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 7, 2020

Chapter 1

OceanQA Dataset

Before discussing the implementation of our solution to the VideoQA task, it is important to outline the data that is used to train and evaluate the model. Rather than using one of the datasets outlined in Chapter **??**, we opted to create a new VideoQA dataset, which we name 'OceanQA'. While it would have been preferable to use an existing dataset (and an existing implementation as a baseline) to allow a fair comparison, none of the existing datasets suited the project requirements[1], for the following two reasons:

1. Most of the existing VideoQA datasets use videos from real-world environments, where objects and events are usually more complex than computer-generated environments. Training models to work with real-world data therefore requires significant computational resources and can take days or even weeks. Given the time and resource limitations that exist for this project, it was sensible to avoid these datasets. More generally, creating a dataset gives greater flexibility over the size and complexity of the data; if faster training is required, we can simply create smaller images or use fewer objects in each video.

2. Hybrid models generally require some form of environment specification (or ontology) so that the model's internal knowledge can be represented explicitly (see [11, 5, 12] for examples from VideoQA and VQA). Since most of the existing VideoQA datasets do not limit objects to be of specific types, or restrict object properties or video events to a given set, they cannot provide such a specification of the environment.

The dataset is generated programmatically and can therefore be made as large as required. However, in order to allow comparisons between models, we generate a fixed dataset of 1400 videos (each video contains 10 question-answer pairs) and use this data to train and evaluate the models outlined in subsequent Chapters. This dataset is divided into 1000 training, 200 validation and 200 testing videos.

[1]The CLEVRER dataset [11] meets these requirements and would have been a good candidate for this project. Unfortunately, it was published in March 2020, six months after the project began.

The generated dataset can be used in either 'full-data' form or in 'QA-data' form. The full-data form contains videos, question-answer pairs and the ground truth of all the information in the videos; this means that every object property, relation and event is labelled by the dataset. This form gives the programmer complete access to the information in the video, which allows baseline models to be constructed, and may also allow internal parts of models to be evaluated. The QA-form, on the other hand, contains only videos and question-answer pairs. Since no additional data about the video is provided, this form reflects a 'real' VideoQA dataset, and should be used for evaluating the model as a whole.

Since the focus of this project is to investigate logical reasoning, we do not attempt to make the job of the neural network difficult by creating complex scenes; the dataset emulates a simple retro-game environment. Each image is also quite small at 256x256 pixels to allow faster network training. The remainder of this Chapter outlines the full details of the OceanQA dataset.

1.1 Videos

Each video in the OceanQA dataset is a sequence of 32 frames. Each frame contains a flat background and a maximum of 16 objects. Objects form the central component of each video, since all of the useful information in each video can be modelled by the following: properties of objects; binary relations between objects; and events, which relate to at least one object, occuring between two consecutive frames of the video. Each object is modelled using the following attributes: object type (or class), position, rotation and colour.

Figure 1.1 shows an example of each of the four possible classes of objects. Each class can be described as follows:

- **Octopus**. The 'main' character in the video - the octopus is the only non-static character and its properties change due to its actions. Each frame contains at most one octopus. The initial frame always contains a red octopus with a randomly assigned rotation.
- **Fish**. Fish are always silver, but can have any rotation. When the octopus comes close to the fish, the fish disappears (gets eaten).
- **Bag**. Similarly to fish, plastic bags are always white but can take any rotation. Bags are harmful to the octopus, so both objects disappear when close.
- **Rock**. Rocks can have four colours: brown, blue, purple and green, but always face upright. When an octopus comes near a rock the octopus' colour will change (if necessary) to match that of the rock (it will be camouflaged).

Objects in each frame are always enclosed by a rectangular box. Object positions are given as $(x1, y1, x2, y2)$, where $(x1, y1)$ is the top left corner of the object, and $(x2, y2)$ is the bottom right[2].

[2]The y (vertical) direction is downward increasing.

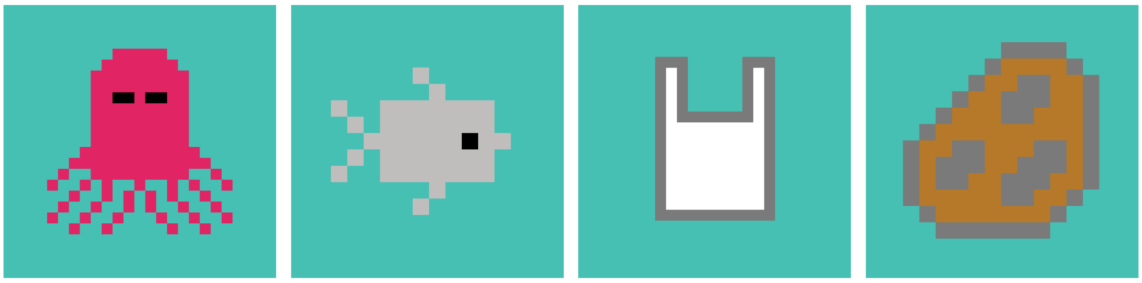


Figure 1.1: Examples of each object type in the videos (not to scale).

The dataset models a single binary relation between objects, *close*. Internally this relation is defined as: object A is close to object B if, after expanding A by 5 pixels on each side, for each pair of parallel edges of A (edges in the horizontal direction and edges in the vertical direction), one of the edges either overlaps with B or is fully contained within B's rectangular box. The *close* relation is symmetric. The algorithm for determining closeness is outlined in pseudocode in Algorithm 1. The dataset does not consider any relations with an arity higher than two.

Algorithm 1 Determine whether one object is close to another

```
procedure CLOSE($obj1, obj2$)
    EXPANDEDGES($obj1$)                                  ▷ Add 5 pixels to each side
    $overlapX \leftarrow obj2.x2 \geq obj1.x2$ and $obj2.x1 \leq obj1.x1$
    $overlapY \leftarrow obj2.y2 \geq obj1.y2$ and $obj2.y1 \leq obj1.y1$
    for $(x, y) \leftarrow obj2.corners$ do
        $matchX \leftarrow obj1.x1 \leq x \leq obj1.x2$ or $overlapX$
        $matchY \leftarrow obj1.y1 \leq y \leq obj1.y2$ or $overlapY$
        if $matchX$ and $matchY$ then
            return $True$
        end if
    end for
    return $False$
end procedure
```

Other than object properties and relations between objects, which encode visual and spatial information from a single frame, the other key data from the video are the events, which encode temporal information from the frames. Events occur between two consecutive frames, and each timestep can contain multiple (or no) events. Each video therefore contains 31 sets of events.

We choose to split events into two disjoint sets: actions and effects. Actions, also known as primitive events, can be thought of as motions that an object can make to alter its state (an object's class, position, rotation and colour). We consider three such actions: move, rotate clockwise and rotate anticlockwise. Rotations have the intuitive effect on an object's rotation. Move causes the object to move 15 pixels in the direction of its rotation. For example, an object at position $(20, 100, 30, 110)$ with a 'right-facing' rotation will be at position $(35, 100, 45, 110)$ after moving.

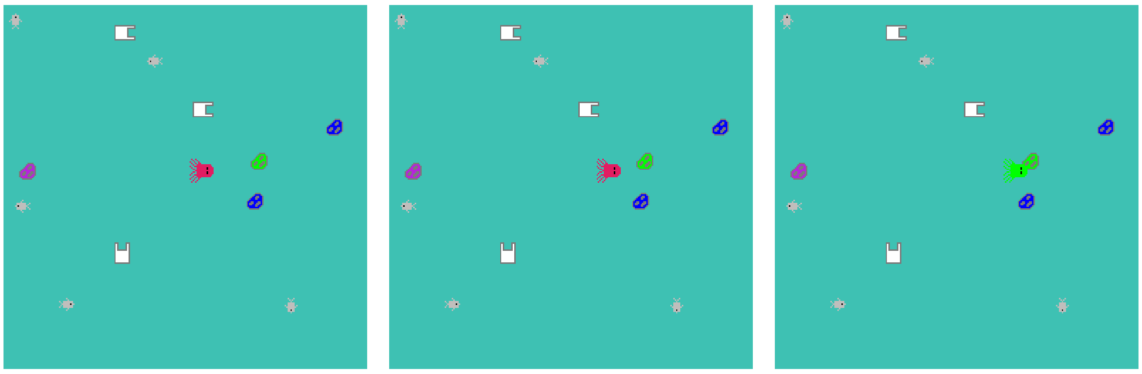


Figure 1.2: An example of an octopus moving close to a green rock and turning green.

Effects, on the other hand, happen as a consequence of some frame state being true; they are triggered by a particular frame state. Hence, effects are also known as triggered events. The dataset contains three effects: change colour, eat a fish and eat a bag. The octopus is the only object which can change colour, and this only occurs when the octopus is close to a rock. As described above, the octopus takes the colour of the rock. The octopus then continues to keep this new colour, even after moving away from the rock. Figure 1.2 shows a snippet from a video where an octopus moves close to a rock and changes colour. A fish or bag is eaten when an octopus moves close, this causes the fish or bag to disappear. However, unlike a fish, when a bag is eaten the octopus also disappears. Examples of all events can be found in Appendix **??**.

To create a video, we first create an initial frame by randomly sampling the number of each object from a set of uniform distributions: $\mathcal{U}(5, 8)$ for fish, $\mathcal{U}(2, 3)$ for bags and $\mathcal{U}(3, 4)$ for rocks, where $\mathcal{U}(l, u)$ refers to a discrete uniform distribution with lower and upper bounds l and u, respectively. Where colours and rotations need to be chosen for objects, these are sampled uniformly from the set of colours and/or rotations that are possible for that type of object. To create the rest of the video, actions for the octopus are randomly sampled for each timestep. We then work out which effects have occurred in each frame and update the properties of each object accordingly. The octopus has a 0.1 probability of choosing to rotate at each timestep. However, if the octopus chooses to move but moving would cause the octopus to be outside the frame, the octopus will rotate instead. Clockwise and anticlockwise rotations are sampled uniformly.

1.2 Questions and Answers

As well as videos, VideoQA datasets must also contain question-answer pairs (QA pairs). Answers to questions 'label' part of the video. For example, if a question asks the model to find the event which occurs between two frames, the answer to that question labels the event. However, any event not associated with a QA pair is unlabelled. The shortage of labels is not specific to events; there can be as many as 512 object instances in a single video, but perhaps only a single QA pair will label an object with a property value. Relations between objects face the same problem. For this reason, VideoQA datasets, unlike many other supervised learning datasets, can contain a lot of sparsely labelled data. On one hand, this may be beneficial to the model, since it only needs to understand the parts of the video which are mentioned in the questions, but, on the other, it can hinder the training of the model, since there is less data than would otherwise be available. Chapter 4 offers one solution to this problem.

The OceanQA dataset contains ten QA pairs per video, sampled randomly from seven question-types. The remainder of this section introduces each question type separately, along with a grammar for each question and answer presented in extended Backus-Naur form (EBNF). Since Natural Language Processing (NLP) is not the focus of this project, we choose to use a small, discrete set of structured question templates, which allow the model to easily extract relevant symbolic information from the questions. Allowing free-form natural langauge questions creates additional complexity and uncertainty for the model, which we felt would distract from the core focus on hybrid machine learning for VideoQA. Generating these free-form questions can also be very time and resource intensive. A further discussion on possible future work on hybrid models for VideoQA tasks which use free-form questions and answers is contained in Chapter **??**.

Equation 1.1 formalises the grammar of the dataset's objects, relations and events, which was implicitly described in Section 1.1. These EBNF grammar rules are used in the rules for the questions and answers.

$$
\begin{aligned}
<\!\textit{object}\!> &::= [<\!\textit{property\_value}\!>]\ <\!\textit{class}\!> \\
<\!\textit{property\_value}\!> &::= <\!\textit{rotation\_value}\!>\ |\ <\!\textit{colour\_value}\!> \\
<\!\textit{rotation\_value}\!> &::= \text{upward-facing}\ |\ \text{right-facing}\ |\ \text{downward-facing}\ |\ \text{left-facing} \\
<\!\textit{colour\_value}\!> &::= \text{red}\ |\ \text{blue}\ |\ \text{purple}\ |\ \text{brown}\ |\ \text{green}\ |\ \text{silver}\ |\ \text{white} \\
<\!\textit{class}\!> &::= \text{octopus}\ |\ \text{fish}\ |\ \text{bag}\ |\ \text{rock} \\
<\!\textit{property}\!> &::= \text{rotation}\ |\ \text{colour} \\
<\!\textit{relation}\!> &::= \text{close} \\
<\!\textit{event}\!> &::= <\!\textit{action}\!>\ |\ <\!\textit{effect}\!> \\
<\!\textit{action}\!> &::= \text{move}\ |\ \text{rotate clockwise}\ |\ \text{rotate anti-clockwise} \\
<\!\textit{effect}\!> &::= \text{change colour}\ |\ \text{eat a fish}\ |\ \text{eat a bag} \\
<\!\textit{frame\_idx}\!> &::= 0\ |\ 1\ |\ ...\ |\ 31
\end{aligned}
\tag{1.1}
$$

As mentioned above there are seven question types. The first two of these are VQA questions, which only require the model to look at a single frame of the video. The remaining five question types require the model to reason across frames. We adapt the repetition count, repeating action and state transition questions from [4] (discussed in Chapter **??**) to the OceanQA environment. We also add two further video-specific questions: questions about actions between two frames and questions about changing property values. The full set of question and answer templates is as follows:

1. Property questions are designed to test a model's understanding of object properties. In the training data 62% of the questions ask about colour, while 38% ask about rotation. All property values are represented in the training data, but not uniformly; some property values are very scarce. This reflects the underlying scarcity of objects with particular property values in the dataset; green octopuses, for example, are quite rare, while silver fish are very common. The EBNF grammar for property questions and answers is as follows:

   <*q_type_1*> ::= What <*property*> was the <*object*> in frame <*frame_idx*>?
   <*ans_type_1*> ::= <*property_value*>
   (1.2)

2. Relation questions test a model's understanding of binary relations between objects. These questions only require a yes-or-no answer. The answer is 'yes' for roughly 15% of these questions in the training data. This imbalance reflects the lack of instances of binary relations between objects, since we only model the *close* relation. Objects are selected randomly from a random frame of the video. The grammar for these questions and answers is as follows:

   <*q_type_2*> ::= Was the <*object*> <*relation*> to the <*object*> in frame <*frame idx*>?
   <*ans_type_2*> ::= yes | no
   (1.3)

3. Action questions ask which action occurred between two frames of a video. 'Move' actions account for around 44% of answers to these questions, while 'rotate clockwise' and 'rotate anticlockwise' account for approximately 28% of questions each. The answer to these questions will never be 'nothing'. The grammar for action questions and answers is as follows:

   <*q_type_3*> ::= Which action occurred immediately after frame <*frame_idx*>?
   <*ans_type_3*> ::= <*action*>
   (1.4)

4. Changed-property questions require the model to reason about how a property of the octopus changes from one frame to the next. The dataset guarantees that only a single (explicit) property changes immediately after <*frame_idx*>. Approximately 79% of these questions ask about the colour of the octopus, while the remaining 21% ask about the rotation. The grammar is as follows:

   <*q_type_4*> ::= What happened to the octopus immediately after <*frame_idx*>*?*
   <*ans_type_4*> ::= Its <*property*> changed from <*property_value*> to <*property_value*>
   (1.5)

5. Repetition count questions ask the model to work out how many times an event occurs in a given video. This requires the model to be able to count the number of occurences of an event. Events are sampled from a uniform distribution; if the event never occurs in the video, the answer is simply 0. Since each event can occur at most once per frame-interval, the answer is guaranteed to be between 0 and 30 (inclusive). The grammar for repetition count questions and answers is as follows:

$$\begin{aligned} \langle \textit{q\_type\_5} \rangle &::= \text{How many times does the octopus } \langle \textit{event} \rangle\text{?} \\ \langle \textit{ans\_type\_5} \rangle &::= 0 \mid 1 \mid \ldots \mid 30 \end{aligned} \tag{1.6}$$

6. Repeating action questions are similar to repetition count questions, but instead of asking the model for a number they ask the model to find the event which occurs a given number of times. Events cannot be sampled uniformly since, in a given video, multiple events may have the same count. Approximately 84% of questions are about actions, while the remaining 16% refer to effects. This imbalance is again due to the underlying scarcity of particular events in the dataset. There is a unique answer to every question. The grammar for repeating action questions and answers is as follows:

$$\begin{aligned} \langle \textit{q\_type\_6} \rangle &::= \text{What does the octopus do } \langle \textit{positive\_int} \rangle \text{ times?} \\ \langle \textit{ans\_type\_6} \rangle &::= \langle \textit{event} \rangle \\ \langle \textit{positive\_int} \rangle &::= 0 \mid 1 \mid \ldots \mid 30 \end{aligned} \tag{1.7}$$

7. State transition questions ask the model to find the action that occurs after a given event. 75% of the answers to these questions refer to the *move* action, while *rotate clockwise* and *rotate anticlockwise* are the answers to 12.5% of the questions each. In addition to asking the model to reason temporally about actions and events, these questions also require the model to understand which instance of an event is the ‘nth’ occurrence. The ‘nth time’ section of the grammar is unused if there is only one occurence of the event in the video. The grammar is as follows:

$$\begin{aligned} \langle \textit{q\_type\_7} \rangle &::= \text{What does the octopus do immediately after} \\ &\qquad\qquad \langle \textit{event\_noun} \rangle \text{ [for the } \langle \textit{nth} \rangle \text{ time]?} \\ \langle \textit{ans\_type\_7} \rangle &::= \langle \textit{action} \rangle \\ \langle \textit{event\_noun} \rangle &::= \text{rotating clockwise} \mid \text{rotating anticlockwise} \mid \\ &\qquad \text{eating a fish} \mid \text{eating a bag} \mid \text{changing colour} \\ \langle \textit{nth} \rangle &::= \text{first} \mid \text{second} \mid \text{third} \mid \text{fourth} \mid \text{fifth} \end{aligned} \tag{1.8}$$

Property, relation and action questions are intended to be used to train and test the model’s understanding of object properties, binary relations between objects and actions between two consecutive frames, respectively. Since the property and relation questions contain an <*object*>, knowledge of property values may be required to select the corresponding object from the frame in order to compute the

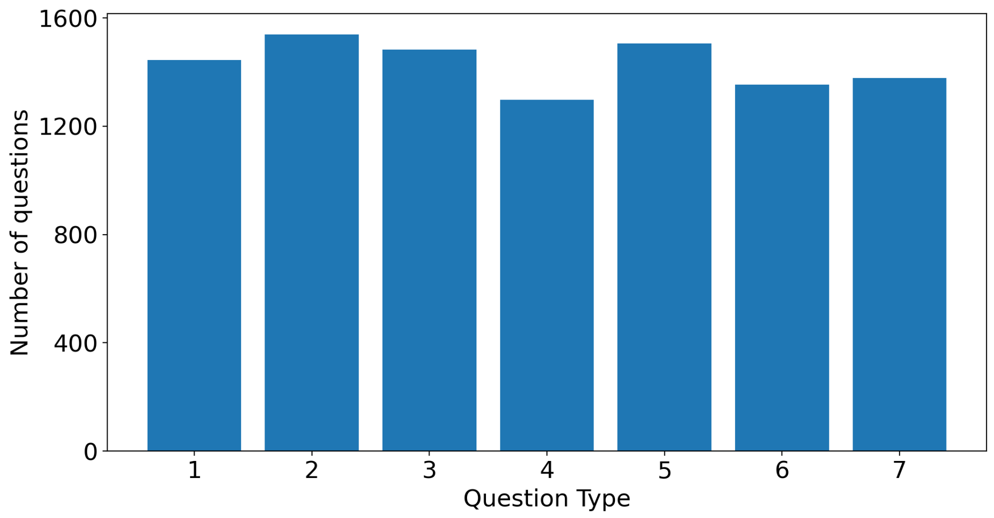


Figure 1.3: The number of QA pairs in the training data for each question type.

answer. This means it would be helpful to train the model's understanding of object properties before relations. Selecting the training data for object properties is, however, non-trivial. Chapter 4 discusses one possible solution to this problem. The final four question types are included to diversify the data the model can be trained on and to evaluate how well the model can combine its understanding of object properties, relations and events. None of the questions require any sort of external knowledge (other than the ability to count).

Questions are generated independently for each video, and questions are sampled uniformly from the seven question templates given above. This leads to an approximately uniform distribution of question types in the dataset, as shown in Figure 1.3. Since there are ten questions in each video the training, validation and testing datasets contain 10000, 2000 and 2000 questions, respectively. Appendix **??** contains a number of example images and QA pairs from the dataset.

1.3 Specification

Unlike end-to-end neural network models, hybrid models require knowledge that has been extracted from a video to be made explicit. For example, if a neural model is shown a video and asked for the colour of the octopus, it will first extract features from the video and then encode the video into a vector. This vector is combined with the encoding of the question, before a final section of the model produces either a short, free-form sentence or a probability distribution over possible answers. Relevant information from the video, including the colour of the octopus, is implicitly included in the vector encoding of the video. Hybrid models, however, usually require that this information is symbolic so that logical reasoning can be employed to find an answer to the question. For this reason it is necessary to define a specification of the video environment, which outlines the set of concepts which the model must learn in order to accurately answer the questions. This specification can also provide a way to inject background knowledge about the environment into the model.

Each environment specification must contain the following information:

1. The number of frames in each video. This project assumes this to be fixed, but relaxing this assumption would not require any major changes to the construction of the dataset or the model outlined in the rest of this report.
2. A set of pairs, $\langle class, is\_static \rangle$. Each element of the set contains the type of an object and a boolean corresponding to whether the object is capable of performing an action. All (relevant) object classes must be mentioned in this set.
3. A set of pairs, $\langle property, \{\ property\_value\ \} \rangle$, corresponding to a set of properties along with a set of their respective values. This representation for object properties assumes that each property has a discrete, finite set of values. Continuous properties are not considered in this project. Object class and position are assumed to be implicit properties since all objects must have a value for them. These implicit properties should not be listed here.
4. A set of binary relations.
5. A set of actions.
6. A set of effects of actions.

Although it has been described in detail already, Equation 1.9 provides the formal environment specification for the OceanQA dataset.

$$
\begin{aligned}
frames &::= 32 \\
objects &::= \{\langle \text{octopus},\ \text{false}\rangle, \langle \text{fish},\ \text{true}\rangle, \langle \text{bag},\ \text{true}\rangle, \langle \text{rock},\ \text{true}\rangle\} \\
properties &::= \ \{\langle \text{colour}, \{\text{red},\ \text{blue},\ \text{purple},\ \text{brown},\ \text{green},\ \text{silver},\ \text{white}\}\rangle \\
&\qquad \langle \text{rotation}, \{\text{upward-facing},\ \text{right-facing},\ \text{downward-facing},\ \text{left-facing}\}\rangle\} \\
relations &::= \{\text{close}\} \\
actions &::= \{\text{move},\ \text{rotate clockwise},\ \text{rotate anticlockwise}\} \\
effects &::= \{\text{change colour},\ \text{eat a fish},\ \text{eat a bag}\}
\end{aligned}
\tag{1.9}
$$

As alluded to already, an environment specification, like the one shown in Equation 1.9, requires that objects, properties, relations, actions and effects are all discrete. Continuous variables could be modelled by discretising, but this may lead to lower accuracy and a large number of possible values. This may have to be treated as a trade-off for using hybrid models. On the other hand, this is one of the first pieces of work which explores the use of hybrid models in VideoQA tasks; future work may be able to overcome this problem.

Chapter 2

Hybrid Property, Event and Relation Learner

We outline a novel paradigm for solving the VideoQA problem. This approach merges deep learning and logic-based machine learning and inference methods in an attempt to learn the concepts required to answer the questions. We name this approach: The Hybrid Property, Event and Relation Learner (H-PERL). Section 2.1 outlines the architecture of an H-PERL model, while Section 2.2 discusses the implementation of the H-PERL components which are common to all models presented in the subsequent chapters.

2.1 H-PERL Models

H-PERL is a generic, pipelined architecture for VideoQA tasks. The pipeline is composed of a number of components which, when strung together, form a model. A model is, therefore, a set of component implementations, and can be thought of as an 'instance' of H-PERL. Chapters 3 and 4 each outline an H-PERL model for the OceanQA dataset.

2.1.1 Architecture

The H-PERL architecture assumes that all of the information in an environment which is required to answer the questions can be modeled using: objects; binary relations between objects; and events (which occur due to an object) between two consecutive frames in the video. For many environments, simple environments (like our OceanQA dataset) in particular, this assumption holds. However, for many VideoQA datasets, particularly those set in the real-world, this assumption may not be suitable. For example, extracting some objects from a video may be non-trivial (as in the case of abstract nouns), and yet information on these objects may still be required to answer the questions. Additionally, H-PERL does not consider relations between objects with an arity larger than 2.

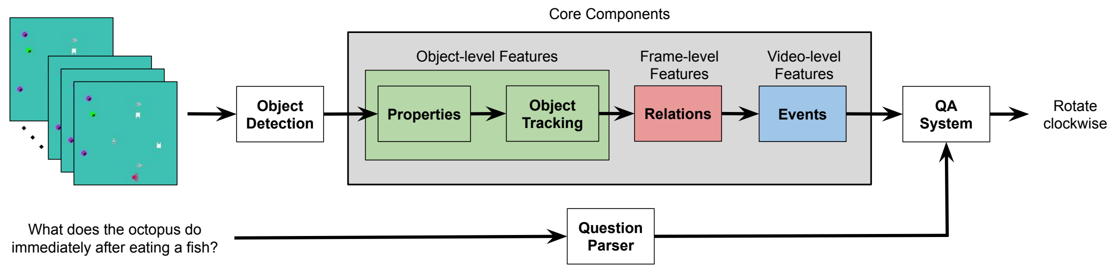


Figure 2.1: An overview of the H-PERL architecture for VideoQA. Green, red and blue shading indicate components which extract features at different levels of abstraction. Grey shading indicates the 'core' components of the architecture.

Figure 2.1 shows the components involved in a typical H-PERL model. During evaluation, information from the video and the question flow, from left to right, through the pipeline. Each H-PERL model assumes that the object detection, question parsing and QA system components are "pre-made" (either pre-trained or manually engineered). We refer to these as 'non-core' components. H-PERL allows the remaining 'core' components to be updated as the model is trained, although they don't necessarily have to be. Each core component in the pipeline accumulates information. This means that each component guarantees that existing information (or features of the data) will not be overwritten (with the small exception of the event component when error correction is used, discussed further in Chapter 3). As shown in Figure 2.1 components in the pipeline work at different levels of abstraction; the object properties and tracking components work to extract object-level features, while the relations and events components work to extract frame and video-level features, respectively.

The following is a high-level description of the tasks each component is required to complete for the H-PERL architecture to work with high accuracy:

1. **Question Parser**. The QA parsing component is used to extract relevant pieces of information from the questions (and answers when training). For example, given the question: "What does the octopus do immediately after eating a fish?" and that the question is a state-transition question, the parsing component would extract, firstly, that the object in question was an octopus, and secondly, that the event was 'eat a fish'. The parsing component, therefore, bridges the gap between the symbolic data, which the model works with, and the natural language questions and answers. When an H-PERL model is being evaluated, only the question needs to be parsed. However, when the model is training this component acts as a question-and-answer parser, since the model requires that the feedback that comes from the answer is also in symbolic form.

2. **Object Detector**. The detection component produces bounding boxes and classes for each object in each frame of the video. Any object not detected at this stage of the pipeline is assumed to be part of the background and is therefore ignored by the rest of the model.

3. **Property Extractor**. Given a set of images of objects from the detection component, the property extraction component assigns a value to every property listed in the environment specification for every object in the set.

4. **Object Tracker**. The object tracker is required to assign an identifier to each object in a given video. The object identifiers assigned in the initial frame of the video can be arbitrary, but then an algorithm is usually applied inductively to each remaining frame of the video in an attempt to assign each object the same identifier as it was given in the previous frame.

5. **Relation Classifier**. The job of the binary relation classifier is, given a symbolic representation of a video, to list all of the instances of binary relations between objects in the video. The set of possible binary relation is defined in the dataset's environment specification.

6. **Event Detector**. The event detection component produces a set of events for each pair of consecutive frames in the video. Each set of events can contain both actions and effects, and each event consists of an event name and an object identifier which signifies the non-static object that took part in that event.

7. **QA System**. The job of the QA system is to take all of the features which have been accumulated by previous components in the pipeline, along with a parsed question, and produce an answer to the question. The QA system is also given the question type, this allows it to make sense of the parsed question and to apply different reasoning for each question type.

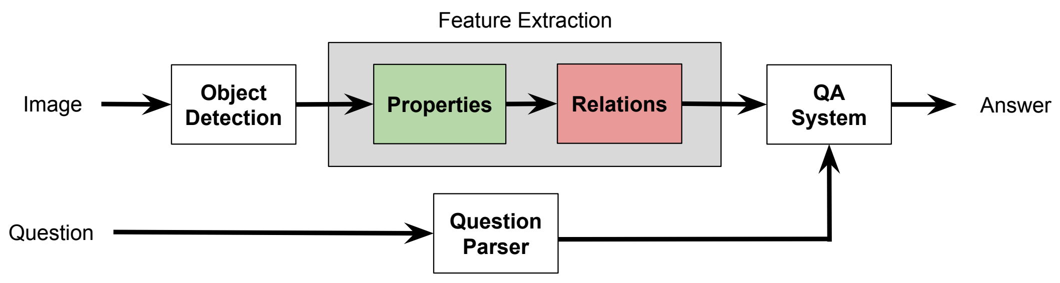
While VideoQA tasks may require the pipeline to contain the above set of components, the H-PERL architecture can also be modified for solving related problems. For example, by removing the tracking and events components and keeping the rest of the pipeline the same, a modified version of the H-PERL architecture could be used to solve VQA (also known as FrameQA) tasks. In fact, the architecture could be modified to work with many different input types, including images, videos, text or speech, provided that the data contains something akin to an 'object', and that extracting these objects (and features from these objects) is feasible. We could also swap out the final component in the pipeline, the QA system, for another task-specific component. For example, we could create a video (or image) captioning architecture by using a captioning component rather than a QA component. Figure 2.2 shows a number of architectures similar to H-PERL that have been modified for other tasks.

At a high level, then, the H-PERL approach consists of three main stages:

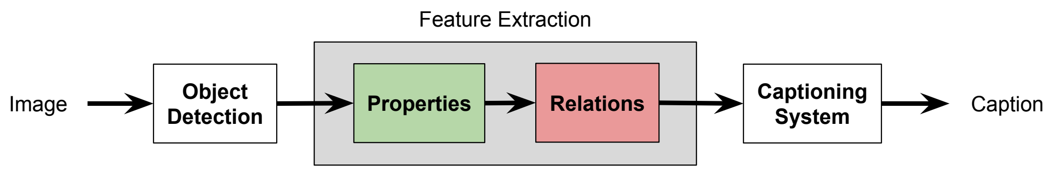
1. Attention restriction

2. Feature extraction

3. Task-specific output generation



(a) An architecture for a VQA task.



(b) An architecture for an image captioning task.

Figure 2.2: Potential modifications to the H-PERL architecture for solving other tasks. Grey shading indicates feature extraction components. Green and red shading indicate components which extract object-level and frame-level features, respectively.

These stages are similar to some approaches used in VQA tasks [9, 10], where the model's attention is firstly restricted to specific parts of the input (in this case objects), before features are extracted from the input and an output is generated. However, unlike other approaches, H-PERL does not dynamically update the model's attention based on the input question. One fundamental property of our approach which differentiates H-PERL from other question-answering models is that, in H-PERL, extracted features are stored symbolically. Symbolic features give the observer a better understanding of what the model is learning and allows the injection of background or commonsense knowledge directly into the model.

2.1.2 Information Representation

The first component in the H-PERL pipeline, the object detector, takes a raw video as input and produces a set of bounding boxes (object positions) and object classes for each frame of the video. After object detection has been applied, data about the video is stored symbolically for each object in each frame of the video. Each of the subsequent components in the pipeline can access this symbolic data along with the raw image of each object (the raw video frames are discarded). Each of these components then accumulates symbolic information about the objects, frames or video. This is what is meant by extracting object, frame and video-level features.

Once the symbolic features have been extracted from the input, each of the components in the architecture need an agreed upon way of representing the information. This symbolic representation is as follows:

- For an object with identifier <*id*> in frame <*frame*>, the object's properties, rotation and class (all referred to as <*property*>), each with value[1]<*value*>, are represented as follows:

$$\text{obs}(<\textit{property}>(<\textit{value}>, <\textit{id}>), <\textit{frame}>) \tag{2.1}$$

- For two objects in frame <*frame*>, with identifiers <*id1*> and <*id2*>, a binary relation, <*relation*>, between the objects is represented as follows:

$$\text{obs}(<\textit{relation}>(<\textit{id1}>, <\textit{id2}>), <\textit{frame}>) \tag{2.2}$$

- An event which occurs immediately after frame <*frame*>, due to an object with identifier <*id*>, is represented by one of the following rules, depending on whether the event was an action or effect:

$$\text{occurs\_action}(<\textit{action}>(<\textit{id}>), <\textit{frame}>) \tag{2.3}$$
$$\text{occurs\_effect}(<\textit{effect}>(<\textit{id}>), <\textit{frame}>) \tag{2.4}$$

Using these representations an entire video can be encoded into a logic program. The final component in the pipeline, the QA system, takes as input a video encoding and a set of parsed questions, and constructs a set of ASP rules which are used, along with the video encoding, to find an answer to the question.

2.1.3 Requirements

To give a complete view of what is needed to construct an H-PERL model, we outline the minimum set of requirements and a number of assumptions. Each instance of an H-PERL model may require additional constraints to be applied on top of these. The set of requirements is as follows:

1. A set of pre-made, non-core components.
2. A VideoQA dataset, where each element is of the form:

$$\langle \text{video}, \{\langle \text{question}, \text{answer}, \text{question type}\rangle\}\rangle$$

3. Environment specification for the given dataset.
4. (Optionally) Background knowledge of the environment, written in ASP.

[1]As described in Chapter 1, the value of an object's position is given as $(x1, y1, x2, y2)$, where $(x1, y1)$ is the top left corner of the object, and $(x2, y2)$ is the bottom right.

H-PERL also requires that the following assumptions be made about the data:

1. All relevant information in the video can be modelled by object properties; binary relations between objects; and events occuring between consecutive frames of the video.

2. Each of the properties, relations and events components can be trained individually and directly using a specific type of question. For example, for the OceanQA dataset, we can use question types 1, 2 and 3 to train the properties, relations and events components, respectively.

3. As mentioned in Chapter 1, properties, relations and events must be discrete. There is also currently no way of modelling continuous variables in the data; for example, we cannot say that the octopus rotated clockwise by $\frac{1}{4}\pi$ radians.

These requirements and assumptions clarify some of the limitations of H-PERL models. Firstly, for some environments pre-trained object detectors (or data to train them with) may be difficult to find. Secondly, it may also be difficult, if not impossible, to construct or train a question parser for free-form, natural language question-answering datasets. Additionally, most QA datasets are not guaranteed to contain questions which can be used to directly train components of the model. Finally, many environments will simply be too complex to be accurately modelled by discrete properties, relations and events. These requirements and assumptions do, however, provide initial directions for future research in the area of Hybrid question-answering models. We discuss some potential extensions to H-PERL in Chapter **??**.

As well as drawbacks, the H-PERL architecture does also provide a number of advantages. One of the most important advantages of hybrid models is the ability to encode commonsense knowledge or background knowledge of the environment directly into the model without having to learn it. Since the model accumulates video information symbolically, we can inject background knowledge at any stage of the pipeline. A number of further advantages of hybrid models are presented in the subsequent chapters, and these, along with the disadvantages, are summarised in Chapter **??**.

2.2 Common Components

The two models, one hardcoded and one trained, which are outlined in subsequent chapters, contain a number of common components. Before describing the implementation of these models, we first outline the details of the components common to both. These components include the non-core components, which the H-PERL pipeline assumes are pre-made, as well as the object tracker, which uses a similar algorithm for both the hardcoded and trained models.

2.2.1 Question Parser

As mentioned previously, the role of the question parsing component is to translate the natual language questions into a set of keywords (and their types). This extracted information is then used by the QA system to construct ASP rules with which the answers to the questions are found. During training, the question parser acts as a question-and-answer parser, as the model needs to collect training data from the answer.

Both the hardcoded and the trained H-PERL models use a very simple hand-engineered question parser. Since the question parser is told the type of the question it has been given, and since the OceanQA dataset uses templated questions, the question parser can simply extract the relevant parts of the question by looking at the corresponding template. This is implemented by splitting the question into a list of words and simply picking the correct word based on the template.

Take the question "What does the octopus do immediately after rotating clockwise for the second time?" as an example. Based on the question template shown in Chapter 1, the parsing component would extract the following from the question:

$$\begin{aligned} \textit{object} &= \text{octopus} \\ \textit{action} &= \text{rotate clockwise} \\ \textit{occurrence} &= 2 \end{aligned} \tag{2.5}$$

Note that the action noun 'rotating clockwise' has been converted to the verb form, as given in the environment specification. In addition, the occurrence, 'second', is converted to a number, which is simpler for the QA system to work with. These conversions are all hand-engineered, as opposed to learnt.

The parsing of all other question types is done in the exactly same way. For the sake of brevity these are not shown in this discussion. The discussion on the QA system in Section 2.2.4 does, however, give more detail. As a general rule, the information extracted from the questions is the parts of the question templates which are contained in <> brackets. For question types 4, 5, 6 and 7 the parser must also extract the *object* keyword, which is guaranteed to be octopus in all four of these question types, since it is the only object which can perform actions.

2.2.2 Object Detector

Chapter **??** describes two object detection algorithms: Faster R-CNN and YOLO. Faster R-CNN has been shown to be the more accurate of the two [6]. YOLO's key advantage, however, is its speed; it has been shown to be capable of processing five times as many frames per second as Faster R-CNN on the Pascal VOC dataset [7]. However, since OceanQA frames are fairly small and simple, and since we don't need real-time performance for VideoQA tasks, we prefer accuracy over speed and therefore choose Faster R-CNN over YOLO for the object detection component.

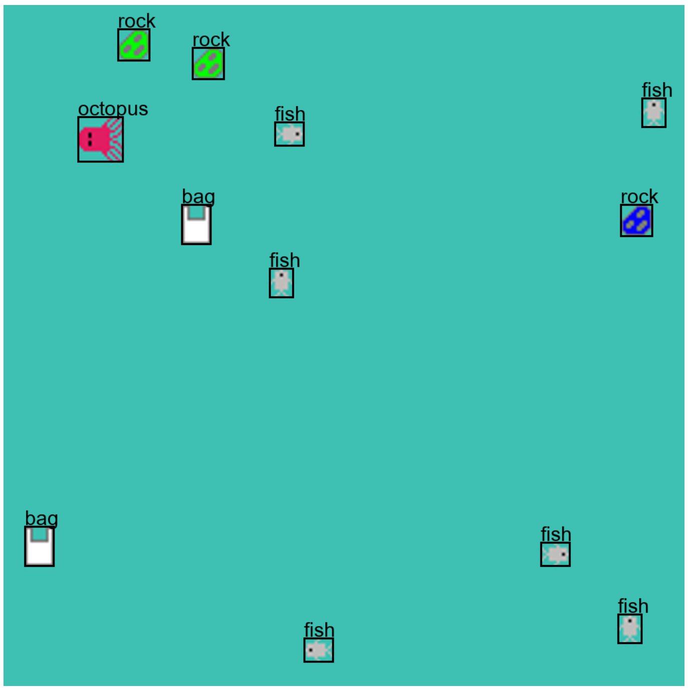


Figure 2.3: A visualisation of the output of the detector on an example frame.

The object detection network uses a pre-built Faster R-CNN model from the *Torchvision*[2] library. The network is trained on the full-data OceanQA dataset (where every object in every frame is fully labelled, as outlined in Chapter 1), using Faster R-CNN's multi-task loss function [8]. The *PyCOCOTools*[3] library is used to implement the core training functionality.

As is standard in Faster R-CNN implementations, we provide the detection model with a feature extraction network. This network's role is to extract features from the input frames and pass these to the pre-built detection network. We provide a three-layer convolutional network, described in Table 2.1, with randomly initialised weights. These weights, along with the weights of the detection network, are updated as the whole model is trained.

An example visualisation of the detector's output is shown in Figure 2.3. The full details of the object detector's performance are given in Chapter **??**.

Layer	In Channels	Out Channels	Kernel Size	Stride	Padding
Conv1	3	32	3x3	1	1
Conv2	32	64	3x3	2	1
Conv3	64	128	3x3	2	1

Table 2.1: Specification of the feature extraction network. Batch normalisation is applied between layers Conv1 and Conv2, and between layers Conv2 and Conv3.

[2]Available at: https://github.com/pytorch/vision
[3]Available at: https://github.com/cocodataset/cocoapi

2.2.3 Object Tracker

The object tracker's task is to assign every object in the video with an identifier. Ideally, an object is assigned the same identifier throughout the video, however, errors in the object detection can make this difficult. The tracker is given a video containing a list of frames, with each frame containing a set of detected objects (each object contains just a class and a position at this stage).

We implement a simple object tracker which assigns integer identifiers, starting from zero, to each object in the initial frame of the video. The tracker then inductively applies an algorithm (discussed below) which attempts to match objects in the previous frame (we call these previous objects), which have been assigned an identifier, with objects in the next frame (next objects), which are unidentified. An object, $obj1$, matches with another object, $obj2$, if the following two conditions are met:

1. $obj1$ and $obj2$ have the same class.
2. The Euclidean distance between the top left corner of $obj1$ and the top left corner of $obj2$ is no more than 30 pixels.

Both the distance metric and the maximum distance value are considered hyperparameters of the tracking component. Many other values for these hyperparameters would work equally well with the OceanQA dataset, and if these values weren't known in advance, they could be found fairly easily through a combination of analysis of the dataset (the maximum distance must be at least 15 pixels) and trial-and-error.

After assigning identifiers to objects in the initial frame, the tracker iteratively applies the following algorithm to each subsequent frame:

1. Each object in the new frame votes for the object in the previous frame which matches with it best.
2. Each previous object collates all of the new objects which have voted for it and chooses the new object with which it best matches.
3. If a previous object does not get any votes, it is because there are no new objects which consider this object their best match. We therefore assume the previous object has disappeared. Each disappeared object is added to a *hidden objects* list, so that if the object reappeared in a subsequent frame the same identifier could be assigned.
4. Each remaining previous object is now matched with a single new object. The new object is assigned the identifier of the previous object it is matched with.

5. Some new objects - namely those which were rejected by a previous object in favour of another new object - remain unidentified. Since all previous objects have either passed on their identifier or disappeared, these remaining objects must be new. For each of these new objects, if it cannot be matched with an object in the *hidden objects* list, a previously unused identifier is created and assigned to the object, otherwise the object is assigned its best match in the *hidden objects* list.

6. Each object in the *hidden objects* list can only stay for a maximum of five frames from when they are added (although this duration is also considered a hyperparameter and could therefore be altered). The final step of the algorithm is to remove from this list any hidden object which has overstayed its welcome.

This simple, heuristic-based tracker works well for simple, well-defined environments such as OceanQA. In fact, Chapter **??** shows that the tracker can achieve perfect performance, assuming the object detection is completely accurate. However, this tracker may perform poorly in more complex environments, such as real-world VideoQA datasets. Other, more advanced object tracking algorithms may fare better, however. The Simple Online and Realtime Tracking (SORT) [1] algorithm is one of the best performers [2]. Since the H-PERL architecture is completely modular, our tracking algorithm could simply be swapped out for a more advanced alternative when the environment required it.

2.2.4 QA System

As shown in Section 2.1, the information extracted from the video by the core components of the H-PERL architecture is stored symbolically. The role of the QA system is then, given these symbolic features and a set of parsed questions and their respective types, to find an answer to each question.

Since the features are stored as ASP predicates, a simple implementation of this component is to construct a set of ASP rules for each question type, and use these rules to find the questions' answers. To construct these rules we use a set of templates, one for each question type. After a question has been parsed, we fill in the question's corresponding template using information extracted by the question parsing component.

Since there are ten questions for each video, the QA system can save time by answering all ten questions in the same ASP program. To do this a set of ASP question-answering rules is generated for each question. Each set of rules contains at least one rule with an $answer$ predicate in the head. The answer predicate contains at least two arguments; the first of these is the question number, and any remaining arguments store the answer to the question. For example, the following is the answer predicate for property questions:

$$answer(<\text{question}>, <\text{property}>, <\text{value}>)$$

After the ASP program has been run, the answer predicates are collated. If there are multiple answers for a single question, a single answer is chosen at random. After this, hardcoded rules generate an answer string using the arguments of the $answer$ predicate, for each question. These answer strings are considered the output of the H-PERL model, and are therefore compared against the expected answer string when evaluating a model's performance.

Before describing the ASP rule templates, we provide a definition of an ASP object descriptor. This descriptor is generated using information extracted from the <*object*> part of the question. As a reminder, the <*object*> grammar rule contains information on an object's class and, optionally, a property value. For example, 'silver fish', 'upward-facing rock' and 'octopus' are all valid instances of the <*object*> EBNF rule. The ASP object descriptor is a set of predicates used in the body of a rule which locate the object mentioned in the question. For object identifier <*id*>, frame <*frame*> and an instance of <*object*> with class <*class*> and property value <*value*>, the ASP object descriptor is as follows:

$$obs(class(<\textit{class}>, <\textit{id}>), <\textit{frame}>), [obs(<\textit{property}>(<\textit{value}>, <\textit{id}>), <\textit{frame}>)] \tag{2.6}$$

If <*object*> does not contain a property value, the property predicate in 2.6 is omitted. In order to simplify the ASP rule templates, which follow, we use the following AspObj function, which constructs the set of predicates shown in 2.6 to be used in the body of ASP rules:

$$\textsc{AspObj}(<\textit{object}>, <\textit{id}>, <\textit{frame}>)$$

We also outline a number of ASP helper rules whose head predicates are used in the body of the question-answering rules. For brevity the full helper rules are not shown, however, the descriptions of the head predicates are as follows:

1. If $changed(<\textit{property}>, <\textit{before}>, <\textit{after}>, <\textit{id}>, <\textit{frame}>)$ is true, then the property <*property*> of the object with identifer <*id*> takes value <*before*> in frame <*frame*> and value <*after*> in frame <*frame*>+1.
2. The $event\_count(<\textit{event}>, <\textit{id}>, <\textit{num}>)$ predicate means that <*num*> is the number of times an object with identifer <*id*> causes event <*event*>.
3. If $event\_occurrence(<\textit{event}>, <\textit{id}>, <\textit{frame}>, <\textit{num}>)$ is true, then <*num*> is the number of times an object with identifer <*id*> causes event <*event*> during or before frame <*frame*>.
4. The $exists(<\textit{id}>, <\textit{frame}>)$ predicate means that the object with identifier <*id*> exists in frame <*frame*>.

In the following, we show, firstly, the question template (as already outlined in Chapter 1), secondly, how the ASP rule is constucted from the information in the parsed question and, finally, how the answer string is generated from the $answer$ predicate, for each type of question:

1. Property questions have the following template:

   What <*property*> was the <*object*> in frame <*frame*>?

   Following question parsing, the ASP rule for question number <*question*> is constructed as follows:

   $$\begin{aligned} answer(\text{<question>}, \text{<property>}, Val) \text{ :- } & \textsc{AspObj}(\text{<object>}, Id, \text{<frame>}), \\ & obs(\text{<property>}(Val, Id), \text{<frame>}). \end{aligned} \tag{2.7}$$

   Then the answer string is simply the Val argument to the $answer$ predicate.

2. Relation questions have the following template:

   Was the <*object*> <*relation*> to the <*object*> in frame <*frame*>?

   The ASP rule for answering these questions is generated as follows:

   $$\begin{aligned} related(\text{<question>}) \text{ :- } & holds(\text{<relation>}(Id1, Id2), \text{<frame>}), \\ & \textsc{AspObj}(\text{<object>}, Id1, \text{<frame>}), \\ & \textsc{AspObj}(\text{<object>}, Id2, \text{<frame>}). \\ answer(\text{<question>}, yes) \text{ :- } & related(\text{<question>}). \\ answer(\text{<question>}, no) \text{ :- } & not\ related(\text{<question>}). \end{aligned} \tag{2.8}$$

   The yes or no answer string is then taken from the $answer$ predicate.

3. Action questions have the following form:

   Which action occurred immediately after frame <*frame*>?

   We create an ASP rule for each action, including the ‘nothing’ action. The outline of each rule is as follows:

   $$answer(\text{<question>}, \text{<action>}) \text{ :- } occurs(\text{<action>}(Id), \text{<frame>}). \tag{2.9}$$

   The answer string is then simply the value of <*action*>.

4. Changed-property questions take the following form:

   What happened to the octopus immediately after <*frame*>?

   The ASP rule for answering this question is as follows:

   $$\begin{aligned} answer(\text{<question>}, Prop, Before, After) \text{ :- } & changed(Prop, Before, After, Id, \text{<frame>}), \\ & exists(Id, \text{<frame>} + 1), \\ & \textsc{AspObj}(\text{octopus}, Id, \text{<frame>}). \end{aligned} \tag{2.10}$$

   The answer string is then constructed as follows:

   Its $Prop$ changed from $Before$ to $After$

5. Repetition count questions resemble the following template:

   How many times does the octopus <*event*>?

   The ASP rule for a repetition count question with question number <*question*> is as follows:

$$\begin{aligned} answer(<\textbf{question}>, Num) \text{ :- } & event\_count(<\textbf{event}>, Id, Num), \\ & \textsc{AspObj}(\text{octopus}, Id, \_). \end{aligned} \tag{2.11}$$

   In Rule 2.11 an anonymous frame number variable is supplied to the AspObj function since we don't care which frames the octopus is in; we simply need to fetch its identifier. The answer string is then simply the value of Num.

6. Repeating action questions take the following form:

   What does the octopus do <*num*> times?

   The ASP rule for these questions is constructed as follows:

$$\begin{aligned} answer(<\textbf{question}>, Event) \text{ :- } & event\_count(Event, Id, <\textbf{num}>), \\ & \textsc{AspObj}(\text{octopus}, Id, \_). \end{aligned} \tag{2.12}$$

   The answer string is given as the value of the $Event$ variable.

7. Finally, state transition questions have the following template:

   What does the octopus do immediately after <*event_noun*> [for the <*nth*> time]?

   After <*event_noun*> is converted to event form, which we denote <*event*>, and the string <*nth*> is converted to a number, denoted <*occ*> (this is simply 1 if the occurrence string is omitted), the ASP rule can be constructed as follows:

$$\begin{aligned} answer(<\textbf{question}>, Action) \text{ :- } & action(Action), \\ & occurs\_event(Action, Id, Frame + 1), \\ & event\_occurrence(<\textbf{event}>, Id, Frame, <\textbf{occ}>), \\ & \textsc{AspObj}(\text{octopus}, Id, Frame). \end{aligned} \tag{2.13}$$

   In Rule 2.13 the type predicate, $action$, is used to ensure that the answer is an action. If this predicate is omitted from the body the answer may incorrectly be given as an effect. Once the answer has been found, the answer string is given as the value of the $Action$ variable.

Hardcoding the question-answering rules does, of course, bring a number of disadvantages; namely that the entire component must be rewritten for every new environment, and that the rules may be very complex and time consuming to write. It also introduces potential for human bias or errors in the rules, meaning that the ASP rules may not work as well as intended. However, although it is not part of this project, there is a possibility of learning these rules using Inductive Logic Programming. This would allow the component to be considerably more versatile and potentially more accurate. Chapter **??** discusses this idea further.

Chapter 3

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 **??**.

3.1 Properties

As mentioned in Chapter 2, 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 3.1 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

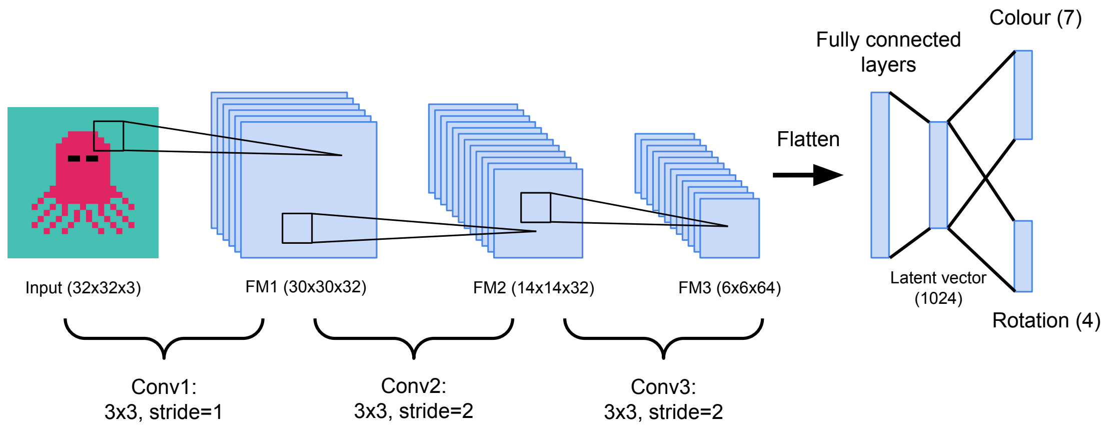


Figure 3.1: Architecture of the properties component neural network. An object is encoded into a latent vector, before a set of fully-connected layers produce a set of probability distributions over the property values. Batch normalisation is applied between convolutional layers. FM stands for feature 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:

$$\sigma(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \text{ for } i = 1, ..., K \tag{3.1}$$

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:

$$H(\mathbf{p}, \mathbf{y}) = -\sum_{c=1}^{K} y_c \log(p_c) \tag{3.2}$$

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.

3.2 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 3.2. The algorithm was discussed alongside techniques used to create the dataset in Chapter 1 and is shown in Algorithm 1.

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.

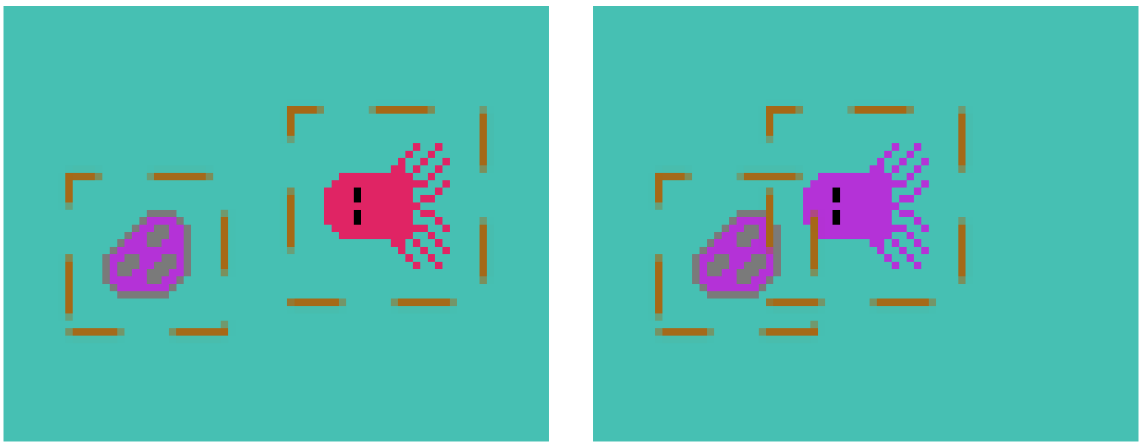


Figure 3.2: Diagrams showing the octopus before and after moving close to a purple rock, and therefore turning purple itself. The brown dashed lines show the bounding boxes expanded by 5 pixels on each side around the objects. If these boxes overlap the objects are deemed to be close to one another.

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.

3.3 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.

3.3.1 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 $i+1$.

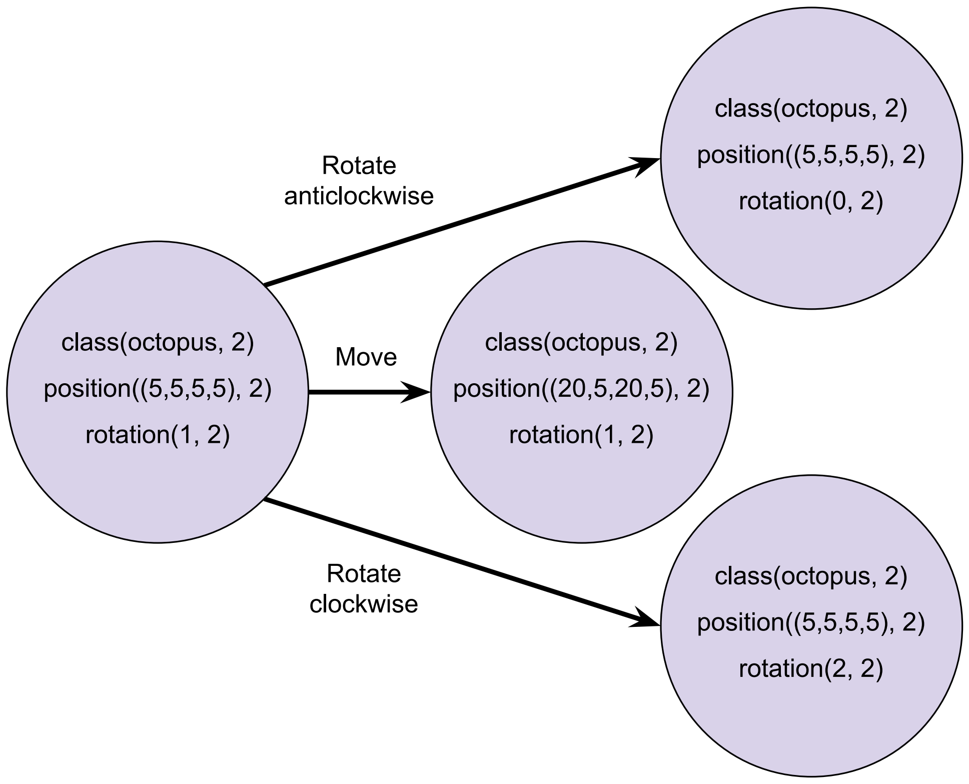


Figure 3.3: A simplified example set of $\mathcal{AL}$ states and transitions. States which model the OceanQA environment contain many more fluents than are shown here.

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 3.3.

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 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:

$$\neg holds(F, 0) \text{ :- } fluent(inertial, F), not\ holds(F, 0). \tag{3.3}$$

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:

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

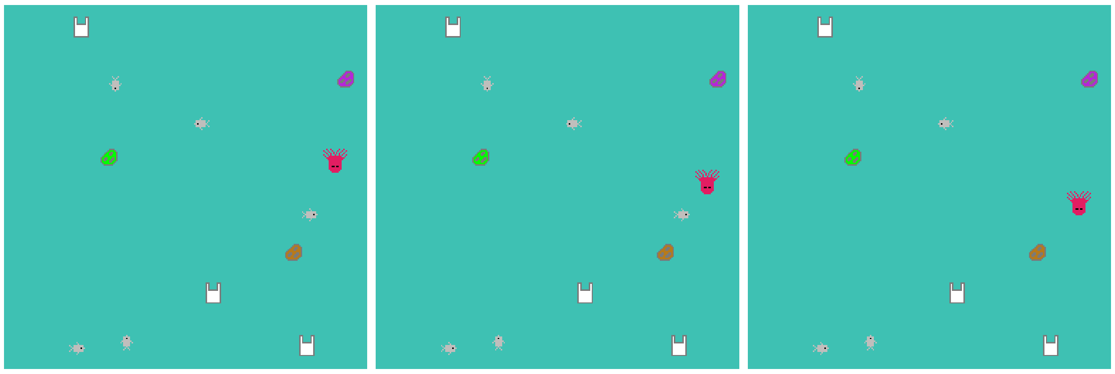


Figure 3.4: An example of the *eat a fish* effect.

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

$$\begin{aligned}occurs\_effect(eat\_a\_fish(Octo), I) \text{ :- } & occurs\_action(move(Octo), I),\\ & holds(class(fish, Fish), I),\\ & holds(disappear(Fish), I).\end{aligned} \tag{3.4}$$

$$\begin{aligned}occurs\_effect(eat\_a\_bag(Octo), I) \text{ :- } & occurs\_action(move(Octo), I),\\ & holds(class(bag, Bag), I),\\ & holds(disappear(Bag), I),\\ & holds(disappear(Octo), I).\end{aligned} \tag{3.5}$$

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:

$$\begin{aligned} \mathit{change\_colour}(\mathit{Old}, \mathit{New}, \mathit{Id}, I-1) \text{ :- } & \mathit{holds}(\mathit{class}(\mathit{octopus}, \mathit{Id}), I), \\ & \mathit{holds}(\mathit{close}(\mathit{Id}, \mathit{IdRock}), I), \\ & \mathit{holds}(\mathit{colour}(\mathit{Old}, \mathit{Id}), I-1), \\ & \mathit{holds}(\mathit{class}(\mathit{rock}, \mathit{IdRock}), I), \\ & \mathit{holds}(\mathit{colour}(\mathit{New}, \mathit{IdRock}), I), \\ & \mathit{Old} \mathrel{!}= \mathit{New}, \mathit{step}(I-1). \end{aligned} \tag{3.6}$$

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

$$\mathit{holds}(\mathit{colour}(\mathit{New}, \mathit{Id}), I+1) \text{ :- } \mathit{change\_colour}(\mathit{Old}, \mathit{New}, \mathit{Id}, I). \tag{3.7}$$

$$\neg \mathit{holds}(\mathit{colour}(\mathit{Old}, \mathit{Id}), I+1) \text{ :- } \mathit{change\_colour}(\mathit{Old}, \mathit{New}, \mathit{Id}, I). \tag{3.8}$$

Rules 3.7 and 3.8 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 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.

3.3.2 Action Optimisation

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

$$\mathit{occurs\_action}(A, I) : \mathit{action}(A) \text{ :- } \mathit{step}(I+1), I >= 0. \tag{3.9}$$

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, 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:

$$:\sim \neg \mathit{obs}(\mathit{exists}(\mathit{Id}), I), \mathit{holds}(\mathit{exists}(\mathit{Id}), I).[1@1, \mathit{exists}(\mathit{Id}), I] \tag{3.10}$$

$$:\sim \mathit{obs}(F, I), \neg \mathit{holds}(F, I).[1@2, F, I] \tag{3.11}$$

Rule 3.10 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 3.11. Rule 3.10 says that we prefer answer sets with the fewest mismatches due to the $exists$<*id*> fluent. Rule 3.11, on the other hand, says that we prefer answer sets with the fewest mismatches between all fluents. Rule 3.10 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:

$$\text{:- } obs(class(octopus, Id), I), occurs\_action(nothing(Id), I). \tag{3.12}$$
$$\text{:- } \neg obs(class(octopus, Id), I), \neg occurs\_action(nothing(Id), I). \tag{3.13}$$

The use of hard rather than weak constraints in Rules 3.12 and 3.13 improves the speed of the ASP optimisation. However, the weak constraints in Rules 3.10 and 3.11 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 [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 3.5. 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.

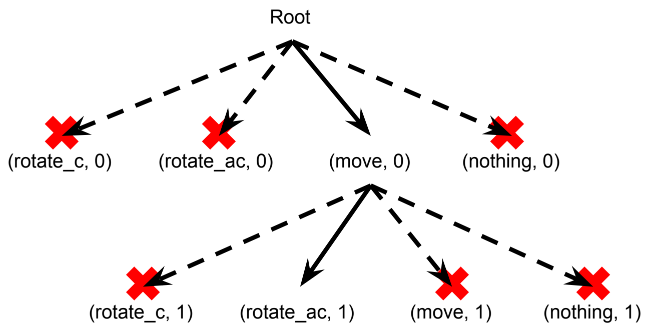


Figure 3.5: Part of an example action search tree. Each action is given as a pair of $(action\_name, timestep)$. Red crosses and dashed lines show pruned branches.

3.4 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 3.6. 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 3.3 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.
5. Finally, the chosen objects are assigned their respective identifiers, while the other objects are assigned new, unused identifiers. As mentioned in Chapter 2, 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.

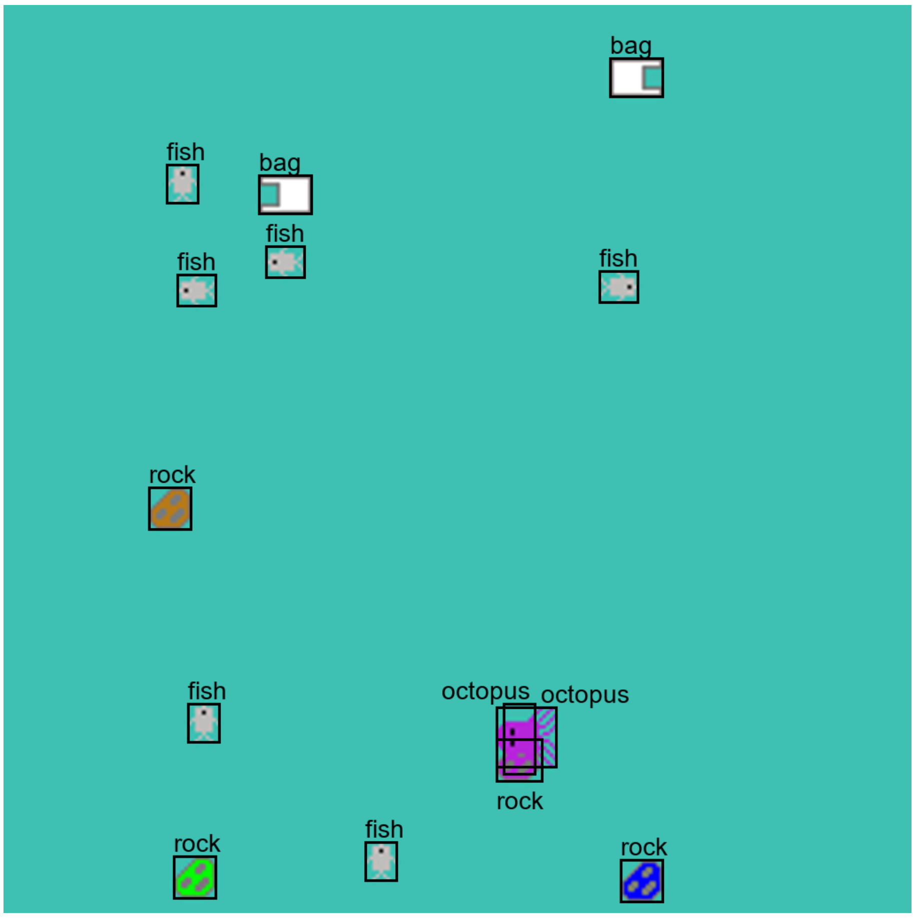


Figure 3.6: An example of an error in the object detection. Two purple octopuses are detected in this frame. This leads to competion for the identifier of the 'real' octopus.

As mentioned in Section 3.3, 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 **??**.

Chapter 4

Trained Model

Bibliography

[1] Alex Bewley et al. “Simple online and realtime tracking”. In: *2016 IEEE International Conference on Image Processing (ICIP)*. IEEE. 2016, pp. 3464–3468.

[2] Gioele Ciaparrone et al. “Deep learning in video multi-object tracking: A survey”. In: *Neurocomputing* (2019).

[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] Yunseok Jang et al. “Tgif-qa: Toward spatio-temporal reasoning in visual question answering”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2017, pp. 2758–2766.

[5] Jiayuan Mao et al. “The neuro-symbolic concept learner: Interpreting scenes, words, and sentences from natural supervision”. In: *arXiv preprint arXiv:1904.12584* (2019).

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

[7] 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.

[8] 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.

[9] Kelvin Xu et al. “Show, attend and tell: Neural image caption generation with visual attention”. In: *International conference on machine learning*. 2015, pp. 2048–2057.

[10] Zichao Yang et al. “Stacked attention networks for image question answering”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 21–29.

[11] Kexin Yi et al. “Clevrer: Collision events for video representation and reasoning”. In: *arXiv preprint arXiv:1910.01442* (2019).

[12] Kexin Yi et al. “Neural-symbolic vqa: Disentangling reasoning from vision and language understanding”. In: *Advances in Neural Information Processing Systems*. 2018, pp. 1031–1042.