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 6, 2020

Chapter 1

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 **??**.

1.1 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 1.1 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.

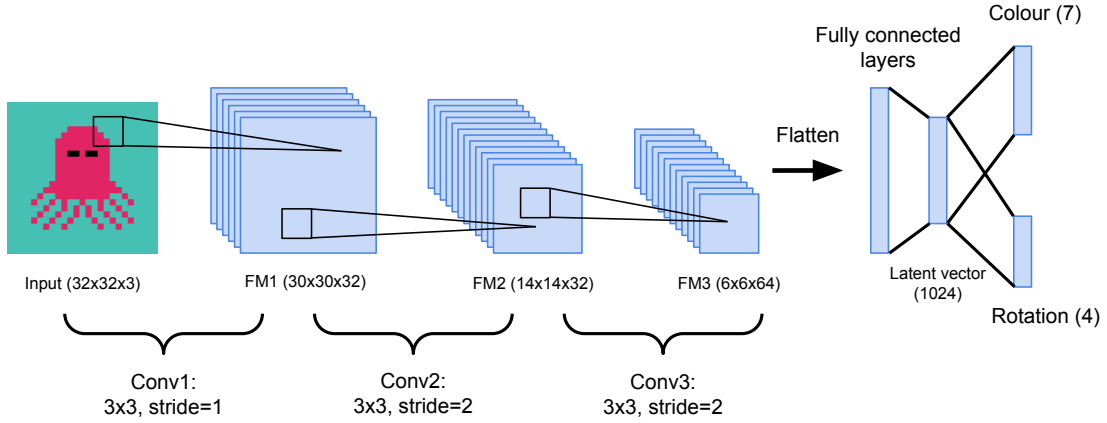[1]Height, width and RGB colour channels make up the three dimensions

Figure 1.1: 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 stands for 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:

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \text{ for } i = 1, ..., K \tag{1.1}$$

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. 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:

$$H(\mathbf{p}, \mathbf{y}) = -\sum_{c=1}^{K} y_c \log(p_c) \tag{1.2}$$

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.

1.2 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 1.2. 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.
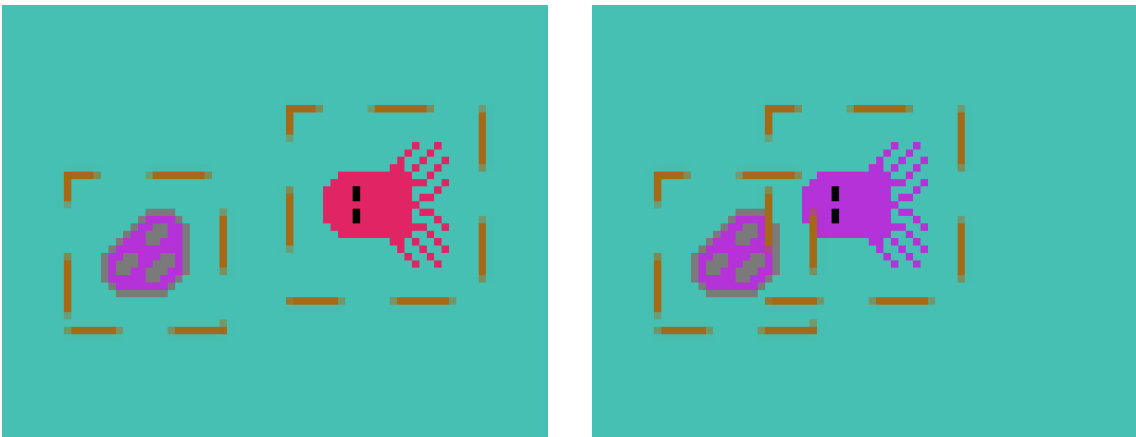

Figure 1.2: Diagrams showing the octopus before and after moving close to a purple rock, and therefore turning purple itself. The brown dashed lines show the bounding boxes expanded by 5 pixels on each side around the objects. If these boxes overlap the objects are deemed to be close to one another.

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.

1.3 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.

1.3.1 The $\mathcal{AL}$ Model

As described in Chapter **??**, $\mathcal{AL}$ 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 $i+1$.
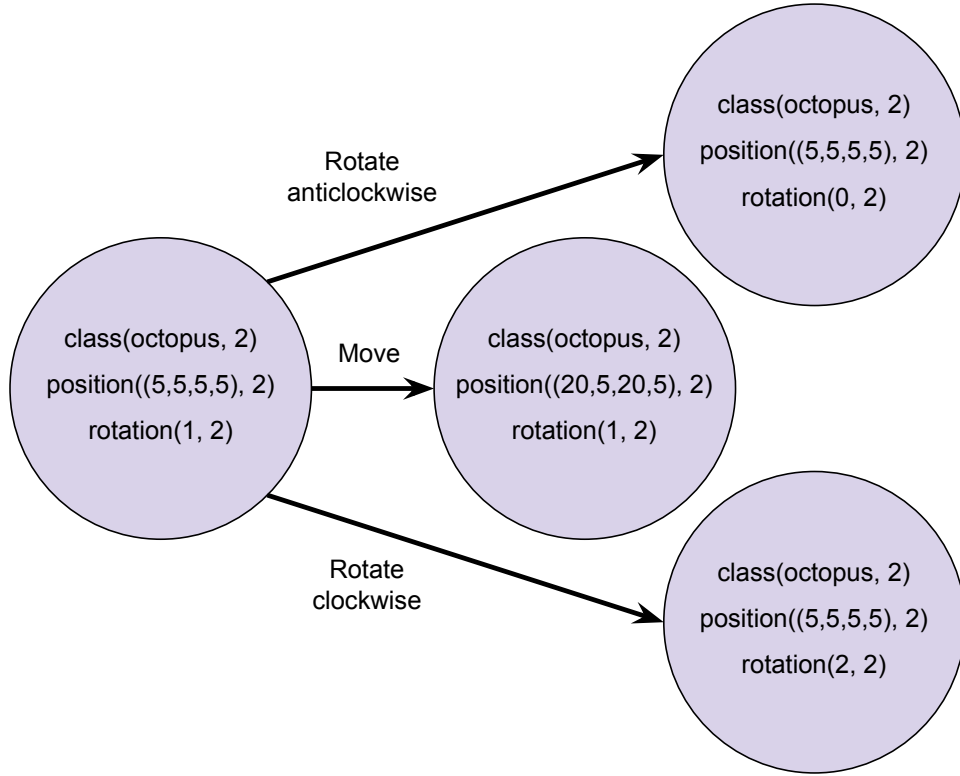
Figure 1.3: A simplified example set of $\mathcal{AL}$ states and transitions. States which model the OceanQA environment contain many more fluents than are shown here.

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 1.3.

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 applied 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 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:

$$\neg holds(F, 0) \text{ :- } fluent(inertial, F), not\ holds(F, 0). \tag{1.3}$$

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:

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

Figure 1.4: 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:

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

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 1.4) and *eat a bag* effects, the following ASP rules are used:

$$
\begin{aligned}
occurs\_effect(eat\_a\_fish(Octo), I) \text{ :- } & occurs\_action(move(Octo), I), \\
& holds(class(fish, Fish), I), \\
& holds(disappear(Fish), I).
\end{aligned}
\tag{1.4}
$$

$$
\begin{aligned}
occurs\_effect(eat\_a\_bag(Octo), I) \text{ :- } & occurs\_action(move(Octo), I), \\
& holds(class(bag, Bag), I), \\
& holds(disappear(Bag), I), \\
& holds(disappear(Octo), I).
\end{aligned}
\tag{1.5}
$$

We also need to ensure the octopus' colour is updated when it comes close to rock. To do this we use the following helper predicate:

$$
\begin{aligned}
change\_colour(Old, New, Id, I-1) \text{ :- } & holds(class(octopus, Id), I), \\
& holds(close(Id, IdRock), I), \\
& holds(colour(Old, Id), I-1), \\
& holds(class(rock, IdRock), I), \\
& holds(colour(New, IdRock), I), \\
& Old\ != New, step(I-1).
\end{aligned} \tag{1.6}
$$

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

$$
holds(colour(New, Id), I+1) \text{ :- } change\_colour(Old, New, Id, I). \tag{1.7}
$$

$$
\neg holds(colour(Old, Id), I+1) \text{ :- } change\_colour(Old, New, Id, I). \tag{1.8}
$$

Rules 1.7 and 1.8 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 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.

1.3.2   Action Optimisation

As we have seen, the $\mathcal{AL}$ model uses the $holds(<f>, <i>)$ predicate to store its information, but the observed data uses $obs(<f>, <i>)$. 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 optimise the 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:

$$
occurs\_action(A, I) : action(A) \text{ :- } step(I+1), I >= 0. \tag{1.9}
$$

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 and the ASP optimiser searches for the answer set with the lowest cost. We use the following two weak constraints to optimise the action search:

$$
\text{:}\sim \neg obs(exists(Id), I), holds(exists(Id), I).[1@1, exists(Id), I] \tag{1.10}
$$

$$
\text{:}\sim obs(F, I), \neg holds(F, I).[1@2, F, I] \tag{1.11}
$$

Rule 1.10 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 1.11. Rule 1.10 says that we prefer answer sets with the fewest mismatches due to the $exists<id>$ fluent. Rule 1.11, on the other hand, says that we prefer answer sets with the fewest mismatches between all fluents. Rule 1.10 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:

$$:- obs(class(octopus, Id), I), occurs\_action(nothing(Id), I). \qquad (1.12)$$
$$:- \neg obs(class(octopus, Id), I), \neg occurs\_action(nothing(Id), I). \qquad (1.13)$$

The use of hard rather than weak constraints in Rules 1.12 and 1.13 improves the speed of the ASP optimisation. However, the weak constraints in Rules 1.10 and 1.11 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 [1], 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 1.5. For most videos in the evlauation dataset, the ASP optimisation is able to run in under 5 seconds, however, if the observed data was noisier, we would expect that the algorithm would have to explore many more answer sets, which could take significantly longer.
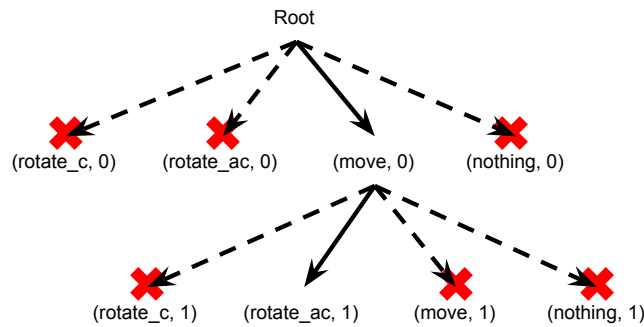

Figure 1.5: Part of an example action search tree. Each action is given as pair of $(action\_name, timestep)$. Red crosses and dashed lines show pruned branches.

Figure 1.6: 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.

1.4   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 re-written 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 1.6. 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 both 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 1.3 and the optimal assignment is chosen. In essence, this means that the object-identifier assignment which leads to the fewest differences between the data 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.

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 1.3, 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] 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.