Chapter 1

Trained Model

In contrast to the hardcoded model, the trained H-PERL model does not use manually engineered relations or events components, and must therefore rely on using components which can be trained. The trained model also uses the QA-data version of the OceanQA dataset, rather than the full-data version which the hardcoded model was able to use to train its properties component. This means that the trained model needs to rely on the data contained in QA pairs alone to train its components.

The model is trained using *curriculum learning* - the model learns concepts incrementally. In the case of H-PERL, this means that the model is, firstly, trained to understand object properties, then trained to understand relations, and, finally, is trained to understand events. After training a component, the component is used to extract its corresponding symbolic information from the training data, which can then be used by the next component to be trained.

1.1 Properties

As mentioned in Chapter **??**, property questions in the OceanQA dataset ask the model to find a property value for a specific object. This object, however, can contain a reference to a property value. This means that, in some cases, knowledge of object properties is required in order to find the specified object in the frame. For example, if a question asked "What colour was the upward-facing fish in frame 12?", and there are three fish, each with unique rotations, in frame 12, one would need knowledge of object properties in order to select the correct image of the fish. This means the training data cannot be collated in the same way as the hardcoded model; the model needs a trained property extractor in order to find the images to train the property extractor with.

In this section we propose a solution to overcome this problem which utilises semi-supervised learning in order to label all of the objects in the dataset with property values. Once these labels have been found, the property component can be trained in the same way as the hardcoded model, outlined in Chapter **??**.

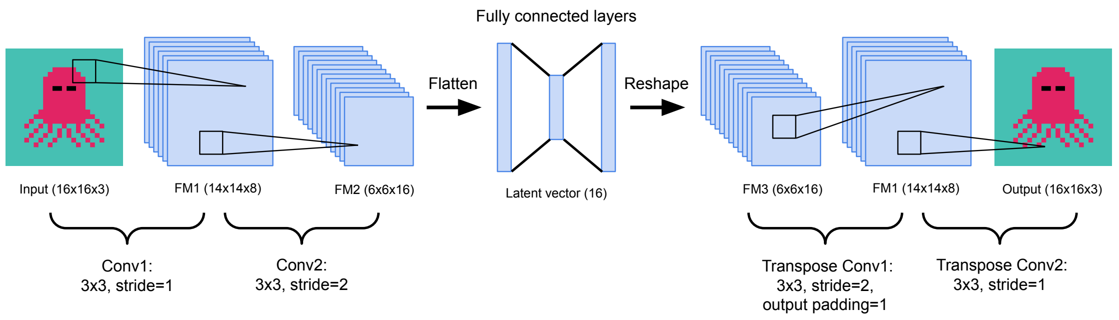


Figure 1.1: An illustration of the autoencoder architecture. Although not shown here, the network contains batch normalisation layers between each pair of convolution layers. FM stands for feature maps, and the dimensions of each feature maps are given as $(height, width, number\ of\ feature\ maps)$.

The first step of the algorithm for finding these labels involves training an autoencoder neural network to extract a 16-dimensional latent vector from each object image. This network is trained in an unsupervised manner using a sample of 40,000 of the objects detected by the object detector in the training data, where each object type is equally represented in the sample. The architecture of the network is shown in Figure 1.1. Each object image is resized to 16x16 pixels and the network is trained with a learning rate of 0.001 for 5 epochs with a mean-absolute error (MAE) loss function.

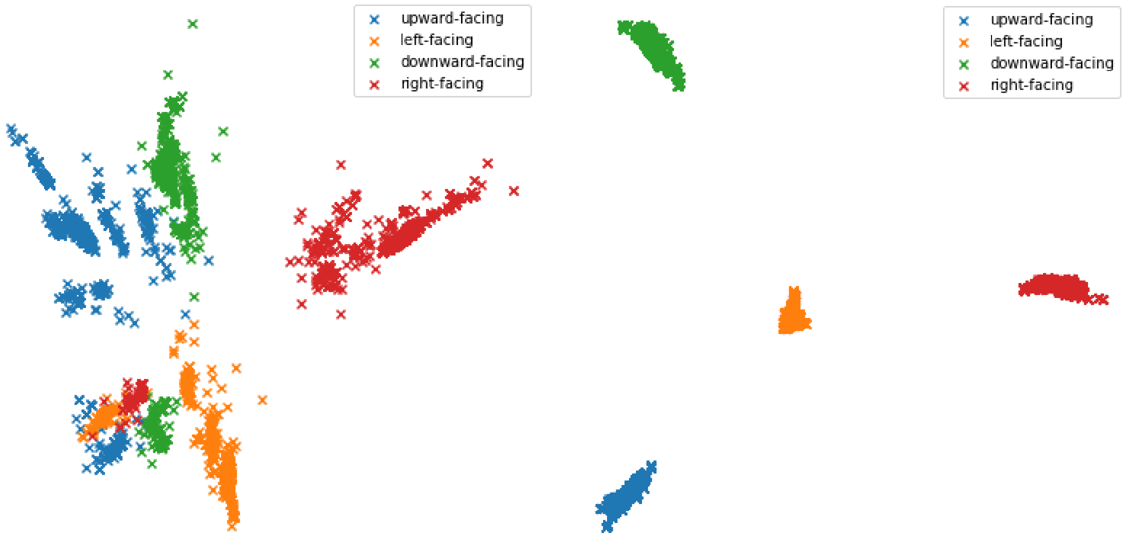
The autoencoder allows the component to work with the latent vectors of objects rather than raw object images. At this stage, every object in the training data is encoded using the autoencoder and stored in vector form. After the autoencoder has been trained and all the objects have been encoded, the properties component splits objects in groups based on their class. The component then proceeds to individually apply the following three high-level steps, which are described in further detail in the sections below, to each object type, t_i:

1. The objects with type t_i are clustered using their latent encoding and each cluster is assigned an integer identifier, c_j. More detail on the clustering is outlined in Section 1.1.1.

2. The QA pairs will label a number of the objects in the training data with some or all of their property values. ASP can be used to find a mapping from each c_j to a set of property-value pairs. The ASP program used to conduct this mapping is described in Section 1.1.2.

3. Once this mapping is known, all of the objects of type t_i in the training data can be labelled. The algorithm for finding these labels is outlined in Section 1.1.3.

As mentioned above, when the property value labels have been found for all objects of all types, the property component can be trained using the same method as the hardcoded model, discussed in Chapter **??**.



(a) Latent space of octopus rotations. **(b)** Latent space of fish rotations.

Figure 1.2: 2-dimensional representation of the latent space of octopus and fish images. The colours show different rotation values. Note, these property values come from the full-data version of the OceanQA dataset, not from clustering. Although not shown here, rocks and bags follow a very similar pattern to fish.

1.1.1 Clustering

An analysis of the full-data version of the OceanQA dataset (where objects come fully labelled with their properties) shows that the information encoded in the latent vector of each object separates objects into distinct groups based on their properties. This makes clustering a strong candidate for explicitly separating the objects into distinct groups. The Principle Component Analysis (PCA) projection of the latent vectors for octopus and fish into 2-dimensions is shown in Figure 1.2.

Clustering of the object latent vectors is done using the *Agglomerative Clustering* algorithm implemented by the *SciKitLearn*[1] library. While many other clustering algorithms are available, Agglomerative Clustering was found to work efficiently with a large number of samples. Unlike some other clustering algorithms, however, Agglomerative Clustering requires that the number of clusters to be produced is known beforehand.

Object Type	#Colours	#Rotations	#Clusters
Octopus	5	4	20
Fish	1	4	4
Bag	1	4	4
Rock	4	1	4

Table 1.1: Estimates of the number of clusters required for each class, calculated by multiplying the number of colour and rotation values found in the property questions.

[1]Available at https://scikit-learn.org/stable/modules/clustering.html

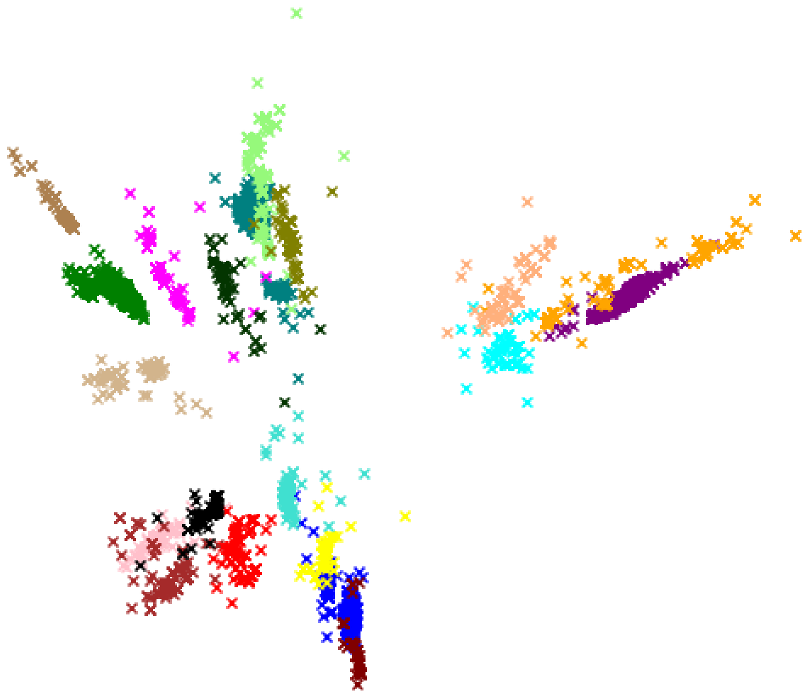


Figure 1.3: Output of the Agglomerative Clustering algorithm for the octopus images.

In order to calculate the number of clusters required for each class, we analyse the number of property values that are linked with that class in the QA pairs. For example, fish only take a single colour, silver, and so only this colour will be mentioned in the property questions that ask about fish. Additionally, fish can take four rotations, and so we expect that all four of these rotations will be mentioned in the questions. In total this gives us four possible property value combinations for fish. We repeat this process for all four object types and list the results in Table 1.1.

In theory, it is possible that this method of calculating the number of clusters required may underestimate, since not all property values may be mentioned in the QA pairs. In practice, however, this makes little difference because if a particular property value is not represented in the QA pairs it is likely to be very rare in the videos and in the evaluation questions.

Figure **??** shows the result of the clustering algorithm applied to the octopus images, projected into two dimensions. After clustering has been completed for a given class, we assign each cluster (each different colour shown in Figure 1.3) an integer identifier. The data collection problem outlined at the start of this chapter has now been reduced to finding the mapping from each cluster identifier to a set of property-value pairs, which optimises some objective function, for each object type t_i. For example, for a cluster c_j, we want to find the optimal values for colour and rotation which correspond to c_j. We denote the optimal mapping from the set of cluster identifiers C to the set of sets of property-value pairs P, for object type t_i as $f^*_{t_i} : C \to P$.

1.1.2 Property Value Optimisation

In order to find $f^*_{t_i}$, we must first define the objective function. Since we have a set of property questions and answers, we choose to maximise the number of questions answered correctly. It is now necessary to find the questions which correspond to object type t_i. We also use the question and answer parsing component (outlined in Chapter **??**) to extract the relevant symbolic information from the question and from the answer. We denote the information extracted from QA pair k as (p_k, v_k, a_k), where p_k is the property mentioned in the question (or $Null$ if there isn't one), v_k is an optional property value mentioned in the question and a_k is the answer, which is a property value.

ASP is then used to conduct the optimisation. One ASP program is constructed for each object type. We outline the ASP program for the object type t_i in the following five stages:

1. Firstly, the search space for the optimisation is constructed using choice rules. For each cluster identifier c_j, the following two choice rules are added:

$$1\{colour\_mapping(c_j, col_1); ...; colour\_mapping(c_j, col_n)\}1. \tag{1.1}$$
$$1\{rotation\_mapping(c_j, rot_1); ...; rotation\_mapping(c_j, rot_m)\}1. \tag{1.2}$$

   Where the set of possible colour values is: $\{col_1, ..., col_n\}$ *and the set of possible rotation values is:* $\{rot_1, ..., rot_m\}$.

   These choice rules generate one answer set for each possible combination of property-value pairs. Notice that no attempt has been made to restrict the property-value pairs to those which were used to select the number of clusters, since the ASP optimisation will naturally choose the property-value pairs which answer the most questions correctly.

2. Secondly, for each QA pair, the object data from the frame given in the question is added to the program. Only the objects with type t_i are added. For example, if a question says "What colour was the leftward-facing fish in frame 23?" and three fish have been detected in frame 23, then only the information for those three fish are listed in the program (fish from other questions will of course be listed in the same program).

   Each of the objects to be added will be given an integer identifier, this identifier is unrelated to the identifier assigned by the tracking component. Each object is also listed with the cluster identifier that it was assigned to. For each object listed in QA pair <*k*>, which is part of cluster c_i and has identifier id, the following predicate is added to the ASP program:

$$obj(id, c_i, \text{<}k\text{>}). \tag{1.3}$$

3. A number of helper ASP rules which convert object data in the form of the obj and <*prop*>$\_mapping$ predicates into the $holds$ predicate, which is used

to answer the questions. These two helper rules are as follows:

$$holds(colour(Val, Id), Q) :- obj(Id, Cluster, Q), colour\_mapping(Cluster, Val). \tag{1.4}$$

$$holds(rotation(Val, Id), Q) :- obj(Id, Cluster, Q), rotation\_mapping(Cluster, Val). \tag{1.5}$$

Rules 1.4 and 1.5 collate the data from the object's cluster and from the property value that that cluster has been assigned and convert this data into $holds$ form. The $holds$ predicate can then be used to answer the questions.

4. Next, the ASP rules corresponding to the questions and the ASP facts corresponding to the answers to those questions are added to the program. We firstly find the tuples (p_k, v_k, a_k) which correspond to the object type t_i and are extracted from the $<k>^{th}$ QA pair. The following rules and facts are then added if v_k is not $Null$:

$$answer(<k>, p_k, V) :- holds(p_{v_k}(v_k, Id), <k>), holds(p_k(V, Id), <k>), obj(Id, \_, <k>). \tag{1.6}$$

$$answer(<k>, p_{v_k}, V) :- holds(p_{v_k}(V, Id), <k>), holds(p_k(v_k, Id), <k>), obj(Id, \_, <k>). \tag{1.7}$$

$$expected(<k>, p_k, a_k). \tag{1.8}$$

$$expected(<k>, p_{v_k}, v_k). \tag{1.9}$$

Where p_{v_k} is the property which corresponds to property value v_k.

For example, the question "What colour is the downward-facing bag in frame 7?" and answer "white", would be converted into the following:

$$answer(<k>, colour, V) :- holds(rotation(down, Id), <k>), holds(colour(V, Id), <k>), obj(Id, \_, <k>). \tag{1.10}$$

$$answer(<k>, rotation, V) :- holds(rotation(V, Id), <k>), holds(colour(white, Id), <k>), obj(Id, \_, <k>). \tag{1.11}$$

$$expected(<k>, colour, white). \tag{1.12}$$

$$expected(<k>, rotation, down). \tag{1.13}$$

Two rules are created for questions where v_k is not $Null$ because the question gives away two pieces of information: the colour and the rotation. However, when v_k is $Null$, the following ASP rule and fact are used instead:

$$answer(<k>, p_k, V) :- holds(p_k(V, Id), <k>), obj(Id, \_, <k>). \tag{1.14}$$

$$expected(<k>, p_k, a_k). \tag{1.15}$$

5. The final part of the ASP program is the set of weak constraints used to find the optimal mapping. We firstly define the helper rule, *mapping*, which is used to collate both the colour and rotation property values for each cluster:

$$\begin{aligned} mapping(C, Col, Rot) \text{ :- } & colour\_mapping(C, Col), \\ & rotation\_mapping(C, Rot). \end{aligned} \tag{1.16}$$

The weak constraints are then defined as follows:

$$:\sim answer(Q, Prop, Val), expected(Q, Prop, Val).[-1@2, Q, Prop, Val] \tag{1.17}$$

$$\begin{aligned} :\sim \ & mapping(C1, Col, Rot), mapping(C2, Col, Rot), \\ & C1 \ != \ C2.[1@1, C1, C2, Col, Rot] \end{aligned} \tag{1.18}$$

The body of Rule 1.17 is satisfied when question Q is answered correctly. Since we are looking to maximise the number of questions answered correctly and ASP always minimises weak constraints, we give this rule a negative weight. This rule is given the higher priority of the two.

Rule 1.18, on the other hand, says that we prefer answer sets where mappings are unique. This rule could also be used as a hard constraint to rule out any answer sets where the mappings are not unique. However, we choose not to enforce this constraint so that the ASP optimiser could choose an answer set where more questions are answered correctly, at the expense of non-unique mappings. The reason for this decision is the following:

> If a group of objects with the same property values is split between two or more clusters, and these objects are commonly asked about in the QA pairs, then the ASP optimiser has the ability to assign multiple object clusters to the same property values, if it leads to a larger number of questions being answered correctly.

We therefore always try to ensure whatever rules the model learns it learns them in order to maximise the number of questions answered correctly. Hence, Rule 1.17 is given the highest priority.

The ASP program is run for each object type t_i, and *<prop>*$_mapping$ predicates contained in the optimal answer set in each case are then used to find the optimal mapping $f^*_{t_i}$, which is then stored and used to label all objects in the training data.

1.1.3 Data Labelling

Once $f^*_{t_i}$ has been found for every object type t_i, all of the objects in the training data can be labelled with property values. Labelling an object's property values requires that the centre point of each cluster be computed. For the sake of efficiency, the cluster centres are precomputed for each object type. The centre for a cluster c_j is computed as the average of the object latent vectors assigned to c_j.

After clusters centres have been computed, we assign property values to an object obj_k with type t_i as follows:

1. The object's image is encoded into a latent vector, v_k, using the autoencoder.
2. For each cluster c_j which corresponds to object type t_i, we compute the cosine distance between v and the centre of c_j. The object is assigned to the cluster with the smallest distance, which we denote c^*.
3. Using the mapping found by the ASP optimisation for object type t_i, we can simply look up the optimal set of property values that correspond to c^*.

Using this method, property values can be assigned to all detected objects in the training videos using only each object's image. After property values have been assigned, we train the property component in exactly the same way as Chapter **??**.

Although the method for training the property component outlined in this section works well when objects are simple and uniform, it is unlikely to scale to more complex, *real-world* datasets. This is because, when training an autoencoder in an unsupervised way, it can learn to extract a lot of noise from the images. When the latent vectors of object images are noisy, clustering will not be as successful in splitting the objects into groups based on their property values.

The speed of the optimisation may also be a concern if the number of property values is large, since the size of the search space scales exponentially with the average number of property values. However, for the purposes of the OceanQA dataset neither of these potential drawbacks causes any problems. The details of the evaluation of the trained property component, along with the entire trained model, is available in Chapter **??**.

1.2 Relations

Unlike the relations component in the hardcoded model, which used a manually engineered algorithm, the relations component in the trained model must learn definitions of binary relations between objects from data. As with the trained properties component outlined above, the relations component uses the QA-data version of the OceanQA dataset. Before the relations component is trained, the trained properties component is used to label all of the objects in the training data with property values.

Instead of using manually engineered functions, we opt to use a neural network consisting of fully-connected layers as the core implementation of the relations component. For a generic environment, one neural network would be created for each binary relation, however, since OceanQA only has one relation, only one network is required for this component. Each of these networks is assigned a binary relation, r, to learn. Each network therefore takes a pair of objects as input, and learns to classify the objects as either being related by r or not.

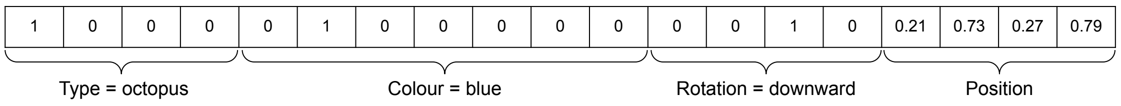


Figure 1.4: The vector encoding for a blue, downward-facing octopus, roughly in position $(54, 187, 70, 203)$. In the component itself the position is given to a much higher degree of accuracy than shown here.

Since fully-connected layers work with numbers, rather than symbolic data, we need to find a way to encode a pair of objects into a vector. To do this we encode each object in the pair separately (an example encoding of an object is shown in Figure 1.4) and then concatenate the two encodings together. The encoding of an object is the result of concatentating each of the following:

1. A one-hot encoding of the object's type.
2. A one-hot encoding of object's colour.
3. A one-hot encoding of object's rotation.
4. The position tuple for the object, where each coordinate has been divided by 256 to produce a number of between 0 and 1.

The neural network takes the vector encoding of the pair of objects as input and passes this vector through a series of fully-connected layers. The details of the layers are shown in Table 1.2. As shown in the table, the output of the network is a single neuron. The sigmoid function is applied to the output of the network to ensure the value is between 0 and 1. If the value of the output of a network learning relation r is denoted o_r, then the relation classifier component's output for a pair of objects, denoted $r(obj1, obj2)$, is as follows:

$$r(obj1, obj2) = \begin{cases} true & \text{if } o_r \geq 0.5 \\ false & \text{otherwise} \end{cases} \tag{1.19}$$

Since the OceanQA dataset has only a single binary relation, *close*, only one neural network is required to be trained for the relation component. Training this network is fairly straightforward; the process is comprised of the following steps:

Layer	Input Size	Output Size
Fully-connected 1	38	1024
Fully-connected 2	1024	256
Fully-connected 3	256	64
Fully-connected 4	64	1

Table 1.2: Details of the layers of each relation classification network. The input is the size of two encoded objects, and the output is used for binary classification, everything else (including the use of four layers) is a hyperparameter of the network. Dropout of 0.2 is also applied between each pair of layers.

1. All of the relation questions in the QA-data version of the training dataset are collated.
2. From each QA pair, a frame number, two objects (including types and, optionally, property values) and the answer are extracted using the question-and-answer parsing component.
3. The frame number and object information is used to look up any missing property values for each object. The property component has already been applied to the dataset, so the property values of all detected objects are guaranteed to be labelled. The full set of property values, as well as the object type and position from the detector, is used to construct the object encoding. The two encodings are concatenated to produce the object-pair encoding.
4. Finally, the set of object-pair encodings, and their associated answers, is used to train the network corresponding to the *close* relation.

After the network has been trained, the component can be used to classify binary relations between objects. When being applied to a given video, the relations component classifies every pair of objects in every frame of the video as being either close or not using Equation 1.19. If two objects in frame <*frame*> with identifiers <*id1*> and <*id2*>, respectively, are deemed to be close by the component, then the $obs(close(<\mathit{id1}>, <\mathit{id2}>), <\mathit{frame}>)$ predicate is stored. For any pair of objects deemed to not be close, nothing is stored; the closed-world assumption (CWA) - that is, that any predicate which cannot be proved to be true is false - is made for relations between objects.

Applying the network to every pair of objects in every frame of a video could become a burden on the speed of the relations component, especially when there are many objects in each frame. Batching pairs of objects together when applying them to the network helps to improve efficiency, but it has its limits. Chapter **??** shows that the relations component is the second slowest, after the detector, of all the components in the trained model. However, for the purposes of this project, the relations component is fast enough. Chapter **??** also shows the full details of this components performance with perfect inputs.

1.3 Events

The final of the three core components in the trained model is the event component. As with the properties and relations component, the event component uses the QA-data version of the dataset for training. However, unlike the previous components, both the properties and relations components are applied to the training data in order to extract relevant symbolic information, prior to training the event component. During training, the event component can therefore access data that has been extracted by all preceeding components in the architecture: the object detector, properties, tracking and relations components.

Since a lot of symbolic information will already have been extracted from the video during evaluation by preceeding components, the event component does not need to look at raw pixels in the video at all. Instead, it is much more efficient to use the data that has already been gleaned from the video to work out which events occur. Unlike the hardcoded events component, the trained component cannot assume to have access to an $\mathcal{AL}$ model of the environment. Instead, it must learn its own symbolic rules using the training data. One option is to learn an $\mathcal{AL}$ model and use this to find which events occur in the same way as the hardcoded component. However, these $\mathcal{AL}$ rules can be quite complex, so we do not consider this option for this project. Instead, we opt to learn ASP rules which, given a symbolic enoding for a pair of frames, return the action which occurs between the two frames. Once the actions for the entire video have been found, additional ASP rules can be used to find the effects.

1.3.1 Terminology

The ASP rules for finding the actions which occur in a video are found using a greedy search algorithm. The algorithm is run three times; once for each each action. Before outlining this algorithm, we define the terminology used as follows:

- **Feature** :- High-level names, whose values categorise the state of an object. We make use of the following features: x-position, y-position, colour, rotation and disappear (defined below). Each object has a specific value for each feature, which come (directly or indirectly) from the information extracted from the video.
- **Feature Sort** :- Each feature can be categorised into the following three *sorts*, based on the set of possible values that an object can hold for that feature:
  1. **Binary**. Disappear is the only binary feature.
  2. **Discrete** (excluding binary). Colour and rotation are discrete features.
  3. **Continuous**. x-position and y-position are both continuous features.
- **Feature Operation** :- An operation that is applied to the value of a feature. Each feature operation is usually intended to be applied to a single feature *sort*. For example, a discrete feature operation cannot be applied to continuous feature values.
- **Operation Result** :- The result of applying a feature operation to a feature value. Every operation result must be a binary value, since we want to use these values in the body of ASP rules.

The goal of the training algorithm is, given a dataset $\{(F_i, F_{i+1}, a_i)\}$, where F_i and F_{i+1} are the set of feature values for frame i and $i+1$, respectively, and a_i is the action which occurs between frame i and $i+1$, find a set of rules with feature operations in the body which best explain the data. In this case, explaining the data means maximising the number of questions answered correctly, where each QA pair contains a frame i (using which we can look up F_i and F_{i+1} in the extracted data) and an answer (which provides a_i).

As mentioned above, each feature operation is associated with a feature sort. For each sort, we now define the set of possible operations that can be applied to feature values of that sort. For every object in the video, a feature operation σ is applied to the pair $(f_{v_i}, f_{v_{i+1}})$, where f_{v_i} and $f_{v_{i+1}}$ are the object's values for feature f in frame i and $i+1$, respectively. Every set of feature operations also includes the $\top$ operation, which always evaluates to true.

Firstly, for binary features we only consider values in the initial frame, however, it would be very simple to expand this to include both frames. In the following, we show the the set of possible feature operations for a binary feature, Σ_{binary}. Each operation takes the pair $(f_{v_i}, f_{v_{i+1}})$ as input:

$$\Sigma_{binary} = \left\{ \begin{array}{l} \sigma(f_{v_i}, f_{v_{i+1}}) := f_{v_i} \\ \sigma(f_{v_i}, f_{v_{i+1}}) := \neg f_{v_i} \\ \sigma(f_{v_i}, f_{v_{i+1}}) := \top \end{array} \right\} \tag{1.20}$$

Secondly, a pair of discrete feature values $(f_{v_i}, f_{v_{i+1}})$ can have the following feature operations applied:

$$\Sigma_{discrete} = \left\{ \begin{array}{l} \sigma(f_{v_i}, f_{v_{i+1}}) := f_{v_i} = v_n, f_{v_{i+1}} = v_m \textit{ for each } (v_n, v_m) \\ \sigma(f_{v_i}, f_{v_{i+1}}) := f_{v_i} = f_{v_{i+1}} \\ \sigma(f_{v_i}, f_{v_{i+1}}) := f_{v_i}! = f_{v_{i+1}} \\ \sigma(f_{v_i}, f_{v_{i+1}}) := \top \end{array} \right\} \tag{1.21}$$

Where (v_n, v_m) loops over all possible combinations of feature values.

Finally, a pair of continuous feature values $(f_{v_i}, f_{v_{i+1}})$ can take the following feature operations:

$$\Sigma_{continuous} = \left\{ \begin{array}{l} \sigma(f_{v_i}, f_{v_{i+1}}) := f_{v_i} > f_{v_{i+1}} \\ \sigma(f_{v_i}, f_{v_{i+1}}) := f_{v_i} < f_{v_{i+1}} \\ \sigma(f_{v_i}, f_{v_{i+1}}) := f_{v_i} = f_{v_{i+1}} \\ \sigma(f_{v_i}, f_{v_{i+1}}) := f_{v_i}! = f_{v_{i+1}} \\ \sigma(f_{v_i}, f_{v_{i+1}}) := \top \end{array} \right\} \tag{1.22}$$

The $\top$ feature operations always gives an operation result of *true*. Intuitively, a $\top$ feature operation has no effect on the outcome of the rule it is in. Since the algorithm has the option to choose the $\top$ feature operation for any feature, we can enforce that all features appear in every rule body at least once. Any feature that the algorithm doesn't care about can be assigned a $\top$ feature operation. For simplicity, we also enforce that each feature can appear at most once in a rule body. The search algorithm therefore tries to learn a set of rules, where each rule takes the following form:

$$\begin{aligned} a_i \textbf{ if } & \sigma_{x\_position}(f_{v_i}, f_{v_{i+1}}), \sigma_{y\_position}(f_{v_i}, f_{v_{i+1}}), \sigma_{colour}(f_{v_i}, f_{v_{i+1}}), \\ & \sigma_{rotation}(f_{v_i}, f_{v_{i+1}}), \sigma_{disappear}(f_{v_i}, f_{v_{i+1}}) \end{aligned} \tag{1.23}$$

Where each $\sigma_{<f>}$ is a member of <f>'s set of possible feature operations.

Each feature operation is also assigned a *weight*. The weight of a rule is then defined as the sum of the weights of its operations. Secondary goals for the optimisation algorithm are, firstly, to minimise the number of rules and, secondly, to minimise the sum of the weights of the rules. In the case of the above feature operations, all operations other than $\top$ are assigned a weight of 1, and $\top$ is assigned a weight of 0. This encourages the optimiser to use $\top$ as much as possible, as long as the set of rules continue to perform well.

Before outlining an Inductive Logic Programming (ILP) search algorithm for finding a set of rules, we first give an example which explains the terminology.

Example. Consider an element of the dataset which includes an action, *move*, and an octopus which is upward-facing, blue and has position $(100, 40, 116, 56)$ in frame 6, and is upward-facing, purple and has position $(100, 25, 116, 41)$ in frame 7. For simplicity we only consider the x-position and y-position features. Each position feature only uses the top left corner of the object's bounding box, since both corners move by the same amount. The position feature operations are given as follows:

x-position operation $\to$ result	y-position operation $\to$ result
$100 > 100 \to \mathit{false}$	$40 > 25 \to \mathit{true}$
$100 < 100 \to \mathit{false}$	$40 < 25 \to \mathit{false}$
$100 = 100 \to \mathit{true}$	$40 = 25 \to \mathit{false}$
$100\ != 100 \to \mathit{false}$	$40\ != 25 \to \mathit{true}$
$\top \to \mathit{true}$	$\top \to \mathit{true}$

Clearly, given only this single example, there are many possible rules which the algorithm could learn, many of which will be undesirable. However, when there are a more significant number of examples, the algorithm should be able to learn rules which correctly model the environment.

1.3.2 Search Strategy

Equation 1.23 shows that, for each rule, the training algorithm needs to search over the set of all possible combinations of feature operations, where each feature is assigned exactly one operation in the rule body. Even for a relatively simple environment, such as OceanQA, this is a very large search space, and when the possibility of searching over an arbitrary number of rules is taken into account the space becomes even larger. The number of feature operations for each feature and the total search space size for a complete algorithm which learns only a single rule is shown in Table 1.3.

To ensure the training algorithm remains tractable, we implement a greedy search over features, rather than a complete search over the entire space. Our search algorithm takes each feature individually and finds the operation for the given

Feature	Search Size
x-position	5
y-position	5
colour	52
rotation	19
disappear	3
Total	74100

Table 1.3: Number of feature operations for each feature. The total search size is then calculated by multiplying all these sizes together.

feature which maximises the number of action questions answered correctly. After each optimisation, we store the set of operations which have been selected as the *accumulated operations*. When the next feature is optimised, each rule in the new hypothesis can choose to use either the full set of stored operations or none - using none means setting all previously optimised feature operations to $\top$.

Algorithm 1 gives a high-level overview of the steps required to find the optimal hypothesis - the set of rules which correctly answers the most questions - for every action. Algorithm 1 makes use of the OPTIMISEFEATURE function, which takes as input: an action, a set of feature operations, the accumulated feature operations and the training data. This function searches over all of the feature operations to find the set of rules which maximise the number of questions answered correctly. It also searches over the number of rules to be used, as well as whether each of those rules should use an accumulated operation set or not. We implement this function using an ASP optimisation program, the encoding for which is outlined in Section 1.3.3. The GENHYP function shown in Alorithm 1 generates the hypothesis for action a using the accumulated feature operations. Section 1.3.4 provides more detail on how events are detected after the hypothesis has been learnt.

In the remainder of this section we provide an inductive description for the task which the OPTIMISEFEATURE function must solve.

Algorithm 1 Finding the optimal hypothesis for each action

```
1: procedure OptimiseRules($actions, features, data$)
2:     $hyps \leftarrow \{\}$
3:     for $a \leftarrow actions$ do
4:         $acc \leftarrow \{\}$
5:         for $f \leftarrow features$ do
6:             $s \leftarrow$ SortOf($f$)
7:             $acc \leftarrow$ OptimiseFeature($action, \Sigma_s, acc, data$)
8:         end for
9:         $hyps \leftarrow hyps +$ GenHyp($a, acc$)
10:    end for
11:    return $hyps$
12: end procedure
```

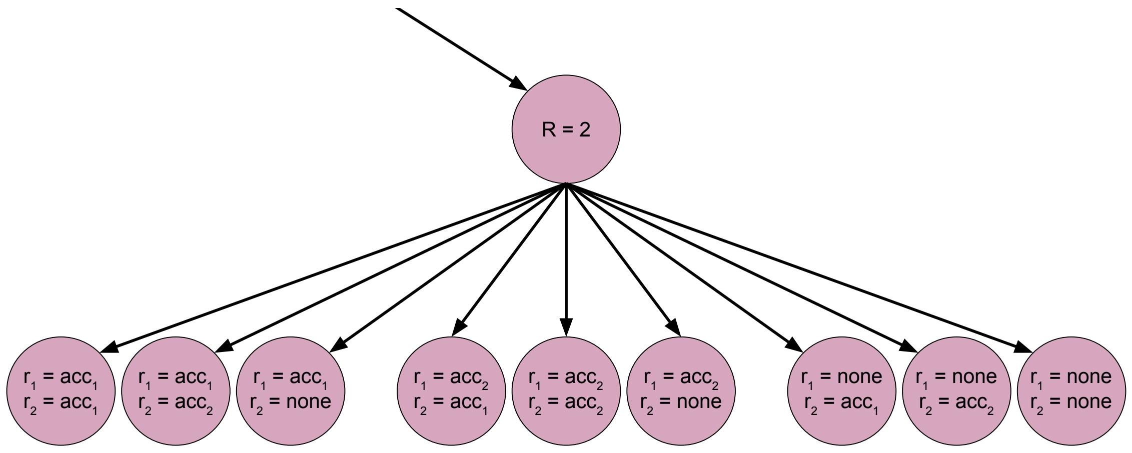


Figure 1.5: Accumulator assignment nodes in the search tree, where the number of rules is two and two accumulators are carried forward.

Consider an optimisation which uses features $f_1, ..., f_{feat}$, and that the algorithm is currently optimising feature f_i. The algorithm will therefore be given: an action a; the set of feature operations Σ_s, where s is the sort of f_i; the *accumulated operations* $acc_1, ..., acc_m$ from the previous feature optimisation, where each acc_j is the set of feature operations used by rule r_j in the previous optimisation; and the training data D. In the case where $i = 1$, the *accumulated operations* contains a single rule whose feature operations are all set to $\top$. The search space for feature optimisation f_i can then be constructed as follows:

1. At the highest level of the search, the algorithm needs to choose how many rules to learn. Therefore we start by creating one node in the search tree for each possible number of rules in $\{1, 2, ..., R_{max}\}$, where R_{max} is a hyperparameter of the search.

2. Then, for each possible number of rules, each rule can choose to use at most one of $\{acc_1, ..., acc_m\}$. Therefore, for each previous node n, which allows R rules, each rule in $\{r_1, ..., r_R\}$ chooses between m accumulators or none, meaning $(m + 1)^R$ child nodes must be added to n, since $(m + 1)^R$ is the number of possible combinations of accumulator choices. This part of the search tree is shown in Figure 1.5 for the node where $R = 2$ and two accumulators are carried over.

3. Finally, after choosing which accumulator to use (or none), each rule chooses a feature operation for the feature f_i. This means, for each leaf node n constructed so far, we add a child node to n for each possible feature operation $\sigma_i \in \Sigma_s$.

Once the search space has been constructed, the optimiser evaluates each of the leaf nodes in the tree outlined above, and chooses the node which maximises the number of action questions answered correctly. The OPTIMISEFEATURE function then returns the accumulated set of feature operations, $\{acc_1, ..., acc_i\}$, so that they can be used for the next feature optimisation.

1.3.3 ASP Encoding

This section outlines the details the ASP program which is used in place of the OPTIMISEFEATURE function described above. As expected, the program creates the search space using choice rules, and evaluates each possible answer set by counting the number of action questions it answers correctly.

The ASP program for a feature f, feature operations $\Sigma = \{\sigma_1, ..., \sigma_n\}$, action $action$, accumulator set $\{acc_1, ..., acc_m\}$ and dataset D is constructed as follows:

1. Firstly, the action-type questions in D (including those which do not mention $action$) are collated, these are denoted D_a. From the questions in D_a we extract the frame number i using the question-and-answer parser, and use this frame number to look up the symbolic representation for frame i and $i+1$. We collate these symbolic representations, along with each action, as the tuple $(F_i, F_{i+1}, a)_j$, where j is a unique integer identifier assigned to each tuple in the dataset. We then represent F_i and F_{i+1} in ASP as described in Chapter **??**, with the exception that F_i is assigned frame identifier j, and F_{i+1} is assigned frame identifier $-j$. This is done so that all frames in the dataset can be added to the same ASP program. The action $action$ for tuple j is then represented as:
$$actual(action, j). \tag{1.24}$$

2. Secondly, each accumulated feature operation, $acc_i = \{(\mathit{feat}, \sigma_{id})\}$, is a set of tuples, where *feat* is the name of a feature and σ_{id} is the identifier assigned to a feature operation (each operation is assigned an integer identifier which is unique within its feature sort). Each element $(\mathit{feat}, \sigma_{id})$ of acc_i is represented using the following predicate:
$$acc(\mathit{feat}, \sigma_{id}, i). \tag{1.25}$$

3. Rules 1.26 and 1.27, which follow, define the search space for the number of rules and give each rule an identifier, respectively.
$$1\{max\_rule\_id(1..\text{<}\textit{max\_rules}\text{>})\}1. \tag{1.26}$$
$$rule(1..Max) \text{ :- } max\_rule\_id(Max). \tag{1.27}$$
Where <max_rules> is a hyperparameter of the component, which specifies the maximum number of rules in the search.

4. Next, we need to assign each rule one of the accumulated sets of operators or none. This is done using the following helper and choice rules:
$$acc\_id(Acc\_id) \text{ :- } acc(\_, \_, Acc\_id). \tag{1.28}$$
$$1\{use\_acc(Acc\_id, Rule) : acc\_id(Acc\_id) \ ; \ no\_acc(Rule)\}1 \text{ :- } rule(Rule). \tag{1.29}$$

There are also a number of rules which set the feature operations (using the $operation$ predicate, described below) based on the accumulator, however, these are not shown here. On the first feature optimisation, we also set any feature which is not being optimised to have operation $\top$, which ensures that all features are represented in the accumulator for successive optimisations.

5. We now need to generate the possible feature operations that feature f can take. We do this using the following ASP choice rule:

$$1\{operation(f, Op\_id, R) : op\_weight(f, Op\_id, \_)\}1 \text{ :- } rule(R). \tag{1.30}$$

Rule 1.30 generates a potential answer set for each operation of f. This rule uses the op_weight predicate, which defines the weight assigned to each operation for a feature, however, here it is simply used to fetch the operation identifier for each operation of f.

6. Next, we add the rules corresponding to feature operations, $\Sigma = \{\sigma_1, ..., \sigma_n\}$, for f. Each operation σ_i is a function of two symbolic feature values from an object. Encoding this operation in ASP therefore requires additional body predicates which can fetch these symbolic values. For example, one of the operations for the *colour* feature is: $f_i = green, f_{i+1} = brown$, which require the use of the $obs(colour(f_i, Id), I)$ and $obs(colour(f_{i+1}, Id), -I)$ predicates. In the general case, we refer to each of these predicates as $p(f_i, Id, I)$ and $p(f_{i+1}, Id, -I)$, and the symbolic representation of the feature operation as $op(\sigma_i, f_i, f_{i+1})$. We can then construct the ASP rule for an operation σ_i from feature f as follows:

$$\begin{aligned} op\_result(f, Id, I, Rule) \text{ :- } & op(\sigma_i, f_i, f_{i+1}), p(f_i, Id, I), \\ & p(f_{i+1}, Id, -I), operation(f, i, Rule). \end{aligned} \tag{1.31}$$

The $op\_result(f, Id, I, R)$ predicate stores the binary operation result of feature f in rule R for an object with identifier Id in frame I. If op_result is in the answer set, it is storing the result *true*, otherwise, it is storing *false*.

7. We also need to add the operation weight for each feature operation. To do this we add the following predicate for each operation σ_i of feature f:

$$op\_weight(f, i, \text{<weight>}). \tag{1.32}$$

Where <weight> is 0 for the $\top$ operation and 1 otherwise.

8. Since it is the only feature which does not come directly from the symbolic information extracted from the video, we must provide a definition for the *disappear* feature. Intuitively, if an object with identifer Id exists in frame I, but not in frame $-I$, then $disappear(Id, I)$ is true (as with actions, disappear can be thought of as occurring between one frame and the next; by convention is is assigned to the initial frame).

The ASP rule for disappear is as follows:

$$\begin{aligned} \mathit{disappear}(\mathit{Id}, I) \text{ :- } & \mathit{obs}(\mathit{class}(\mathit{Class}, \mathit{Id}), I), \\ & \mathit{not}\ \mathit{obs}(\mathit{class}(\mathit{Class}, \mathit{Id}), -I), \\ & \mathit{step}(-I), \\ & \mathit{step}(I). \end{aligned} \tag{1.33}$$

9. We are now in a position to add the rule which generates the prediction of the action action from the operation results. This rule is as follows:

$$\begin{aligned} \mathit{predicted}(\mathit{action}, \mathit{Frame}) \text{ :- } & \mathit{non\_static}(\mathit{Id}, I), \\ & \mathit{op\_result}(\mathit{x\_position}, \mathit{Id}, I, \mathit{Rule}), \\ & \mathit{op\_result}(\mathit{y\_position}, \mathit{Id}, I, \mathit{Rule}), \\ & \mathit{op\_result}(\mathit{rotation}, \mathit{Id}, I, \mathit{Rule}), \\ & \mathit{op\_result}(\mathit{colour}, \mathit{Id}, I, \mathit{Rule}), \\ & \mathit{op\_result}(\mathit{disappear}, \mathit{Id}, I, \mathit{Rule}), \\ & \mathit{rule}(\mathit{Rule}). \end{aligned} \tag{1.34}$$

Where $\mathit{non\_static}(\mathit{Id}, I)$ is true for the octopus with identifier Id in frame I.

10. Finally, we can add the following weak constraints to the ASP program, which is learning rules for action:

$$:\sim \mathit{predicted}(\mathit{action}, I), \mathit{actual}(\mathit{action}, I).[-1@3, I] \tag{1.35}$$

$$:\sim \mathit{actual}(A, I), A\ != \mathit{action}, \mathit{not}\ \mathit{predicted}(\mathit{action}, I).[-1@3, I] \tag{1.36}$$

Rule 1.35 is satisfied when the predicted action matches the actual action (defined by each element of the dataset). This rule is maximising (since the weight is negative) the number of true positives generated by the action rules. On the other hand, Rule 1.36 maximises the number of true negatives, since its body is satisfied if the learnt hypothesis does not predict action and action is not the answer for this element of the dataset.

We also add the following weak constraints to minimise the size of the learnt hypothesis:

$$:\sim \mathit{count\_rules}(\mathit{Cnt}).[\mathit{Cnt}@2] \tag{1.37}$$

$$:\sim \mathit{Sum} = \#\mathit{sum}\{W, R : \mathit{rule\_weight}(W, R), \mathit{rule}(R)\}.[\mathit{Sum}@1] \tag{1.38}$$

Where $\mathit{count\_rules}(\mathit{Cnt})$ is true when Cnt is the number of rules in the answer set, and $\mathit{rule\_weight}(W, R)$ is true when W is the weight of R.

Rule 1.37 attempts to minimises the number of rules which appear in the learnt hypothesis, while Rule 1.38 attempts to minimise the total weight of the rules in the hypothesis. Both of these rules, however, are given lower priority than the true positive and true negative rules, since we always prefer a more accurate solution, even if it is longer.

1.3.4 Detecting Events

The algorithm described in the previous sections is used to learn one hypothesis (a set of rules) for each action using the training data. After training, each hypothesis can be used to detect actions in a given video by constructing an ASP program which contains both the encoding of the video and the encoding of the hypothesis. Running this program will return a sequence of actions.

However, we have not outlined a method for detecting effects. Since there are questions in the dataset which concern only effects, we cannot run an ILP algorithm to learn effect definitions in the same as was done for actions. This is an opportunity for a hybrid model to showcase one of its key advantages; the ability to inject background knowledge of an environment directly into the model, without needing to learn anything.

In this section we outline a number of ASP rules used to detect effects. These rules are included in the same ASP program as the learnt action hypotheses and the encoded video. This program is therefore able to return the set of events which occur in the video.

The ASP rule for effects are as follows:

- Firstly, the rule for the *eat a fish* effect, which follows, says that a fish is eaten by the octopus if it exists in frame I, but not in frame $I + 1$. The $exists(id, i)$ predicate is a helper predicate with the intuitive meaning that an object with identifier id is present in frame i.

$$\begin{aligned} occurs\_effect(eat\_a\_fish(Octo), I) \text{ :- } & obs(class(fish, Fish), I), \\ & not\ exists(Fish, I + 1), \\ & obs(class(octopus, Octo), I), \\ & step(I + 1). \end{aligned} \tag{1.39}$$

- Secondly, the *eat a bag* effect is defined similarly, except that both the bag and octopus must not be present in frame $I + 1$:

$$\begin{aligned} occurs\_effect(eat\_a\_bag(Octo), I) \text{ :- } & obs(class(bag, Bag), I), \\ & not\ exists(Bag, I + 1), \\ & obs(class(octopus, Octo), I), \\ & not\ exists(Octo, I + 1), \\ & step(I + 1). \end{aligned} \tag{1.40}$$

- Finally, *change colour* is defined in terms of the $changed(p, before, after, id, i)$ predicate, which says that property p has value $before$ in frame i, but changed to value $after$ in frame $i + 1$.

$$\begin{aligned} changed(colour, Before, After, Id, I) \text{ :- } & obs(colour(Before, Id), I), \\ & obs(colour(After, Id), I + 1), \\ & Before\ != After. \end{aligned} \tag{1.41}$$

The *change colour* effect is then simply defined as follows:

$$occurs\_effect(change\_colour(Id), I) \text{ :- } changed(colour, \_, \_, Id, I). \qquad (1.42)$$

While it can be very useful to allow background knowledge to be injected into the H-PERL model, there are a few trade-offs. Firstly, the model is less adaptable; porting the model to a new environment requires rewriting the background rules. And secondly, these hardcoded rules may include human biases or errors, this could mean that the rules are not as accurate as they could be.

After the action hypotheses have been learnt and the effect rules written, detecting both the actions and effects has the advantage of being very fast, since no searching for a solution is required. The ASP program for finding the actions and effects is simply run once, and contains no choice rules. As Chapter **??** will show, the trained event detection component is significantly faster than its hardcoded counterpart.