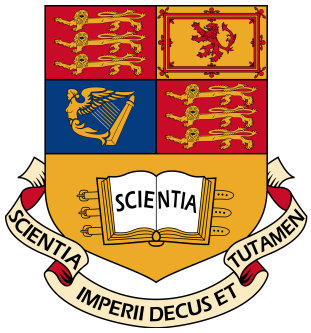


DEPARTMENT OF COMPUTING

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 9, 2020

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.

As with the hardcoded model, full performance evaluation details of the trained model and some of its components can be found in Chapter **??**.

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 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 creates a 'chicken-and-egg' problem when collecting the training data; the model needs a trained property extractor in order to find the images to train the property extractor with. This means the training data cannot be collated in the same way as the hardcoded model.

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 **??**.

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

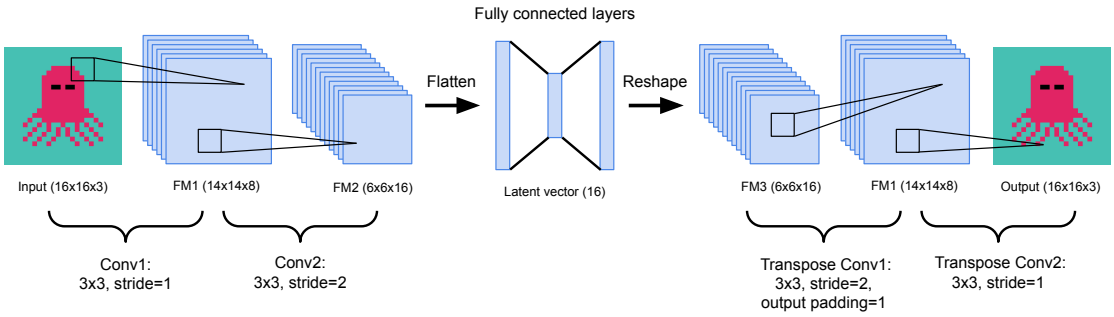


Figure 1.1: An illustration of the autoencoder architecture. Although not shown here, the network contains batch normalisation layers between each pair of convolution layers. FM stands for feature maps, and the dimensions of each feature maps are given as $(\textit{height}, \textit{width}, \textit{number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

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.

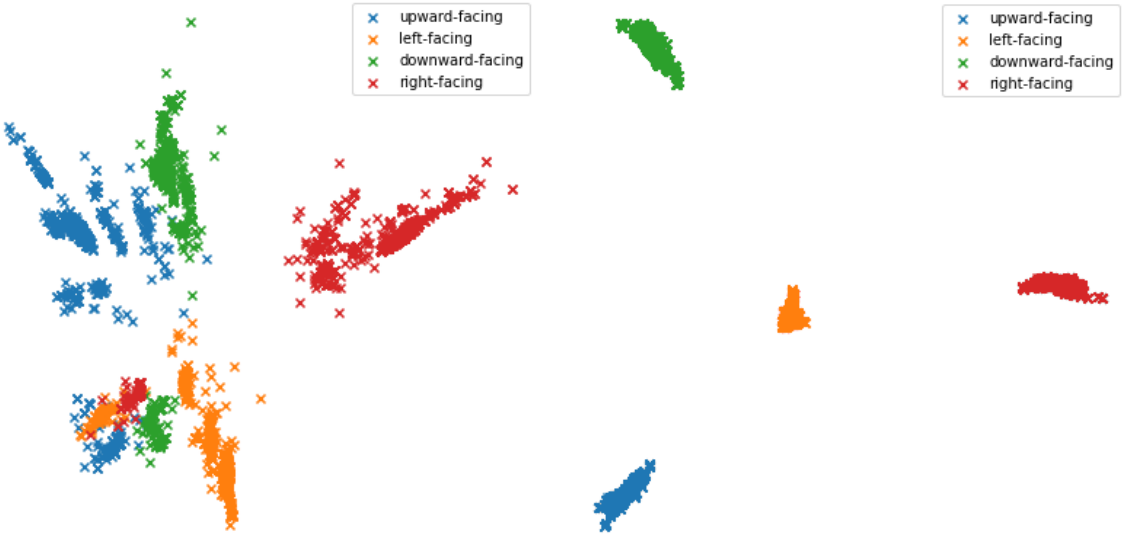
The autoencoder allows the component to work with the latent vectors of objects rather than raw object images. At this stage, every object in the training data is encoded using the autoencoder and stored in vector form. After the autoencoder has been trained and all the objects have been encoded, the 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

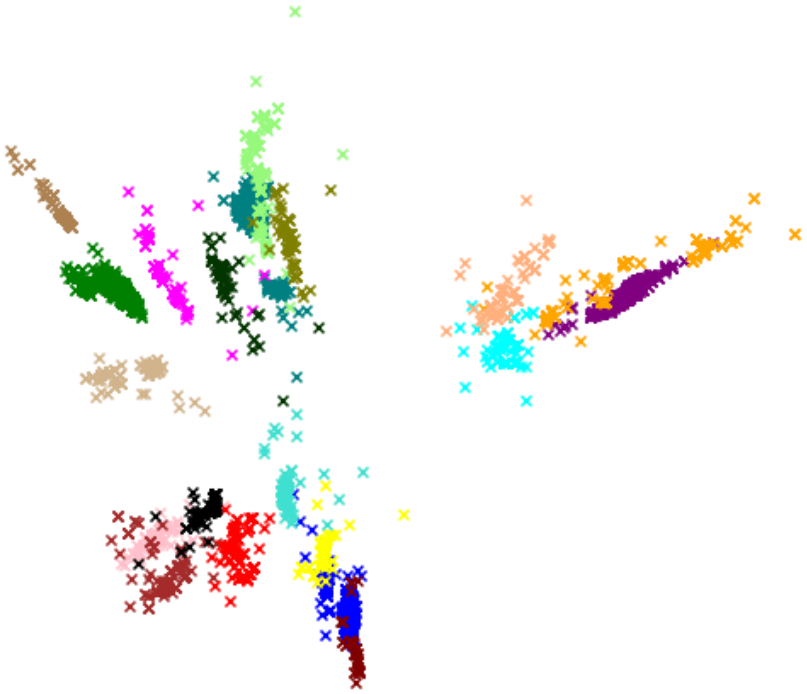


Figure 1.3: Output of the Agglomerative Clustering algorithm for the octopus images.

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 $f^*_{t_i} : C \rightarrow P$.

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:

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

   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:

$$obj(id, c_i, <k>). \quad (1.3)$$

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:

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

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

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 are extracted by the question and answer parsing component from the $<k>^{th}$ QA pair. The following rule and fact are then added if v_k is not $Null$:

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

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

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

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

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:

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

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

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

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

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:

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

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

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:

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

The weak constraints are then defined as follows:

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

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

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 exponentionally 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.

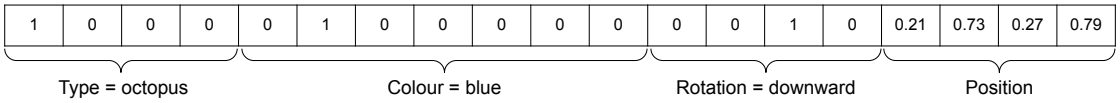


Figure 1.4: The vector encoding for a blue, downward-facing octopus, roughly in position $(54, 187, 70, 203)$. In the component itself the position is given to a much higher degree of accuracy than shown here.

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. 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.

1.3 Events