Chapter 1

Hybrid Property, Event and Relation Learner

We outline a novel paradigm for solving the VideoQA problem. This structure 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 structure: Hybrid Property, Event and Relation Learner (H-PERL). Section 1.1 outlines the structure of an H-PERL model, while Section 1.2 discusses the implementation of a number of H-PERL components which are common to all models presented in the subsequent chapters.

1.1 H-PERL Model

H-PERL is a generic, pipelined structure for finding an answer to a set of questions, given a video. The pipeline is composed of a number of components which, when strung together, form a model (a model can be thought of as an 'instance' of the H-PERL structure). Chapters **??** and **??** each outline an H-PERL model for the OceanQA dataset.

1.1.1 Architecture

An H-PERL pipeline 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, the H-PERL structure is not capable of modelling relations between objects with an arity larger than 2.

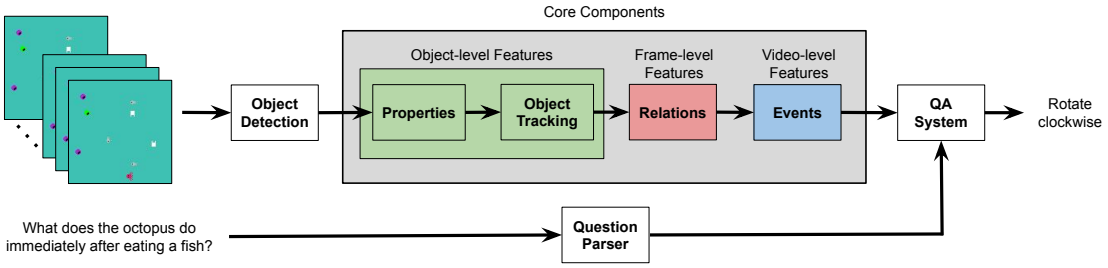


Figure 1.1: An H-PERL pipeline for VideoQA. Green, red and blue shading indicates components which work to extract features at different levels of abstraction. Grey shading indicates the 'core' components of the pipeline.

Figure 1.1 shows the components involved in a typical H-PERL pipeline. 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 **??**). As shown in Figure 1.1 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 pipeline 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 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.

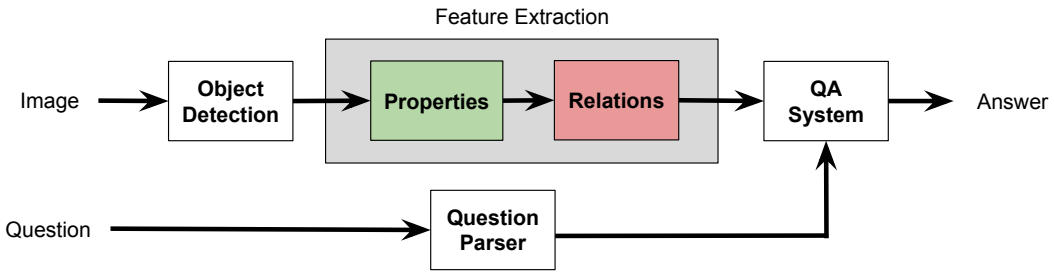
While VideoQA tasks may require the pipeline to contain the above set of components, the H-PERL structure can also be modified for solving related problems. For example, by removing the tracking and events components and keeping the rest of the pipeline the same, the H-PERL structure could be used to solve VQA (also known as FrameQA) tasks. In fact, the pipeline 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 pipeline by using a captioning component rather than a QA component. Figure 1.2 shows a number of pipelines similar to H-PERL that have been modified for other tasks.

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

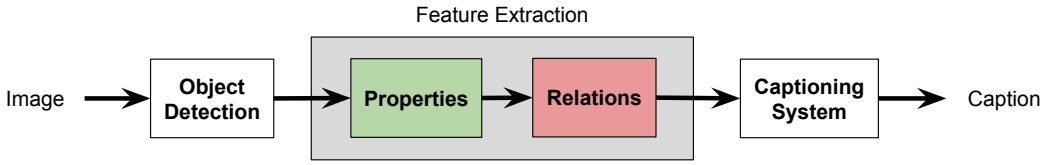
1. Attention restriction

2. Feature extraction

3. Task-specific output generation



(a) A pipeline for a VQA task.



(b) A pipeline for an image captioning task.

Figure 1.2: Potential modifications to the H-PERL pipeline 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 [1, 2], 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. Another key difference between H-PERL and other quesiton-answering approaches is that extracted features are stored symbolically. This gives the observer a better understanding of what the model is learning and allows the injection of background or commonsense knowledge directly into the model.

1.1.2 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 pipeline 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 <*value*>, are represented as follows:

$$\text{obs}(<\textit{property}>(<\textit{value}>, <\textit{id}>), <\textit{frame}>) \tag{1.1}$$

  As described in Chapter **??**, the value of an object's rotation 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.

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

$$\text{obs}(<\textit{relation}>(<\textit{id1}>, <\textit{id2}>), <\textit{frame}>) \tag{1.2}$$

- 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:

$$\text{occurs\_action}(<\textit{action}>(<\textit{id}>), <\textit{frame}>) \tag{1.3}$$

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.

1.1.3 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 (question parser, object detector and QA system).
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.

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 **??**, 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 canbe 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 **??**.

1.2 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 the a similar algorithm for both the hardcoded and trained models.

1.2.1 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 model 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 **??**, the parsing component would extract the following from the question:

$$\begin{aligned} \textit{object} &= \text{octopus} \\ \textit{action} &= \text{rotate clockwise} \\ \textit{occurrence} &= 2 \end{aligned} \tag{1.4}$$

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. 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, which in all of these questions is guaranteed to be the octopus.

1.2.2 Object Detector

1.2.3 Object Tracker

1.2.4 QA System

Bibliography

[1] 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.

[2] 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.