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 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 **??**.

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

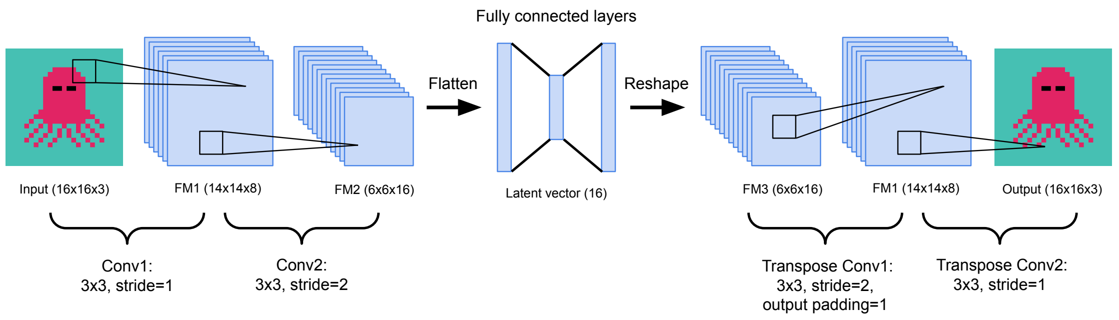


Figure 1.1: An illustration of the autoencoder architecture. Although not shown here, the network contains batch normalisation layers between each pair of convolution layers. FM stands for feature maps, and the dimensions of each feature maps are given as $(height, width, number\ of\ feature\ maps)$.

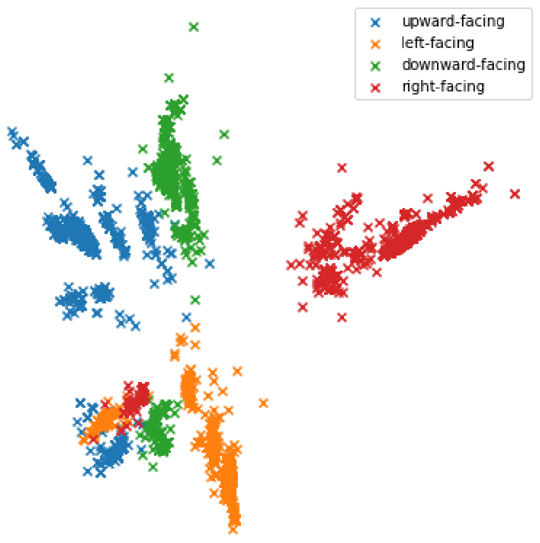
40,000 of the objects detected by the object detector in the training data, where each object type is equally represented in the sample. The architecture of the network is shown in Figure 1.1. Each object image is resized to 16x16 pixels and the network is trained with a learning rate of 0.001 for 5 epochs.

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 apply the following three high-level steps, which are described in further detail below, to each object type, t_i, individually:

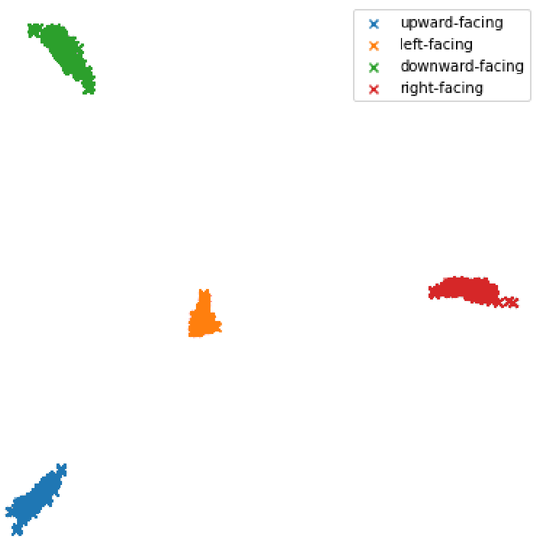
1. The vectors of the objects with type t_i are clustered and each cluster is assigned an integer identifier, c_i.
2. The QA pairs will label a number of the objects in the training data with some or all of their property values. ASP can be used to find a mapping from each c_i to a set of property-value pairs.
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 below.

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 **??**.

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.



(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.

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.

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.

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

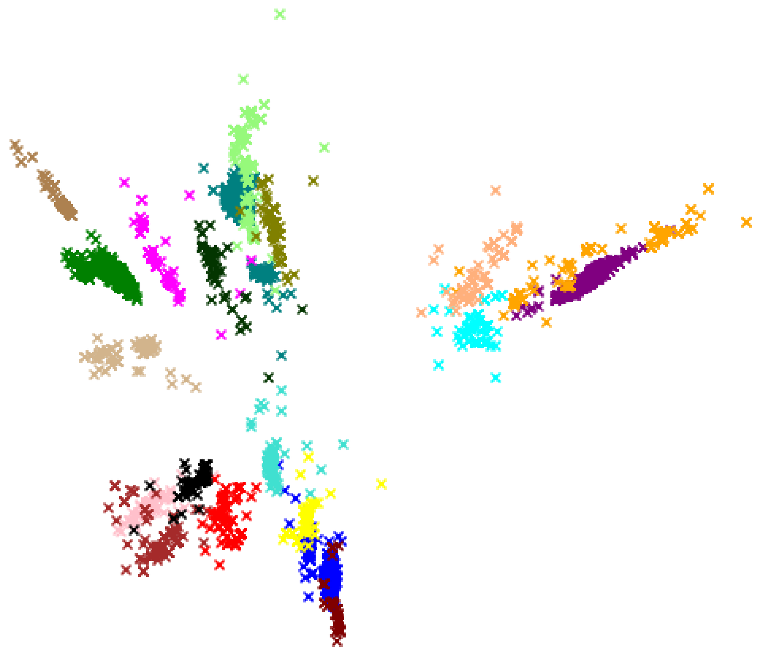


Figure 1.3: Output of the Agglomerative Clustering algorithm for the octopus images.

In 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_i, we want to find the optimal values for colour and rotation which correspond to c_i. 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 $f^*_{t_i} : C \to P$.

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 i as $(p, v, t, a)_i$, where p is the property mentioned in the question (or $Null$ if there isn't one), v is an optional property value mentioned in the question, t is the object type mentioned in the question and a 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 TODO stages:

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

$$1\{colour\_mapping(c_i, col_1); ...; colour\_mapping(c_i, col_n)\}1. \tag{1.1}$$

$$1\{rotation\_mapping(c_i, rot_1); ...; rotation\_mapping(c_i, rot_m)\}1. \tag{1.2}$$

  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.

- Secondly, the ASP rules corresponding to the questions and the ASP facts corresponding to the answers to those questions are added to the program. For the tuple $(p_i, v_i, t_i, a_i)_{<i>}$ extracted by the question and answer parsing component from the i^{th} QA pair, the following rule and fact are added if v_i is not $Null$, where p_{v_i} is the property which corresponds to property value v_i:

$$\begin{aligned} answer(<i>, p_i, V) \text{ :- } & holds(p_{v_i}(v_i, Id), <i>), \\ & holds(p_i(V, Id), <i>), obj(Id, \_, <i>). \end{aligned} \tag{1.3}$$

$$\begin{aligned} answer(<i>, p_{v_i}, V) \text{ :- } & holds(p_{v_i}(V, Id), <i>), \\ & holds(p_i(v_i, Id), <i>), obj(Id, \_, <i>). \end{aligned} \tag{1.4}$$

$$expected(<i>, p_i, a_i). \tag{1.5}$$

$$expected(<i>, p_{v_i}, v_i). \tag{1.6}$$

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

$$\begin{aligned} answer(<i>, colour, V) \text{ :- } & holds(rotation(down, Id), <i>), \\ & holds(colour(V, Id), <i>), obj(Id, \_, <i>). \end{aligned} \tag{1.7}$$

$$\begin{aligned} answer(<i>, rotation, V) \text{ :- } & holds(rotation(V, Id), <i>), \\ & holds(colour(white, Id), <i>), obj(Id, \_, <i>). \end{aligned} \tag{1.8}$$

$$expected(<i>, colour, white). \tag{1.9}$$

$$expected(<i>, rotation, down). \tag{1.10}$$

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

$$answer(<i>, p_i, V) \text{ :- } holds(p_i(V, Id), <i>), obj(Id, \_, <i>). \tag{1.11}$$

$$expected(<i>, p_i, a_i). \tag{1.12}$$

1.2 Relations

1.3 Events