}, \mathit{Id}), I-1), \\ & \mathit{holds}(\mathit{class}(\mathit{rock}, \mathit{IdRock}), I), \\ & \mathit{holds}(\mathit{colour}(\mathit{New}, \mathit{IdRock}), I), \\ & \mathit{Old} \mathrel{!}= \mathit{New}, \\ & \mathit{step}(I-1). \end{aligned} \tag{2.6}$$

We also add the following ASP rule for the *change colour* effect:

$$occurs\_effect(\mathit{change\_colour}(\mathit{Octo}), I) \text{ :- } \mathit{change\_colour}(\mathit{Old}, \mathit{New}, \mathit{Octo}, I). \tag{2.7}$$

Finally, the octopus' colour is updated using the follwing rules:

$$\mathit{holds}(\mathit{colour}(\mathit{New}, \mathit{Id}), I+1) \text{ :- } \mathit{change\_colour}(\mathit{Old}, \mathit{New}, \mathit{Id}, I). \tag{2.8}$$

$$\neg \mathit{holds}(\mathit{colour}(\mathit{Old}, \mathit{Id}), I+1) \text{ :- } \mathit{change\_colour}(\mathit{Old}, \mathit{New}, \mathit{Id}, I). \tag{2.9}$$

Rules 2.8 and 2.9 alter the $\mathcal{AL}$ state, despite not being generated from $\mathcal{AL}$ statements. There are other $\mathcal{AL}$ models for the OceanQA environment (not discussed here) which do not suffer from the same problem. However, these models can be more complex, which leads to a higher probability of human error. Instead we prefer to make a slight adaption to the $\mathcal{AL}$ rules in order to allow a simpler model to be defined.

2.3.2 Action Optimisation

As we have seen, the $\mathcal{AL}$ model uses the $\mathit{holds}(<\!f\!>, <\!i\!>)$ predicate to store its information, but the observed data uses $\mathit{obs}(<\!f\!>, <\!i\!>)$. This distinction is by design; it separates the information (particularly that which is modelled by inertial fluents) in the two data models. However, we need some way of comparing the two models in order to find the optimal action combination.

Firstly, in order to ensure that both data models start with the same information, the fluents for the initial frame in the observed data are copied over to the $\mathcal{AL}$ model. Secondly, the ASP optimisation program generates all possible action combinations and applies each combination to the $\mathcal{AL}$ model to generate the data expected from that particular set of actions. These sets of action can be generated in ASP using the following choice rule:

$$occurs\_action(A, I) : \mathit{action}(A) \text{ :- } \mathit{step}(I+1), I >= 0. \tag{2.10}$$

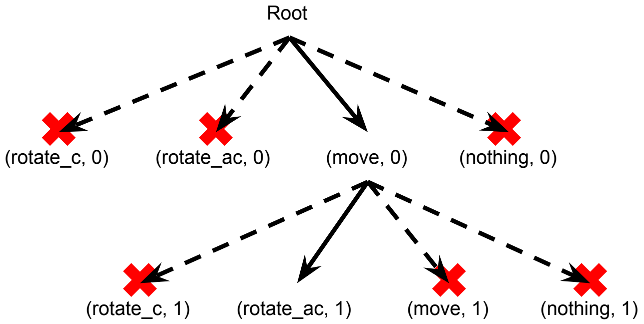


Figure 2.5: Part of an example action search tree. Each action is given as a pair of $(action\_name, timestep)$. Red crosses and dashed lines show pruned branches.

Finally, in order to give each set of actions a score, the expected data is compared with the observed data and the number of mismatched fluents is counted. In ASP a *weak constraint* can be used to assign a cost to an answer set, the ASP optimiser then searches for the answer set with the lowest cost. We use the following two weak constraints to optimise the action search:

$$:\sim \neg obs(exists(Id), I), holds(exists(Id), I).[1@1, exists(Id), I] \tag{2.11}$$

$$:\sim obs(F, I), \neg holds(F, I).[1@2, F, I] \tag{2.12}$$

Rule 2.11 is given a higher priority, so ASP searches for a set of answer sets which minimise the number of times the body is true before considering Rule 2.12. Rule 2.11 says that we prefer answer sets with the fewest mismatches due to the $exists$<*id*> fluent. Rule 2.12, on the other hand, says that we prefer answer sets with the fewest mismatches between all fluents. Rule 2.11 is given a higher priority, firstly, because it is more specific, and, secondly, because we consider an error in modelling the existence of an object as more significant than errors in modelling other fluents.

The following two hard constraints are also applied to ensure that basic rules of the OceanQA environment are not broken:

$$:\text{-}\ obs(class(octopus, Id), I), occurs\_action(nothing(Id), I). \tag{2.13}$$

$$:\text{-}\ \neg obs(class(octopus, Id), I), \neg occurs\_action(nothing(Id), I). \tag{2.14}$$

The use of hard rather than weak constraints in Rules 2.13 and 2.14 improves the speed of the ASP optimisation. However, the weak constraints in Rules 2.11 and 2.12 are necessary since the program must be able to deal with noisy data. If hard constraints were used instead any noise in the data could rule out all possible answer sets.

All of the above rules, as well as the ASP encoding of the $\mathcal{AL}$ model and the observed data, are run in a single ASP program which attempts to find the optimal set of actions for the video. Speed is a genuine concern for this optimisation; the ASP program has to evaluate 4^{32} action combinations, which could take a very long time. The ASP optimiser, however, uses optimisation strategies which can decrease the total search time. These algorithms include an ASP-specific extension of the branch-and-bound algorithm [3], which allows the optimiser to prune branches of the search tree if the branch is provably sub-optimal. Part of an example search tree for the action optimisation is shown in Figure 2.5. For most videos in the evaluation dataset, the ASP optimisation is able to run in under 5 seconds, however, when the observed data is noisier, the algorithm has to explore many more answer sets, which takes significantly longer. This is discussed further in Chapter **??**.

2.4 Error Correction

As we have already discussed, hardcoded models or components come with a number of drawbacks: they can be very complex to write; they have to be rewritten for each new environment; and they may introduce human error or bias. However, hardcoded models also come with a key advantage: if the component has a deep understanding of the environment, it can attempt to correct errors in the input, or 'de-noise' the data that it is given.

The hardcoded H-PERL model presented here attempts to correct only a single type of error - namely, an error in the object detection which leads to uncertainty in the object tracking. An example of a mistake made by the object detector is shown in Figure 2.6. The uncertainty in the object tracker is due to the detector wrongly detecting two objects of the same type very close to each other; if there is only one of that type of object in the previous frame, the tracker will be unsure about which of the new objects should be assigned the previous object's identifier.

To combat this, we allow the tracker to assign the same identifier to multiple objects and then, during the event detection phase, the search space is extended to include searching over which object should be assigned the identifier. The algorithm can be outlined in more detail as follows:

1. The object tracker works in the same way as before, except when multiple objects want to be assigned the same identifier. Unlike before, the tracker now assigns the identifier to all objects involved.

2. For each frame, we keep track of the identifiers and the set of objects which are competing for each identifier.

3. The ASP input in the event detection component is updated to include a choice rule which creates a set of object-identifier assignments. For each assignment, one of the competing objects is assigned the identifier and all others are assigned new, unused identifiers.

4. Each assignment is evaluated using the optimisation outlined in Section 2.3 and the optimal assignment is chosen. In essence, this means that the object-identifier assignment which leads to the fewest differences between the data

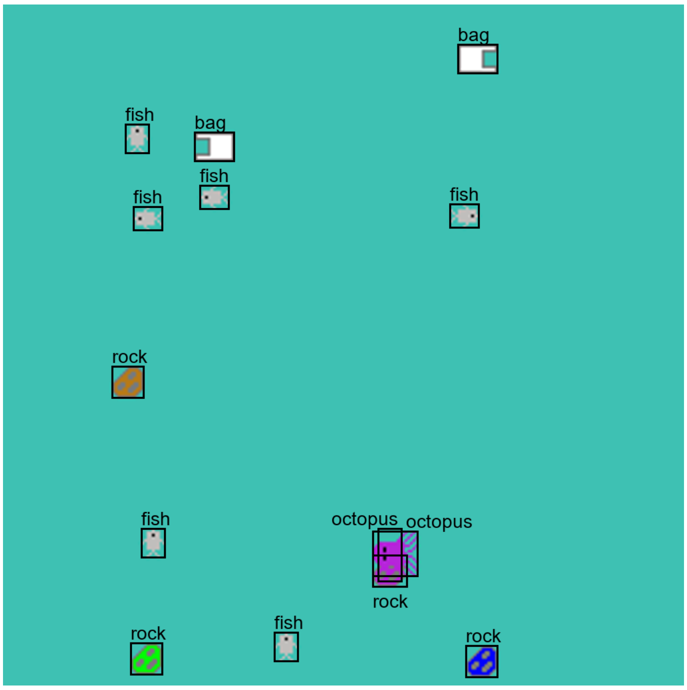


Figure 2.6: An example of an error in the object detection. Two purple octopuses are detected in this frame. This leads to competion for the identifier of the 'real' octopus.

in the $\mathcal{AL}$ model and the observed data is chosen. Intuitively, this reflects the assumption that the assignment with the fewest mistakes is the one most likely to maximise the number of questions answered correctly.

5. Finally, the chosen objects are assigned their respective identifiers, while the other objects are assigned new, unused identifiers. As mentioned in Chapter **??**, this is the only occasion where a component of an H-PERL model can overwrite previously extracted information.

This algorithm for correcting errors is only possible when a model of the dataset's environment is known. Hence, the hardcoded model is the only model outlined in this report which is capable of error correction.

As mentioned in Section 2.3, the amount of time the ASP optimiser takes can be a serious problem, especially if the data is very noisy, or in this case, if the object detector makes a lot of mistakes. However, our detector is highly accurate and so optimisation speed does not cause significant problems.

The hardcoded model has been designed to work with or without error correction. The full details of the model's performance in both modes of operation is outlined in Chapter **??**.

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] Martin Gebser et al. “Multi-criteria optimization in answer set programming”. In: *Technical Communications of the 27th International Conference on Logic Programming (ICLP’11)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. 2011.

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

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

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

[11] 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.

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

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

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

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

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

[17] 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.

[18] 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.

[19] 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).