Warsaw University of Technology

FACULTY OF
MATHEMATICS AND INFORMATION SCIENCE

Master's diploma thesis

in the field of study Data Science

Exploration of the usage of semantic reasoning in reinforcement learning

Olaf Werner

student record book number 291139

thesis supervisor

Doctor of Computer Science Maria Ganzha

WARSAW 2022

Abstract

Exploration of the usage of semantic reasoning in reinforcement learning

There are two main approaches to developing an AI: reasoning in formal languages using facts from ontologies and creating a policy by iteration using actions and rewards. Formal languages require a discrete state space while policy is most often created on continous state space. Other works only partially connect two approaches and it involves a lot of hand engineering. This thesis proposes a new architecture which would reconcile two approaches and get best of both worlds fully automatically.

Keywords: ontology, reinforcement learning, semantic reasoning, image segmentation, planning

Streszczenie

Eksploracja użycia semantycznego rozumowania w uczeniu ze wzmacnianiem

Istnieją dwa główne podejścia do rozwoju sztucznej inteligencji: rozumowanie w językach formalnych z wykorzystaniem faktów z ontologii oraz tworzenie polityki poprzez iterację z wykorzystaniem akcji i nagród. Języki formalne wymagają dyskretnej przestrzeni stanów, podczas gdy polityka jest najczęściej tworzona na ciągłej przestrzeni stanów. Inne prace tylko częściowo łączą dwa podejścia i wymagają dużej ilości ręcznej pracy. Ta praca proponuje nową architekturę, która pogodziłaby dwa podejścia i w pełni automatycznie wykorzystałaby to, co najlepsze z obu światów.

Słowa kluczowe: ontologia, uczenie przez wzmacnianie, rozumowanie semantyczne, segmentacja obrazu, planowanie

Contents

Introduction

Building a machine that could reason about a complex world as well as a human is still a milestone to be achieved. Historically one of the first attempts to achieve it was building a hard-coded knowledge base in formal languages. One of the most famous of such projects was [30]. Unfortunately, such an approach was not able to deliver satisfying results. Researchers had to input formal rules manually, which was a long and error-prone process. For example, Cyc failed to understand that a person was still a human while shaving, because he was holding an electrical razor, so he was assumed to have some electrical parts [14]. On the other hand, reinforcement learning proposed a completely different approach. In this kind of learning, an agent has no prior knowledge about the world. It instead tries to assign values to combinations of states and actions using the reward function. Reinforcement learning has been much more successful. Reinforcement learning is used in many areas like self-driving cars, natural language processing, medicine, engineering, etc. Unfortunately, this approach also has its limitations. One of the main limitations is the curse of dimensionality. The number of combinations of the different states and actions can be enormous.

The first approach had trouble getting knowledge from real experience and the second approach has trouble with more advanced reasoning.

In this thesis, we will try to reconcile two approaches.

1. Proposed Solution high level overview

Philosophy of data processing is a complex one [52]. However, we can make a few important remarks. First of all, representation of data is a crucial aspect in many cases. For example for SVM [7] transforming data in a correct way can have huge impact for separability. Many algorithms also require data in correct formats, usually they expect data to be constant length. In this work, we want to use ontology to guide the reinforcement learning process. To do that, we must "elevate" high dimensional continuous data to low dimensional semantic space, use reasoning in semantic space, and then "ground" it back to a form suitable for the reinforcement learning agent Fig. 1.1. Connections can also occur between other modules than shown in the picture, for example between State and Grounding, as it may be necessary for Grounding to work. Also, some algorithms may implement multiple modules at once, usually Dimension Reduction and Clustering, as in the case of semantic image segmentation [49]. However, the main idea of this graph was to show the process of "elevating" then "grounding" the data to provide an agent with higher quality data and goals. Also, modules of this architecture can be implemented in many different ways. For example, module Clustering can be implemented as SVM [31], self-organizing maps [58] or Natural Language Sentence Generator [49].

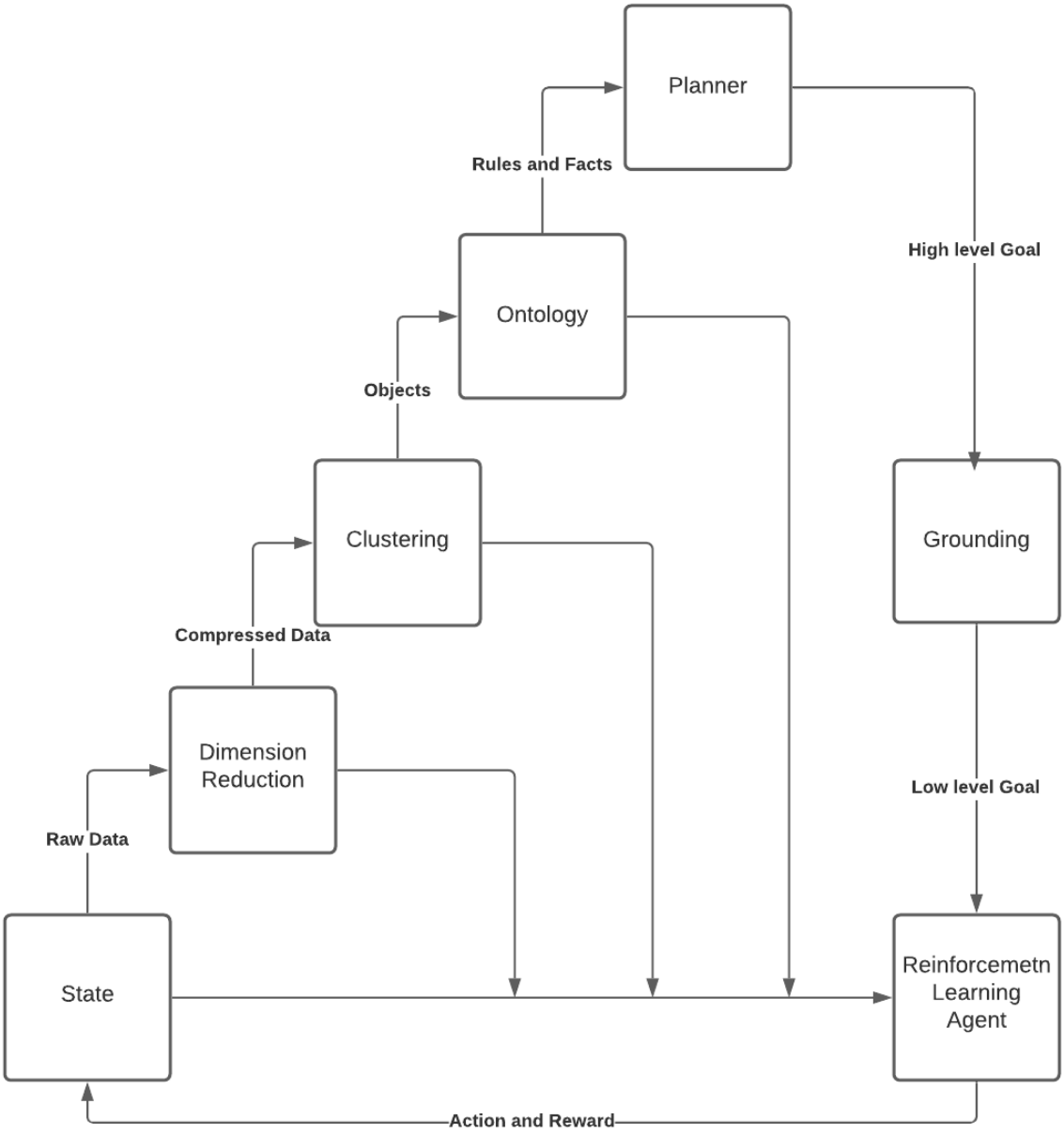

Figure 1.1: Main idea of overall architecture

Architecture modules summary:

1. **State**: is simply raw visual input which the agent sees. This data is highly dimensional and continuous. Because of that it should be preprocessed before given to the Agent.

2. **Dimension Reduction**: This module is lowering the number of dimensions of the initial State. Most of the features the agent sees may be irrelevant for a given task.

3. **Clustering**: In this module, we try to distinguish particular objects from what agent sees. Objects are essentially clusters of correlated features persistent through time. For example, an agent seeing a falling apple will notice that red pixels tend to go together. While this step is also simplifying State, it is distinct from Dimension Reduction. Dimension Reduction is mostly about removing irrelevant features, while Clustering is about discretization and grouping.

4. **Ontology**: Ontology module will store knowledge about the environment, like how objects are moving and what are possible transitions between states. This knowledge will be used in the Planner module. More formal definition is given in the section Ontology and Semantic Reasoning.

5. **Planner**: The Planner module will find the best way to solve a problem based on its predictions of future states thanks to the rules and facts in the Ontology module. After that, it will generate a high-level actions sequence and a new current high-level goal for the Agent to pursue.

6. **Grounding**: Grounding module will map high-level goals to low-level goals for an agent to pursue.

7. **Reinforcement Learning Agent**: The Reinforcement Learning Agent will take preprocessed data and goals to develop a policy for a given problem.

2. Fundamentals

2.1. Reinforcement Learning

Reinforcement Learning [54] is a trial-and-error method in which the agent learn through interaction with the environment by choosing the action according to its policy, which is sent to the environment, then the environment moves to a new state, an immediate reward is received and policy is updated Fig. 2.1. The agent use the actions with the most positive reward repeatedly.

One of the most basic algorithms is Q-learning [54]. In Q-learning agent computes the Q-value that estimates discounted cumulative reward of actions while following policy π: $R = \sum_{t=1}^{\infty} \gamma r_t$, where $\gamma \in [0, 1)$ is a discount rate that shows how much the future rewards contribute to the total reward, and r_t is the reward at a time step t . The agent becomes farsighted if γ values are closer to 1 and the agent becomes shortsighted if γ values are closer to 0. We save Q-values in a Q-table. Therefore the Q-table is of size (number of possible states) times (number of possible actions).

Learning is very simple: every time we get a reward we update the Q-table using the following formula

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\Bigg(\underbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}}}_{\text{new value (temporal difference target)}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \Bigg)}^{\text{temporal difference}} \tag{2.1}$$

This equation does the following: we calculate the new Q-value estimate as the current reward plus discounted maximum Q value of the next state. Then we calculate the difference between it and the old Q-value and update the current estimate by this difference times learning rate.

Policy π_* is optimal when it maximizes: $R = \sum_{t=1}^{\inf} \gamma r_t$. Another way of looking into this is by Bellman optimality equation $V^{\pi_*}(s) = \max_a \{R(s, a) + \gamma \sum_{s'} P(s'|s, a) V^{\pi_*}(s')\}$. In this case value of the state while following π_* $V^{\pi_*}(s)$ is simply the biggest Q value for a given state s.

That's why we take the maximum Q value of the next state when we update the Q table, as

the value of the state is determined by the best possible action to take in this state and best actions in the subsequent states.

During learning, we may not always take currently best possible action to encourage exploration of the different actions. This is called off-policy learning as we do not use the final policy which chooses actions with the highest Q value.

Curse of Dimensionality refers to several problems that arise when dealing with high dimensions. In our case, it would be great if we could visit each state at least once to estimate its Q values. However number of possible states depends on their dimensionality. What's worse, this dependence is exponential. That's why it is important to lower the number of dimensions to make Q table smaller. There are many ways of lowering the number of dimensions and also other ways of making the Q table smaller, but the main idea is the smaller the Q table the easier learning, especially from a computational standpoint.

One of the ways to do it is by storing our policy as a deep neural network. This approach was used in [35] and is called deep-Q learning. It works in the following way: we have a neural network that takes the state as input and gives Q-values as an output. To train it, we create an experience replay. In it we store the agent experiences at each timestep, $e_t = (s_t, a_t, r_t, s_{t+1})$ pooled over many episodes into a replay memory. To update it we randomly sample some e_i to form a minibatch. Then we perform Q-learning updates and backpropagate changes in the deep-Q neural network.

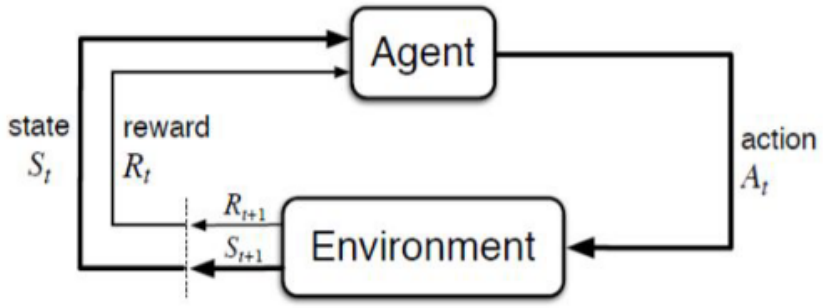

Figure 2.1: Interaction of the agent with the environment. source [54]

One of the more complex approaches in the field which tries to make learning faster is Hierarchical Reinforcement Learning [20]. It splits the initial problem into a hierarchy of subtasks such that higher-level agents use lower-level agents' tasks as if they were their actions. The advantage of hierarchical decomposition is a reduction in computational complexity if the overall problem can be represented more compactly and reusable subtasks learned independently. While the solution to an HRL problem may be optimal for hierarchy it may not be an optimal solution to the original problem. Fig. 2.2 shows an example of the usage of this approach from the paper.

The problem is as follows: our agent is a pawn and it can move to one adjacent square. Labyrinth is made out of four same-sized rooms with exits. Each room is 5 by 5. We want to get out of the labyrinth as fast as possible. To solve this maze we define two Agents: a high-level one that chooses the next room to go to and a low-level one that moves inside the room to get out of it in the direction specified by a high-level agent. Thanks to this approach we compress state-action space. In the traditional approach, we would have 400 (100 tiles and 4 actions possible on each tile) Q values, and in the new approach we have 8 (4 rooms total and two possible neighboring rooms) Q values for higher-level agents and 200 (25 tiles per room and two exits, 4 actions) Q values for the lower level agent.

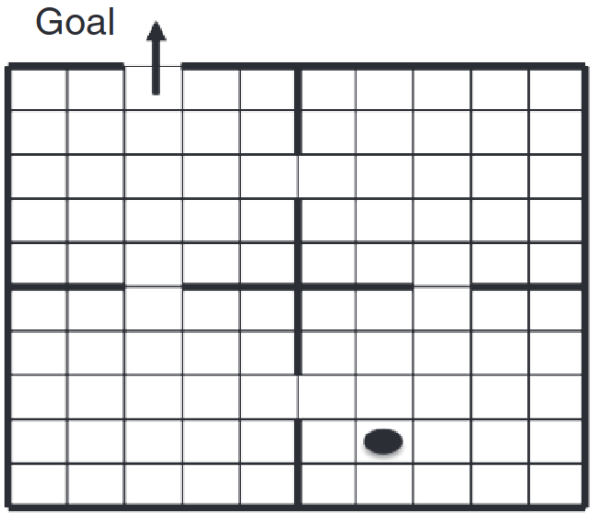

Figure 2.2: Maze problem for HRL agent. Image taken from [20]

There are other ways of dealing with computational complexity. One of them is feature selection. However, because some features can have non-trivial long-lasting implications it is not easy to choose a good feature subset. Paper [18] deals with this issue. The proposed framework adopts conditional mutual information between return and state-feature sequences as a feature selection criterion, allowing the evaluation of implicit state-reward dependency. The conditional mutual information is approximated with the least-squares method, which results in a computationally efficient feature selection procedure.

However, this is not the only way to deal with high computational complexity. Another approach was proposed in the paper [49]. To overcome this issue, instead of focusing on information in its raw visual form, methods for representing the semantic information embedded in the state were used. Language could represent complex scenarios and concepts, making it good for representation. Experiments in ViZDoom [60] environment suggest that language-based agents converge and perform better than vision-based ones.

The next section will cover more general approaches to dimensionality reduction.

2.2. Dimension Reduction

Dimension Reduction refers to techniques for reducing the number of input variables in training data. There are many approaches to it and many of them are described in the paper [33]. Fig. 2.3.

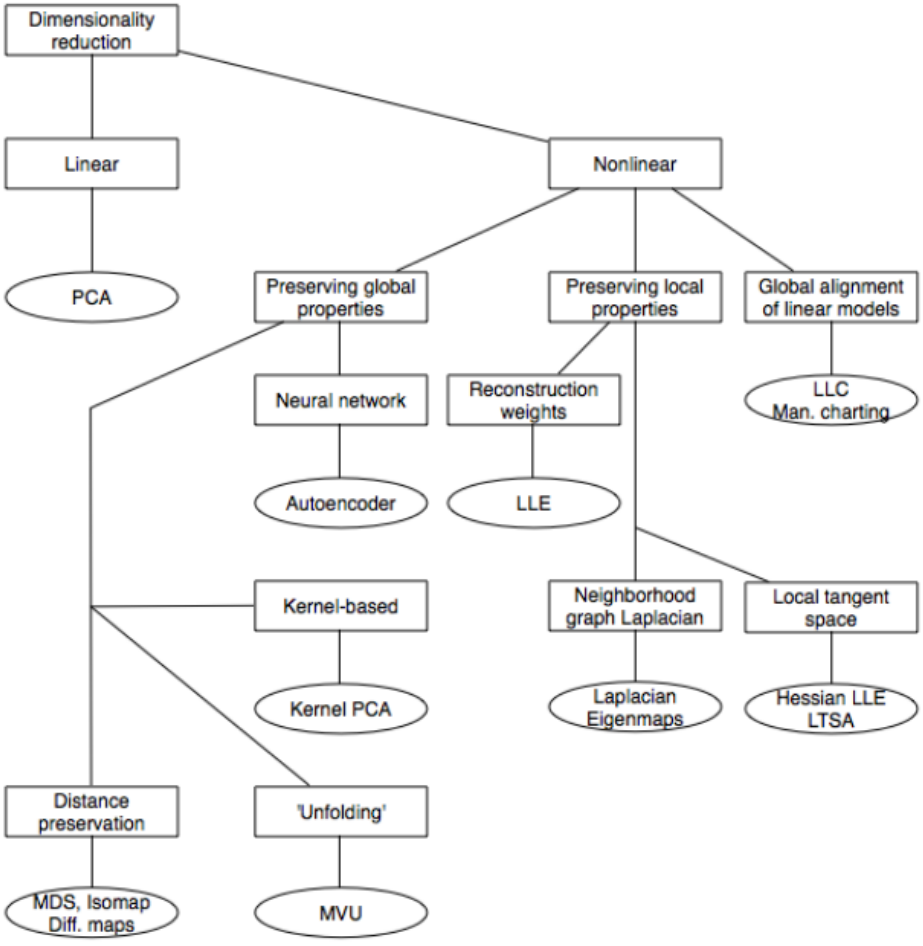

Figure 2.3: Taxonomy of dimensionality reduction techniques. Image taken from [33]

2.2.1. PCA

One of the simplest methods of dimensionality reduction is Principal Component Analysis (PCA) [34]. The goal of this method is to extract the most information (variance) from the data to represent it as a set of new orthogonal variables called principal components Fig. 2.4. PCA uses the eigen-decomposition of positive semi-definite matrices and the singular value decomposition

(SVD) of rectangular matrices, which is determined by eigenvectors and eigenvalues. To perform PCA: organize a data set as an $m \times n$ matrix, where m is the number of dimensions and n is the number of points. Subtract the mean for dimension or row x_i. Calculate the SVD. Then represent data using only a subset of the eigenvectors. PCA is often used for image compression.

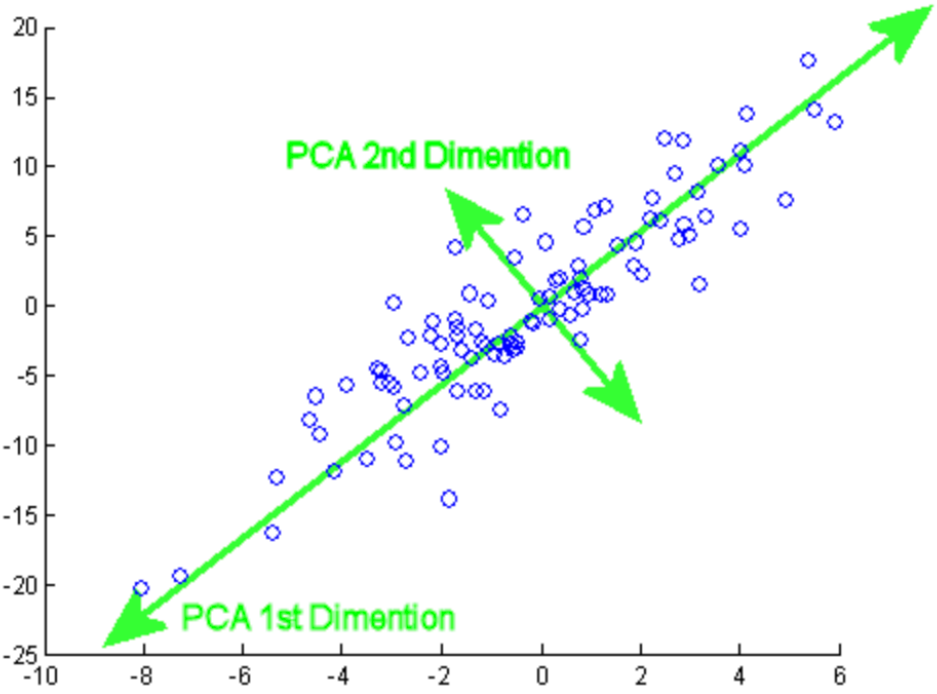

Figure 2.4: Example of the principal components in PCA. Image taken from [41]

2.3. Clustering

Clustering [37] is the process of identifying natural groupings or clusters within multidimensional data based on some similarity measure. We also often try to find the central point in each cluster and call such a point a centroid. Fig. 2.5 shows an example usage of the clustering.

Clustering is a fundamental process in many different disciplines. For example, in the paper [62] the authors use semantic similarity to cluster research articles and achieve as good results as using citations and their keywords.

Clustering can also be used for Image Segmentation. Image segmentation is finding logical partitions of the image. The paper [65] gives an overview of possible methods.

One of these methods is using the K-means clustering algorithm as was done in the paper [6].

Image Segmentation will be valuable for this work, as this is yet another way of reducing the

number of states by assigning them a cluster.

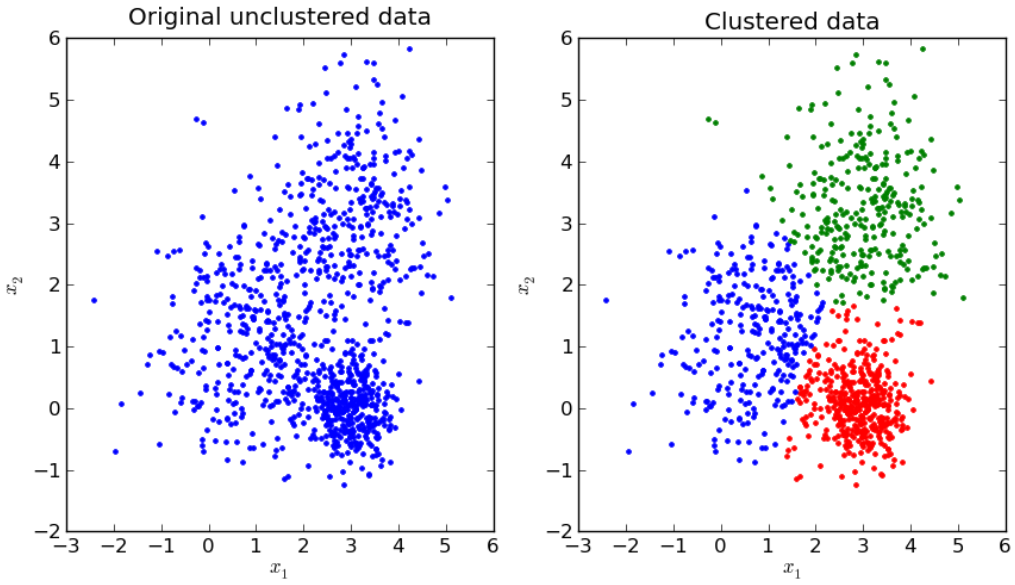

Figure 2.5: Example usage of the clustering

2.3.1. Similarity Measures

To define clusters we need to define distance metric to know what is close or far from the cluster.

The most basic metric is simply using the Euclidean distance between two points
$d(p_1, p_2) = \sqrt{\sum (p_{1,j} - p_{2,j})^2}$
There are more complex metrics. The paper [37] shows more possibilities.

2.3.2. Clustering Validation Indices

After we assign points to their clusters, we must validate quality of the proposed clusters. There are two main qualities we want from our clusters: compactness and separation. Compactness metrics measure to what extent points inside one cluster are similar to each other.

Example Compactness Measures:

- maximum distance between two points in cluster $diam(C) = max_{u,w \in C} d(u, w)$
- average distance between points in cluster $W(C) = \frac{1}{2|N|} sum_{u,w \in C} d(u, w)$
- average distance between points in cluster and centroid $W(C) = \frac{1}{2|N|} sum_{u \in C} d(u, c)$

Separation metrics measure to what extent points between clusters are dissimilar to each other.

Example Separation Measures:

- minimum distance between two points in different clusters

$$dist(C_1, C_2) = min_{u \in C_1, w \in C_2} d(u, w)$$

- average distance between points in cluster

$$W(C_1, C_2) = \frac{1}{2|N|} sum_{u \in C_1, w \in C_2} d(u, w)$$

- distance between centroids

$$W(C) = d(c_1, c_2)$$

It is worth noting that the goals of Compactness and Separation are at odds with each other. In the most extreme case, we can achieve perfect compactness if we give each point its own cluster. On the other side of the spectrum, we can achieve perfect Separation if we simply add all points to a single cluster. To balance these task there are metrics which balance these two qualities. One of the most classic indices is the Dunn index [10].

$$D = min_{i,j \in 1...K} \frac{dist(C_i, C_j)}{max_{a \in 1...K} diam(C_a)}$$

As we can see, the Dunn index tries to find the best ratio between compactness and separation.

2.3.3. Possible Clustering Technique

There are a lot of clustering techniques, but the most simple and famous one is K-means [23]. This algorithm tries to minimize average distances between cluster points and their centroids.

1. Randomly choose initial centroids $c_1, c_2, ..., c_K$ that represent different clusters.
2. For points p_i choose the nearest centroid among $c_1, c_2, ..., c_K$ and assign this point to cluster of the chosen centroid.
3. Create new centroids $c_1, c_2, ..., c_K$ as empirical means of the new clusters.
4. If no points changed their cluster, finish, otherwise go to the step 2 of the algorithm.

2.4. Ontology and Semantic Reasoning

Ontologies [16] are a means of formally modelling the structure of knowledge. This knowledge covers the entities and their relations. An example of such a system can be a company with all its

employees and their relationships Fig. 2.6. Using ontologies in science is nothing new. They have been used extensively in many domains like Medicine [48] or Mathematics [15] to systematize knowledge for humans and computers alike.

One of the most popular models in which data is stored in Resource Description Framework (RDF) [46]. RDF was originally developed to facilitate storage and interoperability of web resources by formalizing the usage of metadata about them. However, because of its flexibility, it is also used to describe other domains. The most basic data model consists of three object types:

1. **Resources**: All things being described by RDF expressions are called resources. A resource may be an organization like in Fig. 2.6. Resources are always named by URIs (Uniform Resource Identifier). Anything can have a URI and thanks to the extensibility of URIs any possible entity can have URI.

2. **Properties**: A property is a specific aspect, characteristic, attribute, or relation used to describe a resource. Each property has a specific meaning, defines its permitted values, the types of resources it can describe, and its relationship with other properties.

3. **Statements**: A specific resource together with a named property and the value of that property for that resource is a statement. These three individual parts of a statement are called, respectively, the subject, the predicate, and the object. The object of a statement can be another resource or it can be literal or other primitive datatype.

Let's look at an example from [57] (Fig. 2.6)

```
<http://example.com/people#ceo> org:headOf <http://example.com/org#id>.
```

This is an example of the statement. The first part (subject) of the statement `http://example.com/people#ceo` is a URI that refers to an actual person. The second part (predicate) is a property used to describe a relation. In this case, it is org:headOf which symbolizes being the boss of the organization. org: is a namespace of the relation. Namespace refers to the ontology from which we take given property. Because multiple premade ontologies can be used to describe our own, to avoid name clashes we add prefixes to their properties. In this case, org refers to this organizational ontology [57]. The third part (object) `http://example.com/org#id` is an URI of an organization in which the previously mentioned person is the head of.

RDF can be implemented in many different serialization formats. Example of such formats are Turtle [56] and RDF/XML [44].

We can also query our database using SPARQL [53]. Lets say that we want to find all bosses of organization with URI `http://example.com/org#id`. We can do that with following query.

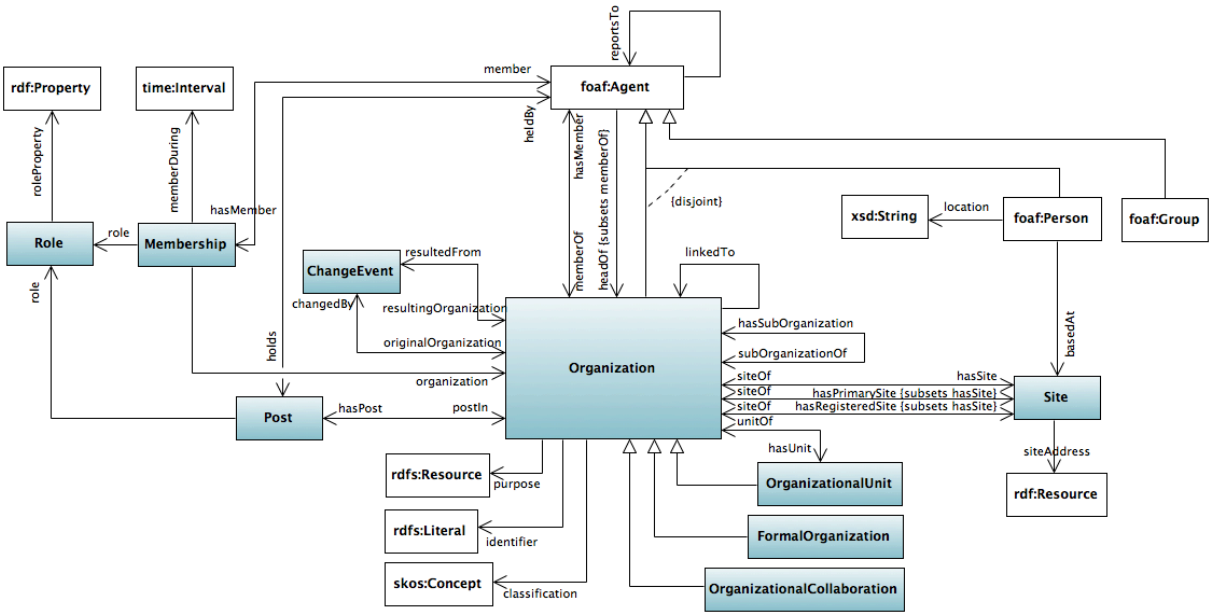

Figure 2.6: Example of Ontology. Source [57]

```
SELECT ?ceo
WHERE
{
  ?ceo org:headOf <http://example.com/org#id>
}
```

This query tries to match resources under variable ?ceo such that they have property org:headOf with resource <http://example.com/org#id>.

Many different tools have been developed for working with ontologies. One such tool is Protege [42]. This framework allows for both ontology creation and using a reasoner.

An example of the usage of ontology is the paper [31]. In this paper, ontology of animals was used for zero-shot classification. Based on the ontology, discriminative rules were developed and each rule was getting scored based on its quality using reinforcement learning. Then new unseen animals were shown, information about their attributes given for example ”Crocodile is a reptile which can swim, is green, has big teeth” and then animals were classified. Thanks to the usage of ontology better results were achieved.

Ontologies are not only a means to store and retrieve knowledge. It is also possible to perform reasoning on it.

Semantic reasoning allows us to figure out implicit knowledge from rules and facts in our ontology. For example, let us take the following rule ”Father of a father is grandfather” and two facts ”Adam is a father of Ben” and ”Ben is a father of Charles”. From this, we can infer that

"Adam is the grandfather of Charles". Of course, more advanced rules and facts can be used.

Description logics (DLs) are a family of knowledge representation languages that are used in ontological modeling. DLs are logic (usually decidable subsets of first-order logic), and they have formal semantics: a precise specification of the meaning of DL ontologies. This formal logics makes it possible to exchange DL ontologies without ambiguity and also to use logical deduction to infer additional information from the facts stated explicitly in an ontology and to validate its correctness.

The paper [27] discusses description logic in more detail, as well as differences between logics, as the more expressive ones are also more computationally expensive.

DLs are important underpinnings for the OWL Web Ontology Language [63]. OWL is an extension of RDF as it allows greater expressiveness. Its main feature is support for more logical characteristics of properties like symmetry or disjointness. OWL has three sublanguages with varying expressiveness.

- **OWL Lite**: supports primarily a classification hierarchy and simple constraints. Owl Lite has a lower complexity compared to OWL DL.

- **OWL DL**: gives maximum expressiveness while retaining computational completeness and decidability. OWL DL includes all OWL language constructs, but their use is restricted. OWL DL is named after description logics that form the formal foundation of OWL.

- **OWL Full**: is meant for maximum expressiveness and the syntactic freedom of RDF, but it gives no computational guarantees. As an example, in OWL Full a class can be treated simultaneously as a collection of individuals and as an individual in its own right. OWL Full also allows an ontology to augment the meaning of the pre-defined vocabulary.

One of the possible implementations of a reasoner which supports Protege [42] is Hermit [13]. Hermit takes an OWL file of an ontology and can determine whether or not the ontology is consistent, identify subsumption relationships between classes, and much more. It also is very fast for most of the ontologies.

Semantic reasoning has real-life uses. For example, in the paper [3] this technology was used to improve the recommender system. Traditional content-based recommenders suggest items similar to those the user liked in the past. Such mechanisms lead to recommending only items that are similar to those the user already knows. Novel strategy diversified the offered recommendations by employing reasoning mechanisms. These mechanisms discovered extra knowledge about the user's preferences leading to significant increases in recommendation accuracy.

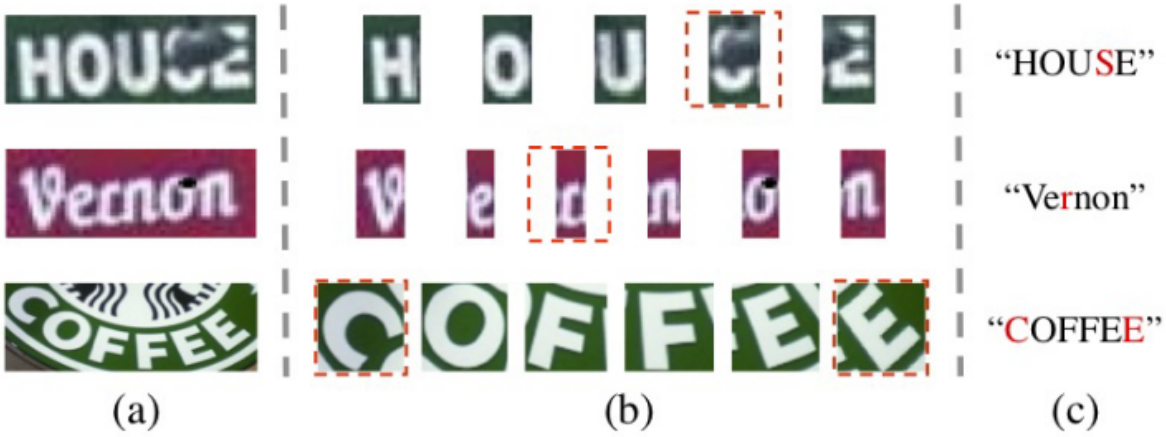

Figure 2.7: Examples of text in the wild. (a) are some difficult scene text images, (b) are individual characters extracted separately from (a), and (c) are the corresponding semantic word contents. The characters with red dashed boxes in (b) are easy to misclassify based only on visual features. Image taken from [64]

Another example is the paper [64]. Scene text image contains two levels of content: visual and semantic information. To make better use of semantic information, a new framework named semantic reasoning network (SRN) was developed. The method achieved better results on multiple benchmarks. The SRN was also faster than the traditional RNN based methods. Fig. 2.1 shows an example of challenges faced by SRN.

It is also possible to create ontology from scratch. The paper [58] developed autonomously symbolic planning operators using continuous interaction experience. Creation of the symbolic knowledge in this paper was achieved in two stages. In the first stage, the robot explored the environment by executing actions on single objects, formed effect and object categories, and gained the ability to predict the object/effect categories from the visual properties of the objects by learning the complex relations among them. In the next stage, with further interactions that involved stacking actions on pairs of objects, the system learned logical high-level rules. Finally, these categories and rules were encoded in Planning Domain Definition Language (PDDL) [12], enabling symbolic planning. Learning was realized in a physics-based simulator. This method rules were used to build tower using the real robot.

The paper [28] explores in more depth usages of semantic reasoning and most importantly showcases tools and processes to transform ontology to the vector which can be a more useful form for Reinforcement Agent. It also explores the concept of semantic similarity which can be used as a distance function between states.

This thesis is about Semantic Reasoning because reasoning allows for the creation of plans

like in [59]. Plans allow one to assess the states without actually visiting them and finding the best one. Let's say that our agent is making a cup of tea. To make the tea we need to put hot water into a cup and then put a teabag into it. If the agent understands what hot means then it won't try to make tea by pouring cold water into a cup. There are three main problems with this approach: computational cost if the ontology is too complex, the problem of creating an ontology for an agent to use, and executing the plan in terms of low-level actions.

2.5. Grounding

Grounding is transforming high level, semantic goals to low level ones which can be followed. As an example, a semantic goal could be "pick up block", and a low level goal after transformation is an altitude of the block. This example is from the subsection Mapping Predicates to Low-Level Subgoals. The papers [55] and [32] give an overview of using natural language grounding and other methods in the field.

One of the most recent approaches can be found in the paper [19]. The authors investigated the use of natural language to drive the generalization of control policies and introduced the new multi-task environment MESSENGER with text manuals describing the environment. MESSENGER does not start with knowledge connecting text and state observations, the control policy must ground the game manual to entity symbols and dynamics in the environment. A new model, EMMA (Entity Mapper with Multi-modal Attention) uses an attention module that allows to focus over relevant descriptions in the manual for each entity in the environment. EMMA learns a latent grounding of entities and dynamics from text to observations using environment rewards. EMMA achieved successful zero-shot generalization to unseen games with new dynamics. However, the win rate on the hardest stage of MESSENGER remains low. A possible improvement could be those text manuals were not formalized ontology and they had to be written by humans.

Another novel approach was presented by [2]. The main interest was the autonomous acquisition of a multitude of skills. Language-conditioned reinforcement learning (LC-RL) approaches are used as they allow for an expression of abstract goals as sets of constraints on the states. However, most LC-RL agents are not autonomous and require additional instructions. Their direct language condition strongly limits the expression of behavioral diversity. To resolve these issues, a new conceptual approach was developed the Language-Goal-Behavior architecture (LGB). LGB separates learning and language grounding via an semantic representation of the environment. DECSTR is an implementation of LGB framework as an intrinsically motivated learning agent

given a semantic representation describing spatial relations between objects. In the first stage (G to B), it explores its environment and creates semantic configurations. In the second stage (L to G), it trains a language-conditioned goal generator to generate semantic goals that match the constraints expressed in language-based inputs. Intermediate semantic representations helped in many ways.

2.6. Most Inspiring papers

In this section we will cover in more detail two papers that were important in creating this thesis and understanding them even in a cursory manner will be very helpful.

2.6.1. World Models

The paper [17] explore building a generative neural network model for reinforcement learning. World models can be trained quickly in an unsupervised manner to learn a compressed spatial and temporal representation of the environment. By using features extracted from the world model as inputs to an agent it can train a very compact policy. It is even possible to train an agent entirely inside of its own simulated environment using its world model, and transfer this policy back into the actual environment.

The whole architecture is made of three parts: Variational Autoencoder (V), Recurrent Neural Network with a Mixture Density Network output layer MDN-RNN (M), Controller (C).

Variational Autoencoder

The environment provides an agent with a high dimensional input observation at each time step. The V model is giving a compressed representation of each observed input. Here, a Variational Autoencoder as V model is used to compress each frame into a smaller latent vector z.

Recurrent Neural Network with a Mixture Density Network

Compression of what happens over time is also needed. The M model serves as a predictive model of the future z vectors that V is expected to produce. Since many complex environments are probabilistic, RNN outputs a probability density function $p(z)$ instead of a deterministic prediction of z.

Controller

The Controller (C) model is responsible for choosing the actions which maximize the expected cumulative reward of the agent. In the experiments C was trained separately from V and M, so that the world model (V and M) would be most complex. C was a single layer linear model that maps z_t and h_t directly to action a_t at each time step: $a_t = W_c[z_t h_t] + b_c$ In this linear model, W_c and b_c are the weight matrix and bias vector that maps the concatenated input vector $[z_t h_t]$ to the output action vector a_t.

Final model

Finally, all components are combined as shown in Fig. 2.8. The observation is first processed by V at each time step t to produce z_t . The input into C is this latent vector z_t concatenated with M's hidden state h_t at each time step. C will then output an action vector a_t and affect the environment. M will then take the current z_t and action a_t as an input to update its own hidden state to produce h_{t+1} to be used at time $t + 1$.

Final training

Now it was possible to teach the agent in following way:

1. Collect 10,000 rollouts from a random policy.
2. Train VAE (V) to encode each frame into a latent vector $z \in R^6 4$, and use V to convert the images collected from (1) into the latent space representation.
3. Train MDN-RNN (M) to model $P(z_{t+1}, d_{t+1}|a_t, z_t, h_t)$.
4. Define Controller (C) as $a_t = W_c[z_t h_t] + b_c$.
5. Use CMA-ES (Covariance-Matrix Adaptation Evolution Strategy) to solve for a W_c that maximizes the expected survival time inside the virtual environment.
6. Use learned policy from (5) on actual environment.

Results

Thanks to this approach, the Agent was able to get state-of-the-art results. This proved that using better environment representation (world model) has a good impact on results. Also, an agent could be trained in a virtual environment which for some applications may be valuable as testing in the real world is expensive.

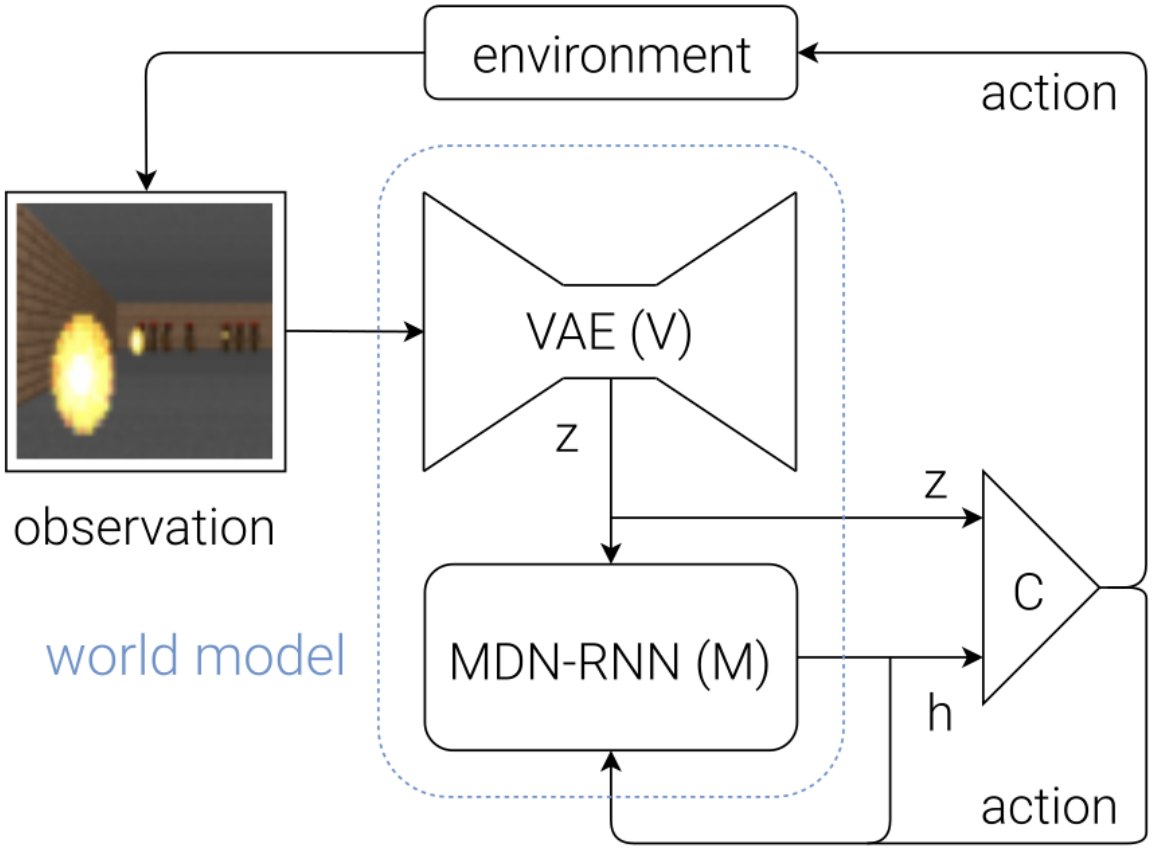

Figure 2.8: Diagram of Agent model from World Models [17]. The observation is first processed by V at each time step t to produce z_t . The input into C is this latent vector z_t concatenated with M's hidden state h_t at each time step. C will then output an action vector a_t and affect the environment. M will then take the current z_t and action a_t as an input to update its own hidden state to produce h_{t+1} to be used at time $t + 1$.

However, the results were not perfect. For example, the VAE encoded unimportant detailed brick tile patterns because it did not know what was important for the task. Also, C module was very simple and was not using any rule-based reasoning.

2.6.2. From Semantics to Execution: Integrating Action Planning With Reinforcement Learning for Robotic Causal Problem-Solving

Reinforcement learning is an appropriate method to learn robot control. Symbolic action planning is useful to resolve causal dependencies and to break a causally complex problem down into a simpler sequence of high-level actions. Problem is that action planning is based on discrete high-level action and state spaces, whereas reinforcement learning is often driven by continuous spaces. The paper [11] is solving a problem with the integration of both approaches to deal with the block stacking challenge.

This involves the grounding of the high-level representations to low-level subgoals, abstraction of the low-level space to the high-level space, reinforcement learner to achieve the low-level subgoals, and the integration of an action planner with the reinforcement learning using the abstraction and grounding mechanisms as shown in Fig. 2.9.

Predicates and planning

Planning is done on predicates. Predicate of the planning domain determines a Boolean property of one object in the environment. The set of all predicates is denoted as $P = \{p_1, ..., p_n\}$. The high-level world state S is defined as the set of all predicates which can be positive or negative.

The high-level action space consists of a set of grounded STRIPS operators a that are precondition and effect literals.

Mapping Predicates to Low-Level Subgoals

After planning we need to map high-level goals to low-level subgoals. This is realized with a set of functions f_{subg} that we define manually for each predicate p. For a given predicate p, the function f_{subg} generates the low-level state s_p that determines p, based on the current state s and the high-level goal g: $f^p_{subg}(s, g) = s_p$

To illustrate how f_{subg} can be implemented, consider the following example from the paper [11]. For a block-stacking task a predicate (at_target o1) which indicates if an object is at a given goal location on the surface (at_target o1). Then the respective function $f_{subg}(s, g)$ can be implemented as follows: $f^{(at\_target o1)}_{subg}(s, g) = [g[0], g[1], g[2]]$ In this case, the function gets the

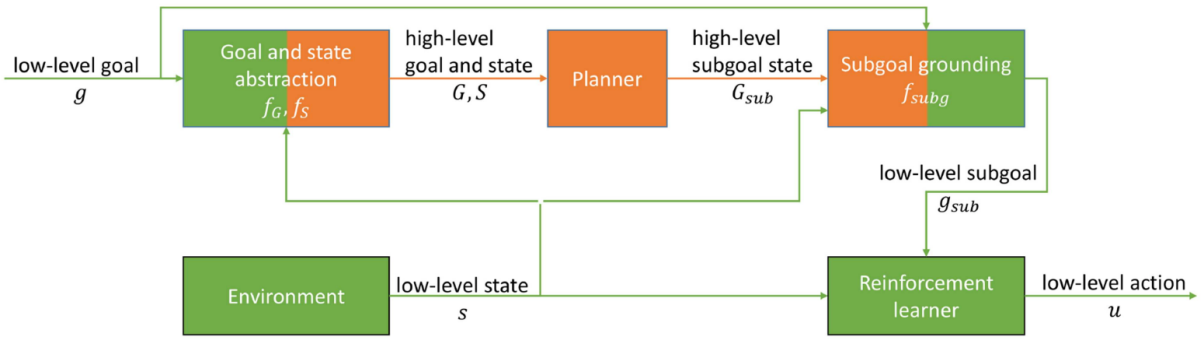

Figure 2.9: Proposed integration model [11]. Low-level planning elements are green and high-level elements are orange. The abstraction functions f_S , f_G map the low-level state and goal representations s, g to high-level state and goal representations S, G. Then they are used as an input for the planner which calculates a high-level subgoal state G_{sub} . The subgoal grounding function f_{subg} maps G_{sub} to a low-level subgoal g_{sub} under consideration of the current low-level state s and the low-level goal g. The reinforcement agent train based on the low-level subgoal g_{sub} and the low-level state s.

target coordinates for the object $o1$ from g and does not require any information from s.

Results

While the agent initially learns more slowly, in the end, it achieves better results, especially in a noisy environment where image is randomly blurred. This also proves that mapping between high and low-level goals and planning is possible and beneficial. However this work has some issues. First of all functions f_{subg} require manual definitions for each predicate p. Another disadvantage is that the domain knowledge must be hand-engineered.

3. Proposed solution

In this chapter we will explain the method used in this thesis. Each of the subsections will cover parts of the overall architecture (see again Fig. 1.1) covered in section Proposed Solution high level overview.

3.1. Training environment

To train any kind of agent we need some environment to test it. As this thesis is not solving any domain-specific problem, we need an artificial environment. It would also be good if we could compare the results of our approach with those of other works. OpenAI Gym [38] allows us to set up many different environments quickly and many papers cited here used environments from there. OpenAI enables the simulation of many Atari games, classic control tasks like pool balancing, and even an entire robot arm simulator physics engine.

Another popular environment collection is ViZDoom [60]. This video game simulator allows for the creation of custom scenarios for an agent to train on. It also was used in multiple papers cited here.

Both environment collections support Python programming language [43].

However, because Reinforcement Learning is very resource intensive, we will test our solution on "PongNoFrameskip-v4" environment from OpenAI Gym. We choose Pong as it is a very simple game with uniform background and objects with simple shapes. In this classic Atari game, we have a ball and two paddles. Points are scored when the ball touches the left or right side of the screen. The ball can bounce off walls and paddles. Our agent is controlling the paddle. There are 6 possible actions in the environment: NOOP, FIRE, RIGHT, LEFT, RIGHTFIRE, LEFTFIRE. However, because NOOP and FIRE are wait actions, LEFT is the same as LEFTFIRE and RIGHT is the same as RIGHTFIRE we limit possible actions to RIGHT and LEFT which makes the paddle go up and down respectively. Agent receives +1 reward when it scores a point and -1 reward when the opponent scores a point. The game ends when either side scores 21 points total. This game has also another very important feature. In this paper,

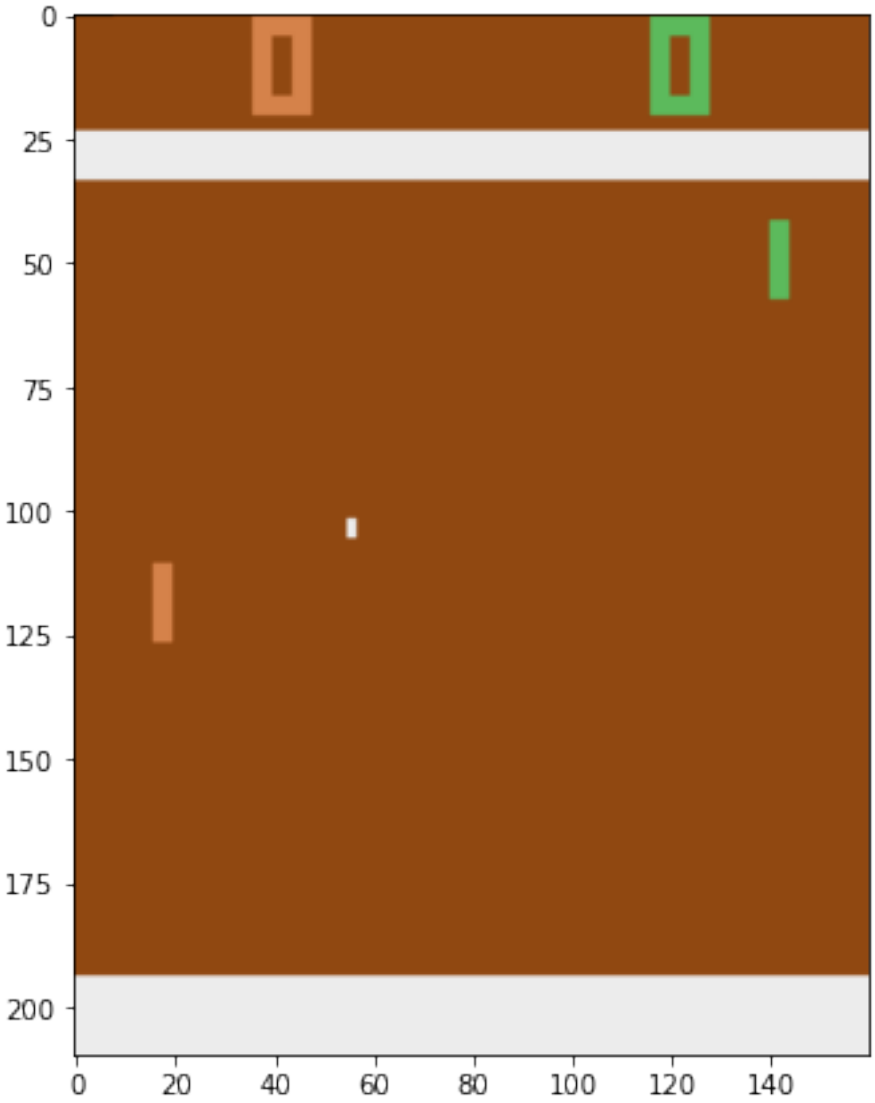

Figure 3.1: Example screenshot of Pong environment from OpenAI Gym.

we will try to create good semantic embedding so it would be good if we have something to compare against. In OpenAI Gym we can get our state either as a 210x160 image (Fig. 3.1) or as 128 bytes of the RAM of the game. From the RAM of Pong, we can directly extract the y position of both paddles and the x and y position of the ball. This contains all the information needed to play the game.

3.2. Dimension Reduction and Clustering

3.2.1. Object detection

The first step for creating meaningful ontology is to find objects to reason about. Our field brings two main problems: first, we do not have explicit labels so our approach needs to be unsupervised, and second, it needs to be incremental as we can not and do not need to load all possible states into memory. It also needs to be reasonably fast, as RL is very resource-intensive. Because object segmentation can be costly, we will take advantage of the visual simplicity of Atari games.

To speed up calculations, we use frame-skipping [24] equal to 4. That means we are only giving our agent every 4th frame and its chosen action is repeated in between those frames.

We first rescale the image to 84x84 by using inter-area interpolation (resampling using pixel area relation) and then we convert it to grayscale. Then we use Otsu's thresholding [39]. This method separates the image into foreground and background. It does it by finding a threshold and assigning all pixel values above it to one class and below to another. This threshold maximizes variance between classes. Because backgrounds are uniform, this method works well. However, we do not recalculate this value for each frame and we lower this threshold by 10%. In this way, if an object was detected in one frame, it will be detected in the next frame, and in edge cases, where it is not clear if something should be background or not, we would rather detect more objects than less.

After that, we take connected components which are groups of pixels that belong to the foreground and are neighbors of each other. We treat them as a singular objects (see Fig. 3.3).

However, as we can see, some of the objects (the wall on the bottom) are very large. To fix that, we cut objects in the following way: we first define the maximum space an object can take. We use one-tenth of the image both horizontally and vertically. If the object is bigger than that we cut it into as few areas as possible so each of the objects created is of the same size and within bounds predefined earlier (see Fig. 3.4). If we need to cut an object both horizontally and vertically then we first cut it horizontally then we cut new objects vertically if needed.

3.2.2. Object classification

After extracting objects we want to cluster them into different classes of objects. To do that we first use Incremental PCA on the batch of objects extracted from the frame to lower their dimensions as they are still pretty big (8*8=64 dimensions total for an individual object). This reduces the number of dimensions to 10; as we will see later, this is enough to represent almost

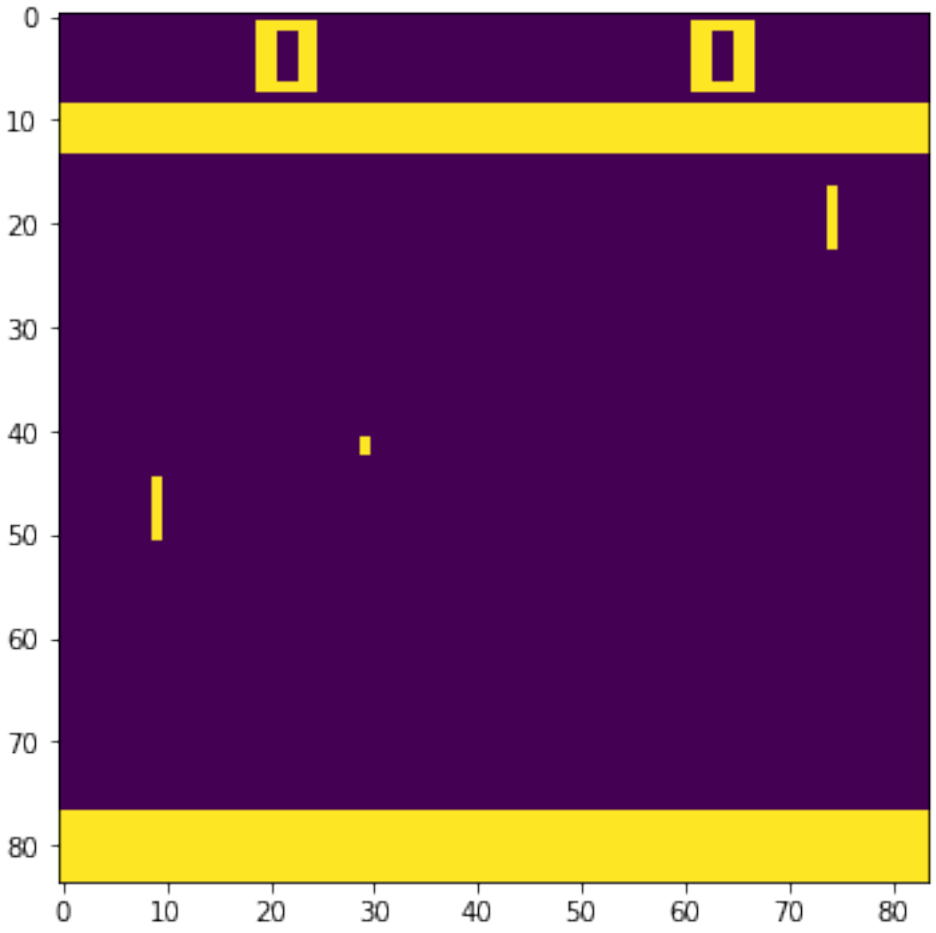

Figure 3.2: Image preprocessed by rescaling, grayscaling and otsu thresholding

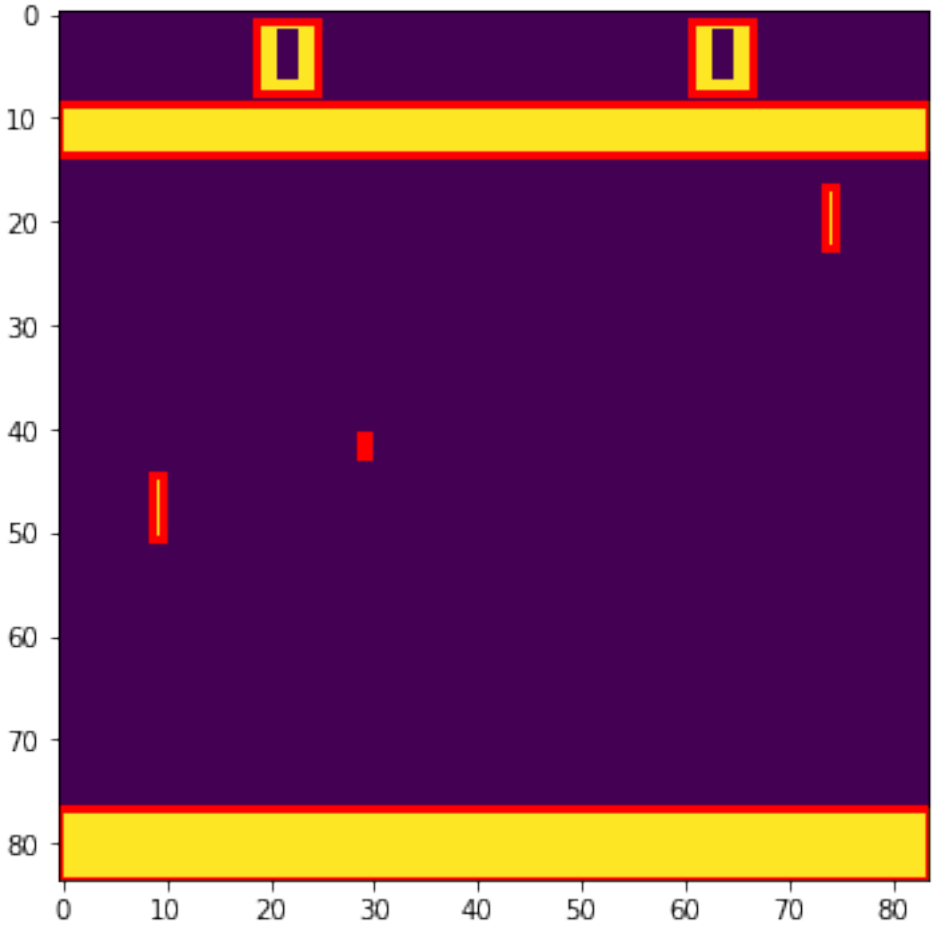

Figure 3.3: Found objects by taking connected components of 3.2

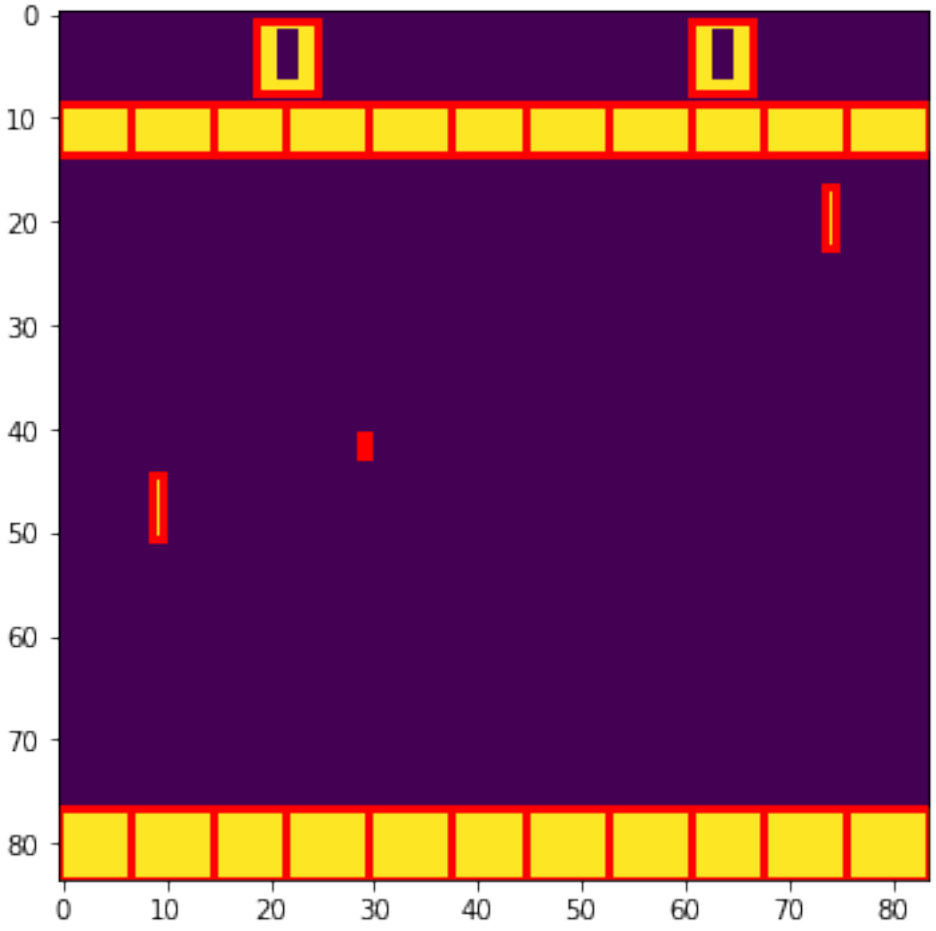

Figure 3.4: Found objects after further cutting 3.3

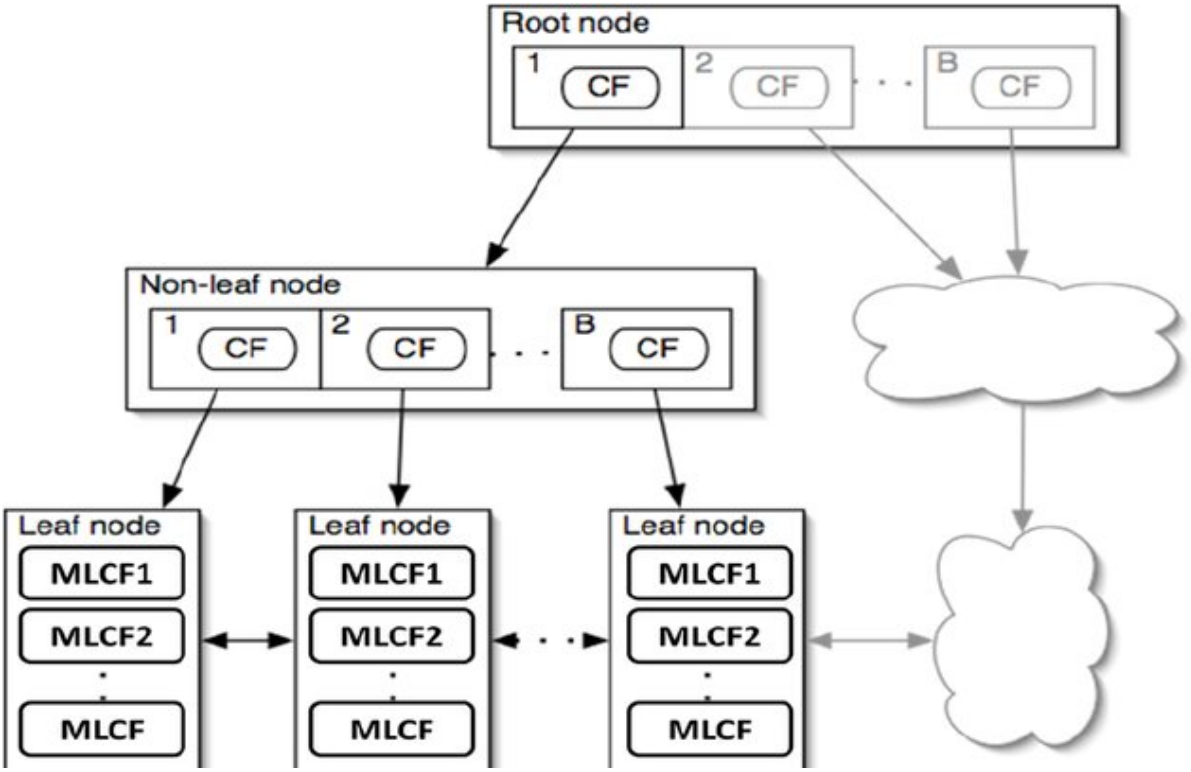

Figure 3.5: Structure of BIRCH ([22])

all variation.

Then we apply Birch Clustering [66]. It works in the following way: It constructs a tree data structure (see Fig. 3.5) with the cluster centroids being leafs. The structure is made of nodes with each node having many subclusters. The branching factor is the maximum number of subclusters in a node. Each subcluster maintains a linear sum, squared sum, and the number of samples in that subcluster. Subcluster can also have another node as its child, unless the subcluster is a leaf node.

When a new point enters the root, it is merged with the subcluster closest to it and the linear sum, squared sum and the number of samples of that subcluster are updated. This is done recursively till the leaf node are updated.

A leaf nodes can be either the final cluster centroids or can be further clustered as input for another clustering algorithm.

This algorithm has two major advantages compared to k-means: first, it is incremental, and second, it suggests the number of clusters as it can simply be the number of leaves.

We use the following parameters for Birch tree for object clustering:

1. threshold: The maximum radius of the subcluster obtained by merging a new sample and the closest subcluster, otherwise a new subcluster is started. Value: 0.5.

2. branching factor: Maximum number of CF subclusters in each node. If a new samples enters such that the number of subclusters exceed the branching factor then that node is split into two nodes with the subclusters redistributed in each. The parent subcluster of that node is removed and two new subclusters are added as parents of the 2 split nodes. Value: 50.

3. maximum number of clusters: If the number of leaves exceeds maximum number of clusters, then we perform agglomerative clustering on subclusters centers using ward linkage and final number of clusters being equal to maximum number of clusters. Otherwise we use leaf nodes directly for classification. Value: 10.

3.2.3. Semantic embedding

After object detection and clustering we transform our frame into a graph. First, we extract the following information for each object on the frame: its object class predicted by Birch, the x position of its center, y position of its center. Then each of the found objects becomes the node of the graph and we connect all nodes which are close enough (this is determined by the proximity parameter) on the frame by using euclidean distance from their centers and edge weights are these distances (see 3.6). However, we are unable to give our Deep Q net a graph directly. So we must use whole graph embedding [5].

We use Karate Club [47] implementation of graph2vec [36] semantic embedding method after modifications. It works like this: we first transform our graph using Weisfeiler-Lehman hashing [26] [21] using object class as labels. Weisfeiler-Lehman hashing works in the following way (see Fig. 3.7). For each node, we look at its label and the labels of its neighbors and we give this combination a unique hash. This hash then becomes a new label and the process is repeated.

Then we treat those hashes as words and create a word document. We also use other information compared to the original graph2vec so the created document has the following format depending on the algorithm used. For distributed memory (PV-DM) each node is a sentence containing: its object class, x position, approximate x position (x divided by 5), approximate y position (y divided by 5), and Weisfeiler-Lehman hashes. For the distributed bag of words (PV-DBOW) we also add information regarding embedding concatenated with its class label, x position, and y position, it is because the bag of words does not look at the order of the words so we do this concatenation to differentiate between the position of the objects depending on their class, and in case of multiple objects of the same kind their mirror positions.

Example document representation of an object: ’8’, ’pos_x4’, ’pos_y21’, ’apr_pos_x0’, ’apr_pos_y4’, ’8pos_x4’, ’8pos_y21’, ’8apr_pos_x0’, ’8apr_pos_y4’,

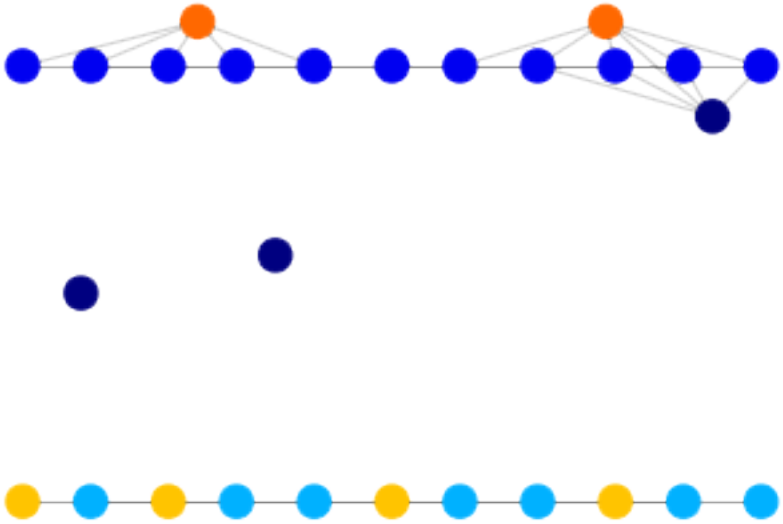

Figure 3.6: Graph created from Fig. 3.4 colors of nodes symbolize different classes of objects.

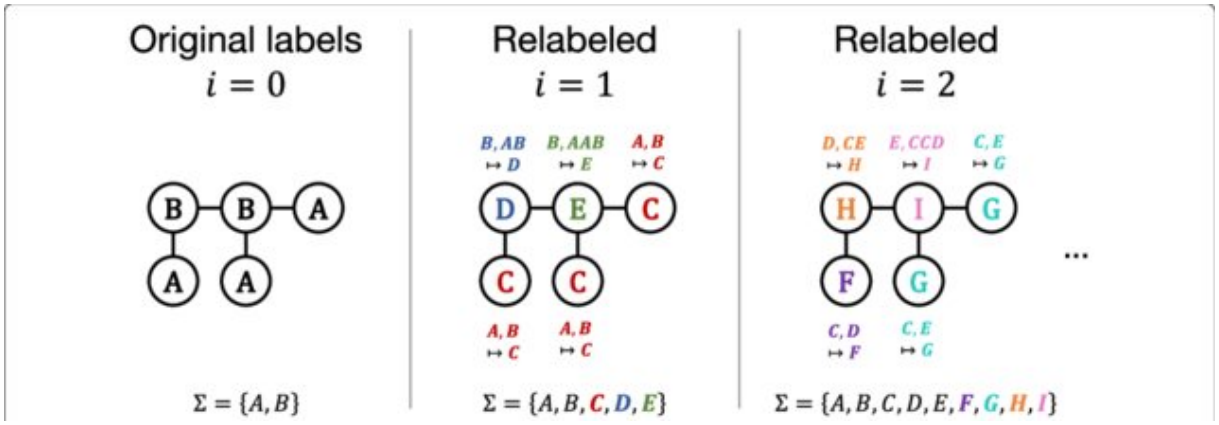

Figure 3.7: Weisfeiler-Lehman hashing explanation. Each combination of labels of neighbours of the node becomes new kind of label and the process is repeated. Image taken from [26]

'8apr_pos_x0apr_pos_y4', '8pos_x48pos_y21', 'apr_pos_x0apr_pos_y4', 'pos_x48pos_y21', 'dab5edacccee84e7f5d123489f3e151c', '4975d4c3aeda67b432297ab45ea7e3f8'.

Then we train our language gensim model. However during implementation it turned out that graph2vec which is build on Gensim [45] implementation of doc2vec [29] DOES NOT support incremental learning (see github issues [50], [9]). That forced us to basically make a training data set first and train later.

Before we will use embedding techniques we first train PCA and Birch in the following fashion.

First, we only train PCA until it reaches stability. When training PCA we only take unique object sets from each frame to try to reconstruct rare objects better. We define stability in the following way: for a given object collection from a frame, we compare its PCA transformation with a new transformation after training PCA for 250 batches. If the mean absolute difference between transformations is less than 0.05 then we stop training it further.

A similar method is then applied to Birch. For the given object collection from the frame we compare its Birch labels with new labels after training Birch for 250 batches. If balanced accuracy is greater than 95%, with new labels being predictions and old labels to compare against, then we stop training the model further.

Thanks to this we achieve two things: first we take care of concept drift as when PCA was still being improved Birch was creating multiple copies of clusters for the same objects because transformations of PCA were changing. The same story was for interactions between doc2vec and birch. When labels of Birch were still swapping places it caused doc2vec to embed the same document in different ways and caused its vocabulary to become bigger as it had to account for a much greater variety of Weisfeiler-Lehman hashes.

And second thing is that we also save computational resources because we stop training PCA and Birch after they reach stability.

When setting up PCA, Birch and doc2vec, our agent is set to doing random actions to maximize exploration and because agent can not receive any meaningful state yet.

Parameters which we will run doc2vec model on:

1. proximity: How close nodes must be to have an edge between them. Value: 20.

2. number of documents: Number of documents to accumulate before starting training of doc2vec model. Value: 30000.

3. embedding dimension: Size of embedding dimension Value: 128.

4. minimum count: Minimum number of occurrences of given word before it can be used by an model. Value: 2.

5. epochs: Number of epochs used for training and inferring embedding. Value: 20.

6. learning rate: Learning rate used by an model. Value: 0.025.

7. window size: The maximum distance between the current and predicted word within a sentence. Value: 16.

8. hierarchical softmax: If 1 this algorithm will use hierarchical softmax. Value: 1.

9. DM: Defines the training algorithm. If DM=1, distributed memory (PV-DM) is used. Otherwise, distributed bag of words (PV-DBOW) is employed. We will test both algorithms.

10. negative: number of "noise words" negative sampling will use. If set to 0, no negative sampling is used. Value: 0.

11. max vocabulary size: Maximum size of vocabulary, if to be exceeded least frequent words will be dropped. Value: 30000.

The model creates embedding and this embedding is an input to the Deep Q-model.

3.3. Reinforcement Learning Agent

We use Torch [40] implementation of the Deep-Q learning [35]. We will be using the following architecture for this model 3.8. It is composed of three convolutional layers and one dense layer. All activation functions are ReLU [1]. For all convolutional layers, their strides and kernels are the same sizes in both dimensions for a given layer. The input Layer consists of an 84x84x2 image. The first channel represents the current state and the second channel difference between the current and previous state. The first convolutional layer has 32 channels with kernel size 8 and stride 4. The second convolutional layer has 64 channels with kernel size 8 and stride 4. The third convolutional layer has 64 channels with kernel size 3 and stride 1. The fourth layer is a dense layer made of 256 neurons. The last output layer has a size equal to the number of possible actions in our case 2. When using embedding we will be using only a dense layer.

Other learning parameters which will be constant for all the models are:

1. total frames processed: 2000000

2. exploration rate (ϵ): maximum at 1 and minimum at 0.01 after 30000 frames with exponential decay.

3. learning rate: 0.0001 with Adam [25] as gradient descent method with $\beta_1 = 0.99$ and $\beta_2 = 0.999$
4. mini-batch size: 32
5. replay buffer size: 100000
6. initial buffer size: 10000
7. reward discount (γ): 0.99

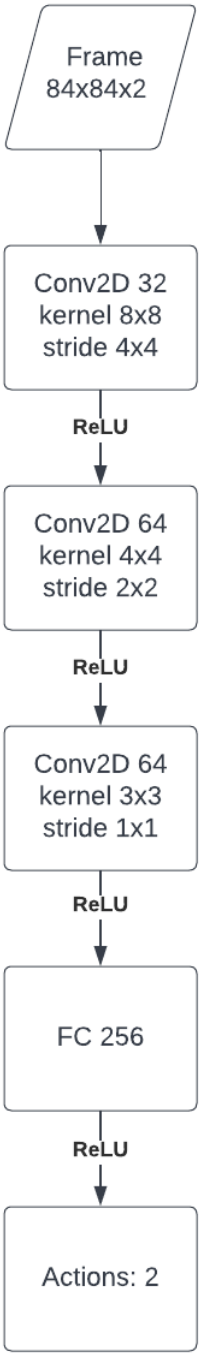

Figure 3.8: Architecture of our Torch agent policy model

3.4. Ontology, Planner and Grounding

3.4.1. State map graph

For our ontology and reasoning, we will be using the graph. It turned out that using it was much easier and faster to use than descriptive logic and RDF. The main problems turned out to be storing data and making recurrent queries. Another issue was explaining the dynamics of the game semantically. While calling our graph ontology may be a bit of a stretch it still holds information regarding state dynamics and we could represent this graph as a collection of RDF triples if we wanted.

This graph will be directed. Its nodes are states and its edges are transitions between them. Transition is made of starting state, an ending state, action, and reward. However, it would be unfeasible to store all of this directly as both storage and further reasoning would be expensive. That's why we cluster nodes and edges using two Birch trees one for nodes and the other for edges.

Then a graph is made from centroids of these clusters.

Parameters of node Birch trees will be

1. threshold: 0.011*D where D is dimension of embedding.
2. branching factor: 100.
3. maximum number of clusters: no limit

Parameters of edge Birch trees will be

1. threshold: 0.015*D where D is dimension of embedding.
2. branching factor: 100.
3. maximum number of clusters: no limit

Because this graph will be a rough sketch of state dynamics we also train a model for predicting the next state given the current state and chosen action. That's because centroids are not "real" states which occurred so we need to know with greater accuracy where the agent will go after choosing given action. For the same reason, we also train the model for predicting rewards given the current state and action. That way we can have both short and long-term planning.

3.4.2. Long-term planning

For long-term planning, we will be calculating the path following which we will get the greatest reward. This problem can be reformulated as the shortest path problem with weighted edges. To do that, weights must correspond to inverted rewards. So positive rewards are negative and vice versa.

The existence of negative weights introduces a few problems. First is that Dijkstra [8] algorithm for finding shortest paths does not work when we have negative edges. Other algorithms like Bellman-Ford [61] can work with negative edges, however they still fail when there are negative cycles in the graph. We cannot assure that such a negative cycle will not emerge, what's more, in the case of perfect policy, we even expect it to appear. To handle these issues, we will be setting neutral rewards to 1, positive rewards to 0, and negative rewards to 100, and instead of finding the overall best path, we will be finding the closest states with positive rewards. Our algorithm for finding this path will be Dijkstra. We will be using a cutoff equal to 100 so paths longer than 100 will be ignored. In this way we will be avoiding negative rewards.

3.4.3. Short-term planning

For state prediction, we will be using Deep Neural Network (see Fig. 3.9. This neural network will be getting state, difference between current state and previous state and action and will be giving next state as an output. This Network is made out of one fully connected neural layer made of 512 neurons. We are using ReLU as the activation function.

Other learning parameters which will be used in the model are:

1. learning rate: 0.0001 with Adam [25] as gradient descent method with $\beta_1 = 0.99$ and $\beta_2 = 0.999$
2. mini-batch size: 64
3. epochs: 100

For reward prediction, we will be using SVM [7]. This algorithm works by finding a hyperplane that can separate points into two classes. This hyperplane can be described as set of points x which are subject to $\mathbf{w}^\mathsf{T}\mathbf{x} - b = 0$. Then we classify all points x_i to 1 if $\mathbf{w}^\mathsf{T}\mathbf{x_i} - b \geqslant 1$ or to -1 if $\mathbf{w}^\mathsf{T}\mathbf{x_i} - b \leqslant -1$. However, as points may not be perfectly separable we use hinge loss function: $\max\Big(0, 1 - y_i(\mathbf{w}^\mathsf{T}\mathbf{x}_i - b)\Big)$. Then we can rewrite our problem as optimization of following equation $\lambda\|\mathbf{w}\|^2 + \Big[\frac{1}{n}\sum_{i=1}^{n}\max\Big(0, 1 - y_i(\mathbf{w}^\mathsf{T}\mathbf{x}_i - b)\Big)\Big]$. Here $\lambda > 0$ determines the trade-off between margin size and ensuring that the $\mathbf{x}_i$ lies on the correct side of the margin.

We can solve this problem using SGD (Stochastic Gradient Descend) [4]. By performing one step of this descent we can use our SVM as an incremental learning algorithm.

This task will be a classification problem with classes being positive, negative, and neutral rewards. As inputs, it will be getting state, the difference between the current state and previous state, and action. Because this is a multiclass problem we will be using One Versus All approach. This means that we will create three SVMs and each will be solving a problem where instead of 3 classes there 2 classes: one chosen class and the rest of the classes as one single class.

Other learning parameters which will be used in model are:

1. learning rate: $\frac{1}{(\lambda*(t+t0))}$ where t is epoch number and $t0$ is chosen by a heuristic proposed by Leon Bottou [4]
2. λ: 0.0001
3. epochs: 1

When we can predict both states and rewards we can implement short-term planning. It works like this: from starting state generate all possible action sequences with a given depth. Depth is the length of these sequences. Then for each sequence predict the end state and total reward, using predictions of the previous states.

3.4.4. Final reasoning

Our final reasoning will work in following way:

- Assign cluster to current state using node clustering Birch tree.
- Use the Djikstra algorithm to find the shortest path to the state which has a positive reward edge adjacent to it.
- Generate scenarios using state and reward prediction models with chosen depth.
- Sort scenarios with the following priorities: first it maximizes reward, second it follows the furthest node on the proposed path, third if both scenarios end in the same node it chooses one which is closer to the final node centroid using the euclidean metric.
- For grounding, return information about the best scenario. This information is the proposed action sequence, predicted state, and reward while following this sequence.

Training routine for models covered in this section will be following:

- After end of episode gather entire trajectory and prepare it for training (split into current and next states).

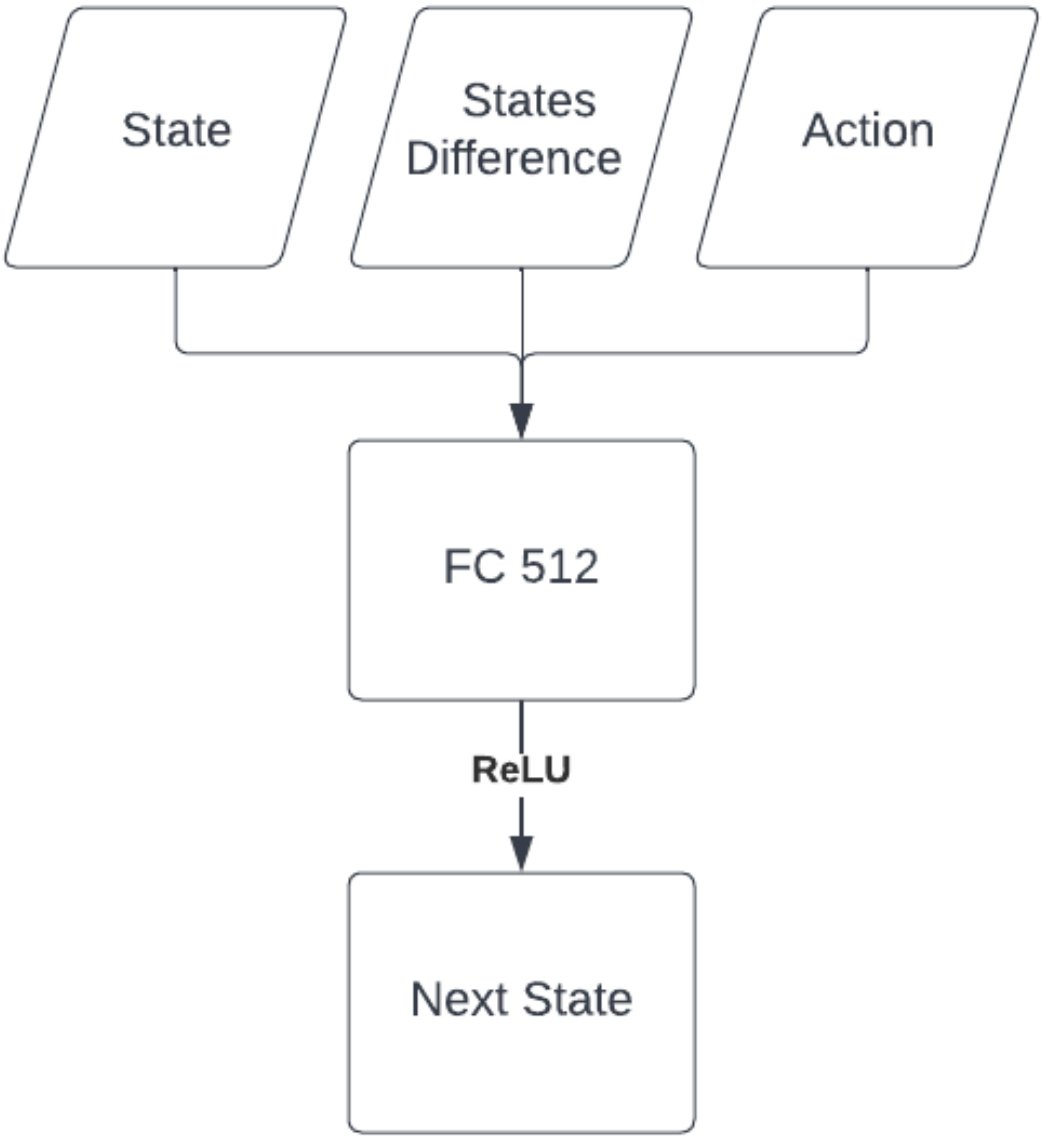

Figure 3.9: Model for prediction of the next states

- Train node clustering birch tree, edge clustering birch tree, state prediction neural net, reward prediction SVM.
- Reconstruct graph using new centroids from Birch trees.

4. Tests

4.1. Dimension Reduction and Clustering tests

We will test our embedding on an entirely new run. This run was random as training of Dimension Reduction and Clustering modules takes place before we train our Agent. The new example run had 1188 timesteps and number of all detected objects was 94902.

4.1.1. PCA test

To asses quality of PCA we will simply take percentage of explained variance. It turned out to be 99%.

We also checked how long it took it to reach stability.

As we can, see while we saw some bump in error it was going down overall and it achieved desired results.

4.1.2. Birch test

Birch trained on 577 batches before reaching stability. It seems that at some point it was getting unfamiliar states, that's why we see such a drop in balanced accuracy between old and new labels, and then it managed to get back to more familiar states where classification did not change even after training for 250 additional batches (see Fig. 4.2).

What about found objects? As we can see Birch found 14 distinct clusters, however because we set maximum number of clusters to 10 some of the classes were merged. As we can see on Fig. 4.3, objects found are pretty reasonable. We can see "paddle" cluster with number 0 got merged with cluster representing "digit one" or cluster number 3 which represents "wall". However some of the clusters centers are blurry. The reason for that is they represent rarer objects so PCA did not try to reconstruct them as much as the other objects. Fig. 4.4 show distribution of classes of found objects. As we can see some classes are much rarer than others. This is because some classes can only be detected when paddle is touching the wall.

To assess quality of Birch clustering we will use the following metrics:

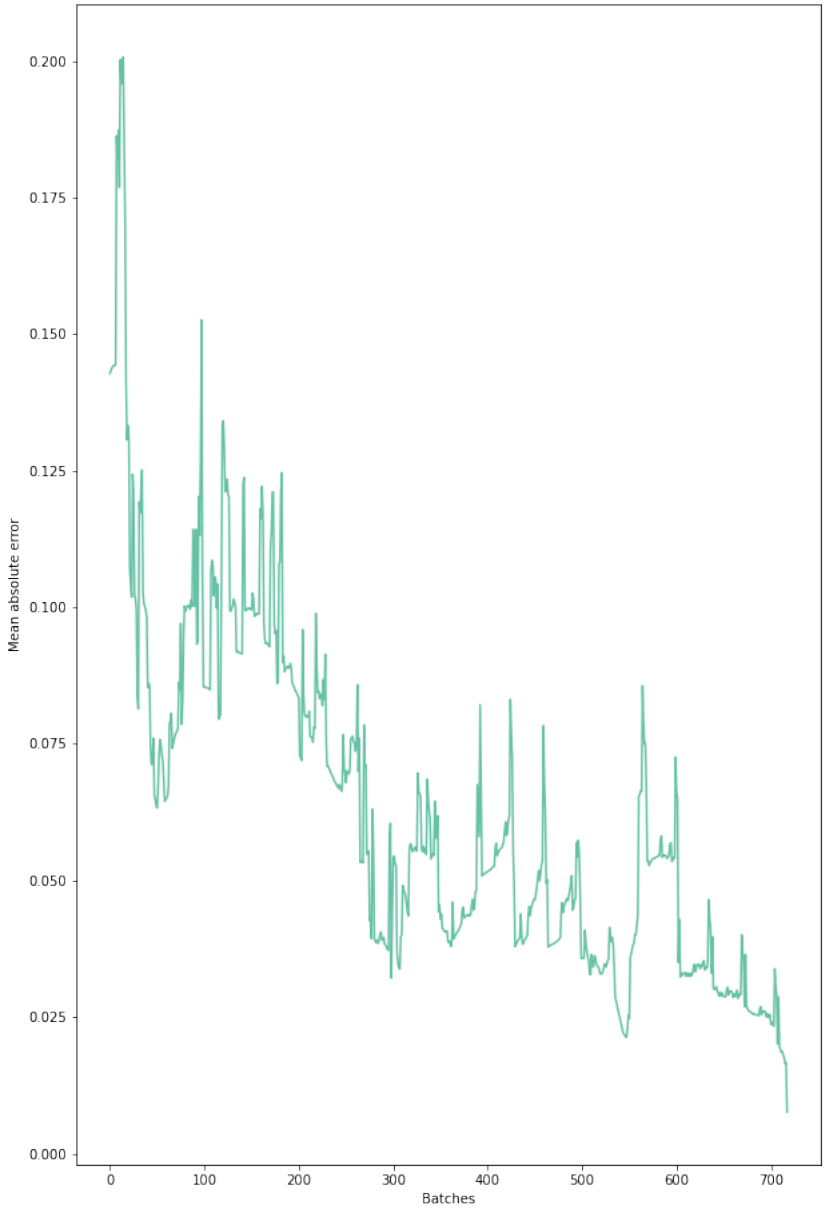

Figure 4.1: PCA mean absolute difference on task of transforming the same inputs after training for 250 batches

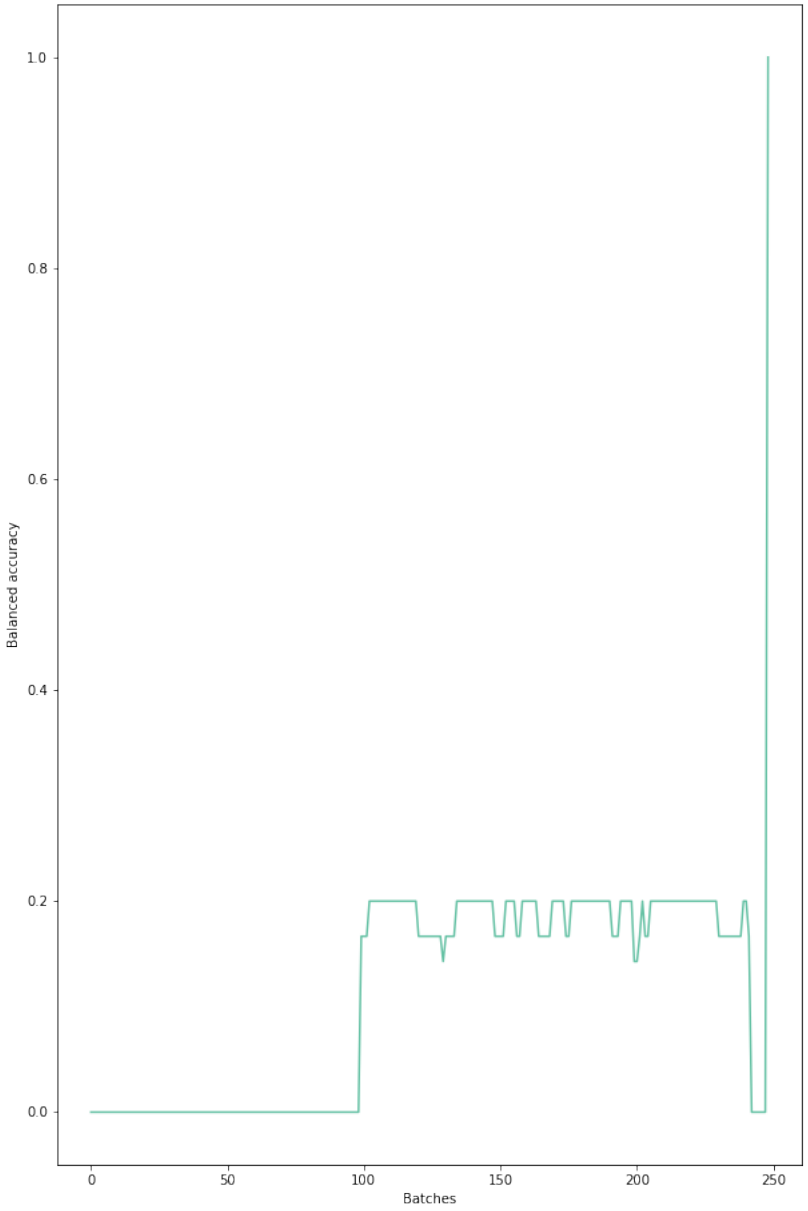

Figure 4.2: Birch balanced accuracy on task of predicting the same outputs after training for 250 batches

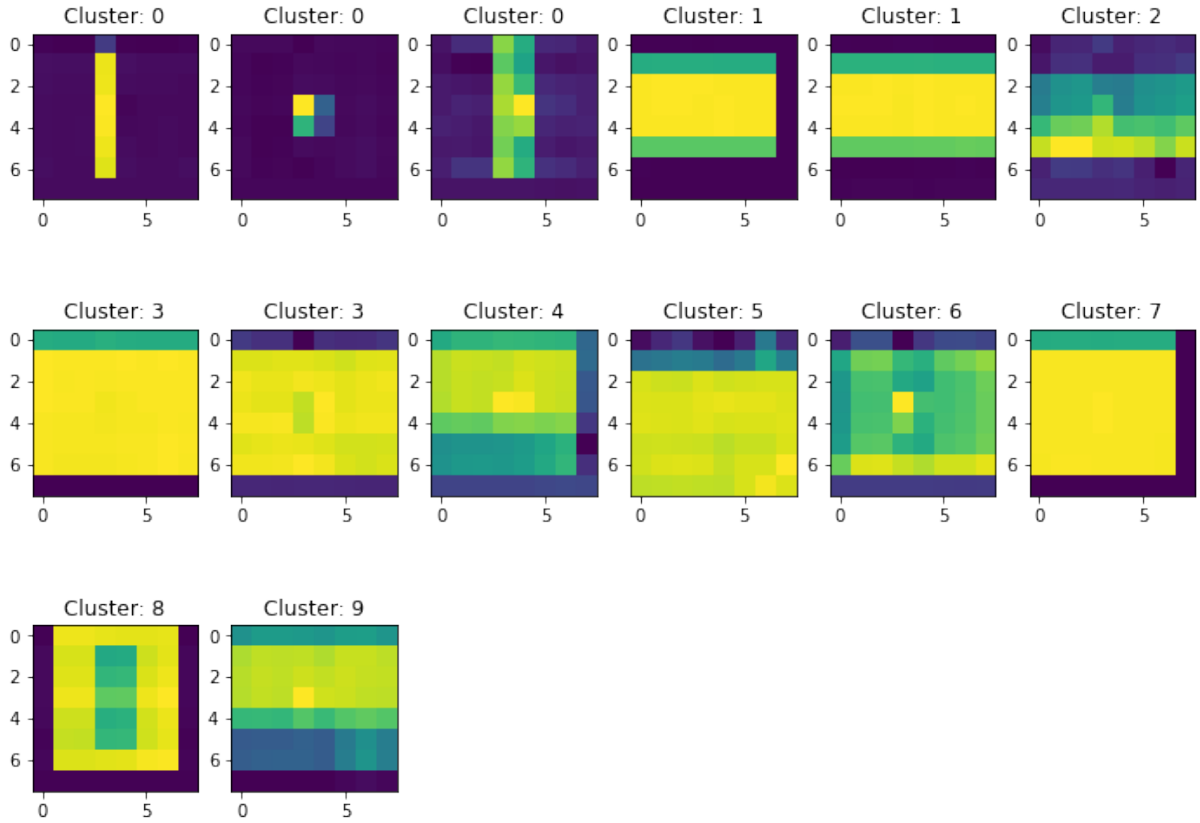

Figure 4.3: Clusters centers found by the Birch

1. Silhouette metric: 0.887

2. Davies Bouldin: 0.528

3. Dunn Index (using farthest distance for diameter, nearest distance for inter cluster distances) we calculate it for every tenth object because of memory constraints: 0.114

As we can see, clustering scores that we get are not bad. While Silhouette score is very good at 0.887, Davies Bouldin score is 0.528 which is the average result, and Dunn index is bad because we got only 0.114. The reason why clustering scores are very different among these metrics is because of object distribution inside the cluster. Points inside the cluster are very concentrated around one point in the cluster, often overlapping, and then there are a few semi-outliers that still belong to the cluster. As a result, the mean distance between points in the same cluster is amazing for Silhouette, but when we look at the diameter of the cluster the cluster quality is worse. To understand it better it is worth looking at Fig 4.5 and 4.6.

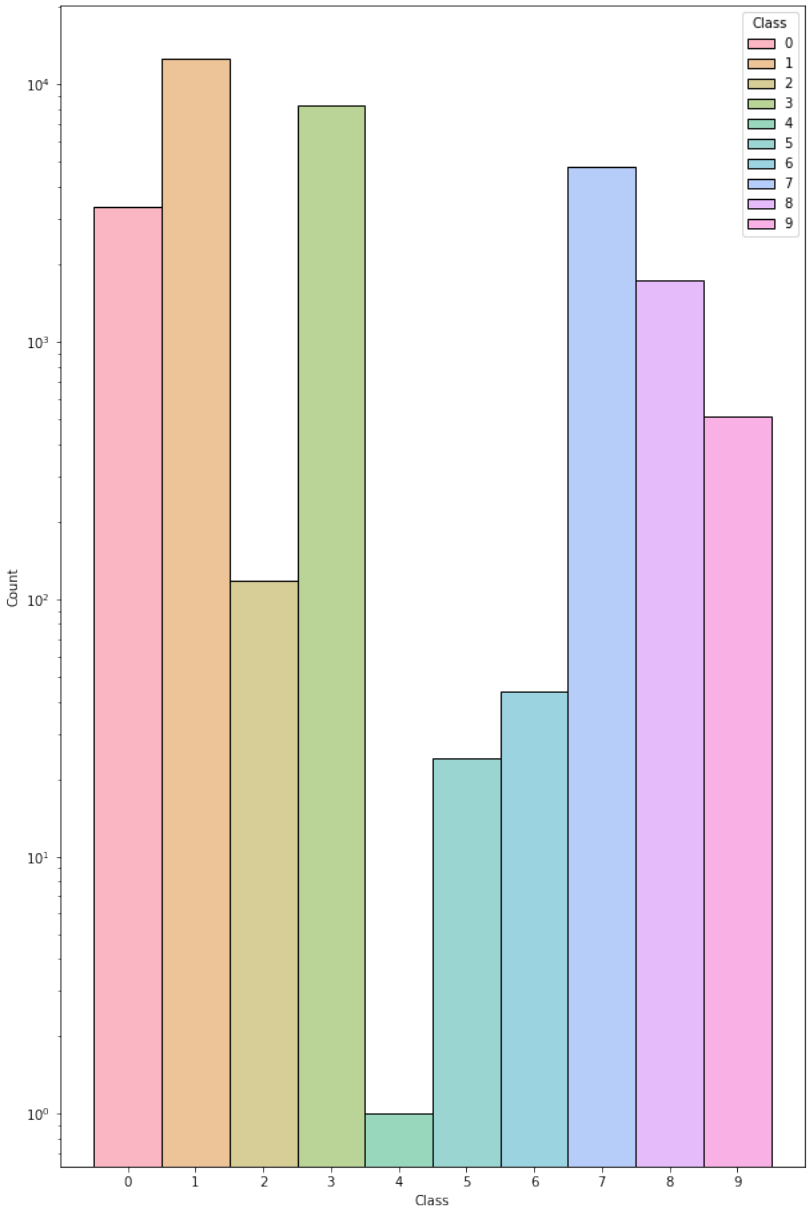

Figure 4.4: Object distributions found by the Birch, scale is logarithmic

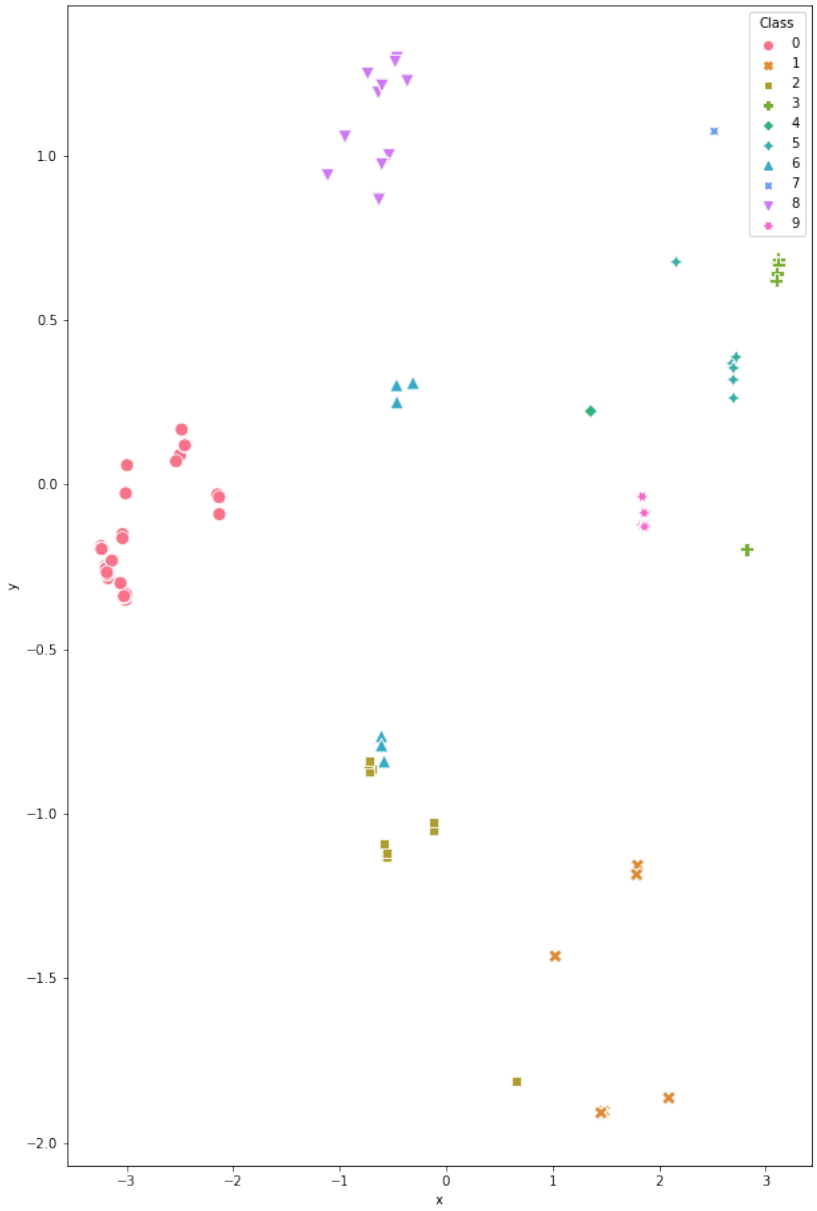

Figure 4.5: Object scatter plot found by the Birch. X and Y are the first and second Principal Components and together they explain 90% of the variance

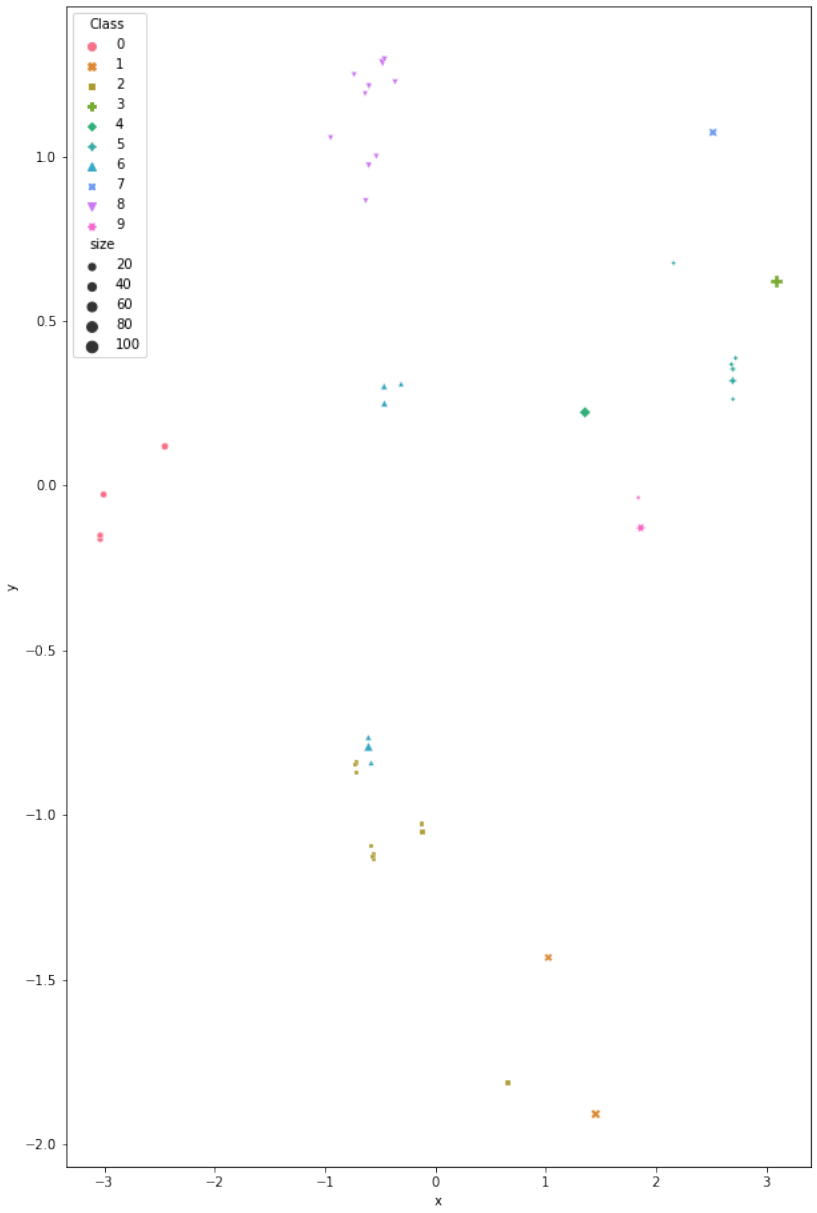

Figure 4.6: Object scatter plot found by the Birch. This plot was adjusted by the size of overlapping points. Overlapping points which covered less than 2% of overall cluster size got removed. X and Y are the first and second Principal Components and together they explain 90% of the variance

4.1.3. Embedding test

After training, our total vocabulary size is 9277 unique words. It is quite a lot for a language whose entire purpose is to describe the Pong game. On the other hand, some of the words are specific like '2apr_pos_y2' which means "paddle at approximately 15 pixel horizontally" or 'ec7a9c75e534f5a438c36cfb0c72ca7b' which means "wall surrounded by exactly two walls and nothing else".

To test embedding, we first need to establish criteria for comparing two different graphs which we are going to embed. We will use the following metric:

First, let's define the distance between two objects. If they are of the same kind then the distance between them will be the euclidean distance between their centers, if they are of two different kinds we add a class difference penalty to their distance.

To define the distance between two graphs, we take the sum of minimums of distances from objects from one graph to objects from the other graph. If the number of objects is different, then we take minimums from the smaller graph and add the object difference penalty.

We will use 100 for both kinds of penalties.

All of this can be summarised in the following formula: $\sum_{a \in A} min_{b \in B}(dist(a,b) + class(a,b) * classPenalty) + (\|A\| - \|B\|) * numberPenalty$

Where:

- A is the first Graph which is not greater than the second Graph
- B is the second Graph
- a is an object from A
- b is an object from B
- $class(a,b)$ is a function which gives 0 if a and b are of the same class
- $dist(a,b)$ is a function that is the euclidean distance between centers of the objects.
- $classPenalty$ is a penalty for the mismatching classes.
- $numberPenalty$ is a penalty for the different number of objects.

For comparing embedding we will use traditional cosine similarity [51]. This measure is used in NLP because it focuses on the words that the two documents do have in common, and the occurrence frequency of such words. This metric is also used by Gensim to find the most similar documents.

$$\text{cosine similarity} = S_C(A, B) := \cos(\theta) = \frac{\mathbf{A} \cdot \mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum_{i=1}^{n} A_i B_i}{\sqrt{\sum_{i=1}^{n} A_i^2}\sqrt{\sum_{i=1}^{n} B_i^2}},$$

Where A and B are our embeddings to compare.

One of the most basic ways to assess the quality of a semantic model like doc2vec is to see if documents embedding from the training set will be most similar to themselves after inferring their embedding from models (see Fig 4.8).

Lets start with Distributed memory (PV-DM). Distributed memory (PV-DM) achieved fatal results (see Fig. 4.7). The score we achieved is 0.138. However many states are very similar or outright identical, especially for starting states, so what is the accuracy if we allow the distance between to be less than 5? The answer is 0.218, which is still very bad. This algorithm failed for our case and later tests won't be performed for it.

What about the distributed bag of words (PV-DBOW)? In this case (see Fig. 4.8) score we achieved is 0.965, which is alright. However, as in the previous case, many states are very similar or outright identical, especially for starting state, so what is the accuracy if we allow the distance between to be less than 5? The answer is 0.999. Which is very good. However it is obvious that on the training set we will get a very good result, so this test is mostly just a sanity check of our model. How does it fare on the validation set?

We will do it in the following way. First, we will embed states from the validation run then we will find the most similar embedding from the training set and compare the two graphs. If as previously we allow state difference to be less then 5 then we get 0.665 (see Fig. 4.9), which is bad. If we relax our demands to less than 100 so still no missing/mislabeled objects are allowed we get 0.840 and finally if we want our demands to be less than 200 so 1 missing/mislabeled object is allowed we get 0.969. However, it is worth noting that often it is impossible to find a perfectly close graph from the training set. So we use the graph similarity metric directly for comparison. Then for this direct findings (see Fig. 4.10) we get 0.843 when distance is less than 5, 0.999 for distance less than 100 and 1.0 when distance is less than 200. When we take a ratio of these accuracies we get 0.789, 0.841, 0.969 for the allowed distance being 5, 100, and 200 respectively.

Let's also look at how the cosine similarity of embedding correlates to the difference in graphs (see Fig. 4.11). For the validation run, we will calculate differences between adjacent frames. The Pearson Correlation between those two metrics is -0.463. That means that there is a quite strong negative correlation between them. Correlation is negative because when two documents are semantically similar the cosine similarity goes to 1 and when they are similar graph distance should go down.

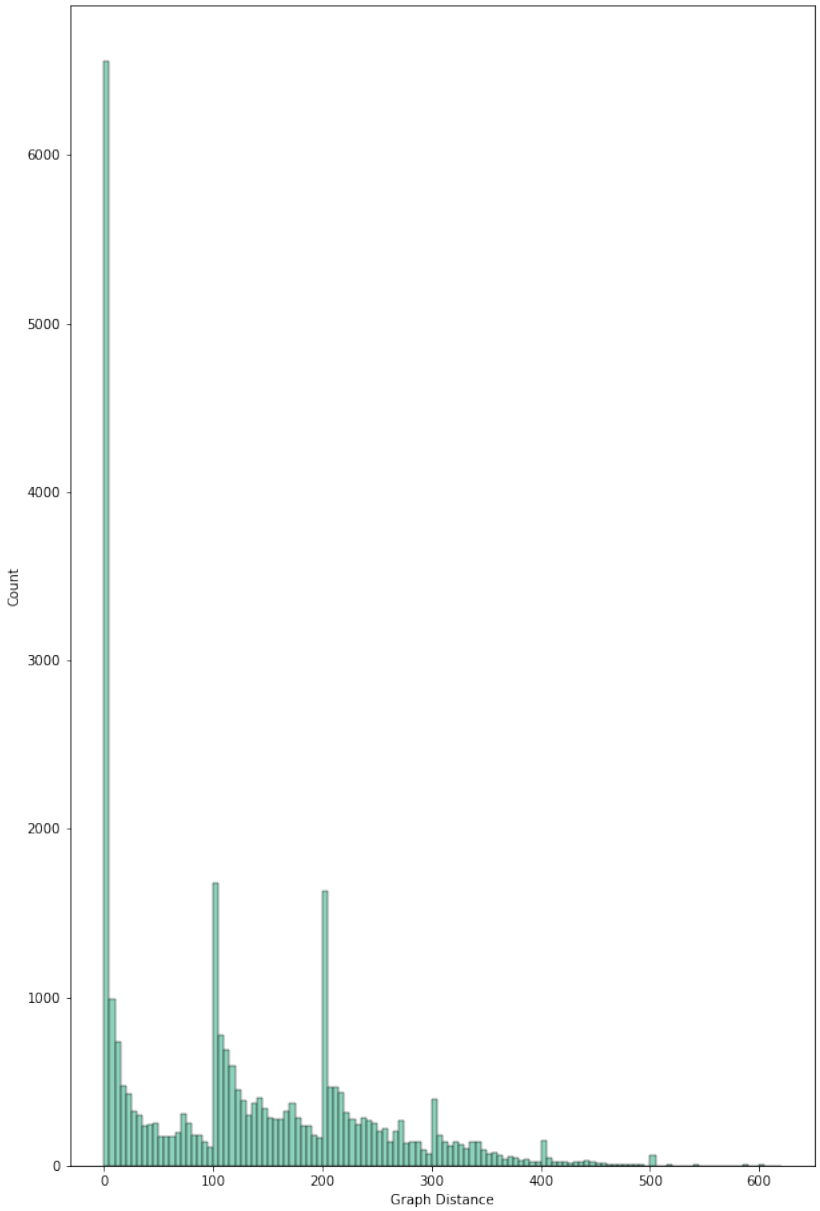

Figure 4.7: (PV-DM) Histogram of the number of graphs from the training set and their graph distance from graph from training set which was closest according to cosine similarity.

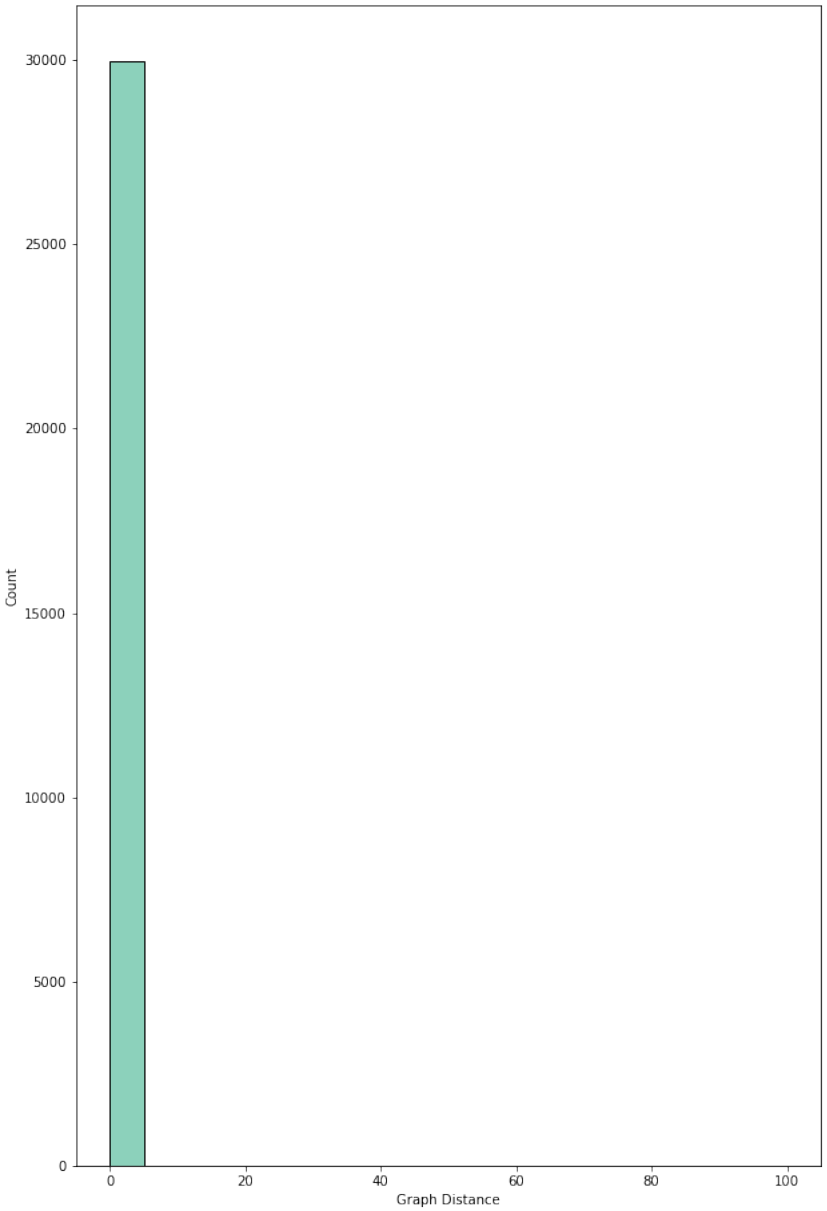

Figure 4.8: (PV-DBOW) Histogram of the number of graphs from the training set and their graph distance from the graph from the training set which was closest according to cosine similarity.

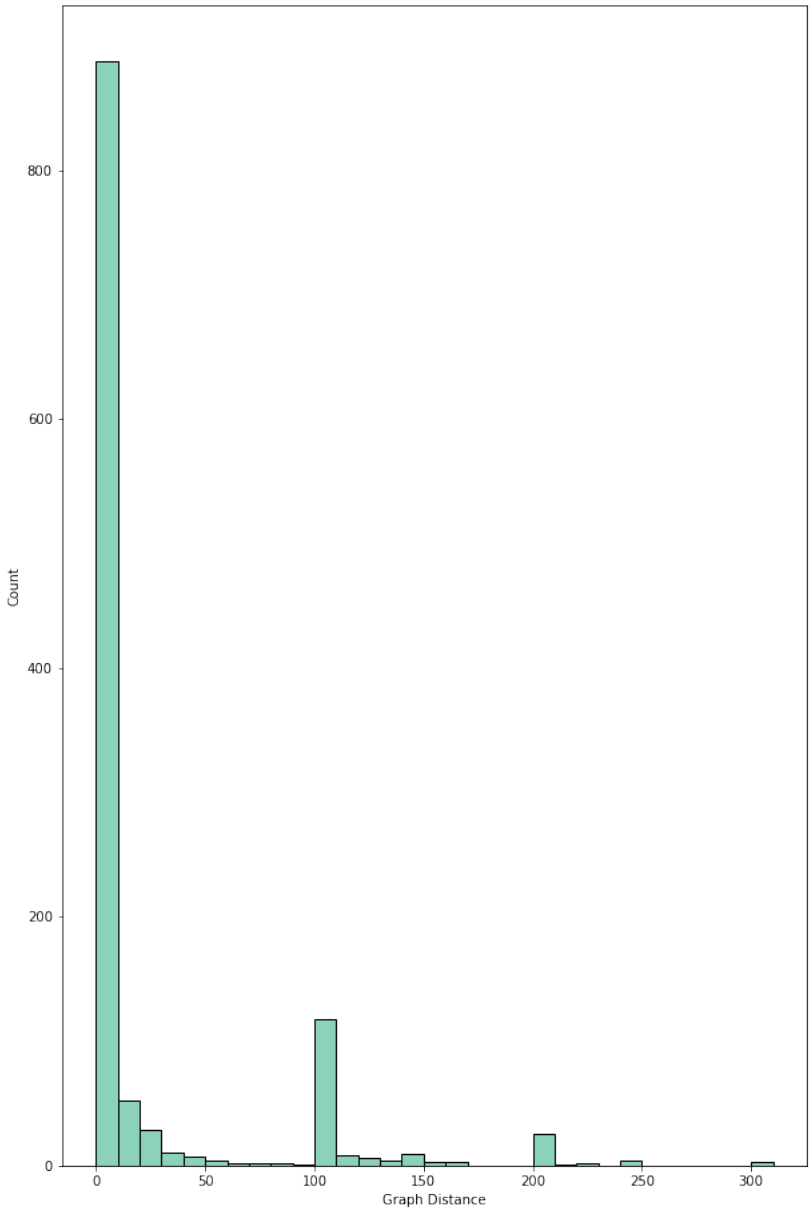

Figure 4.9: (PV-DBOW) Histogram of number of graphs from validation run and their graph distance from graph from training set which was closest according to cosine similarity

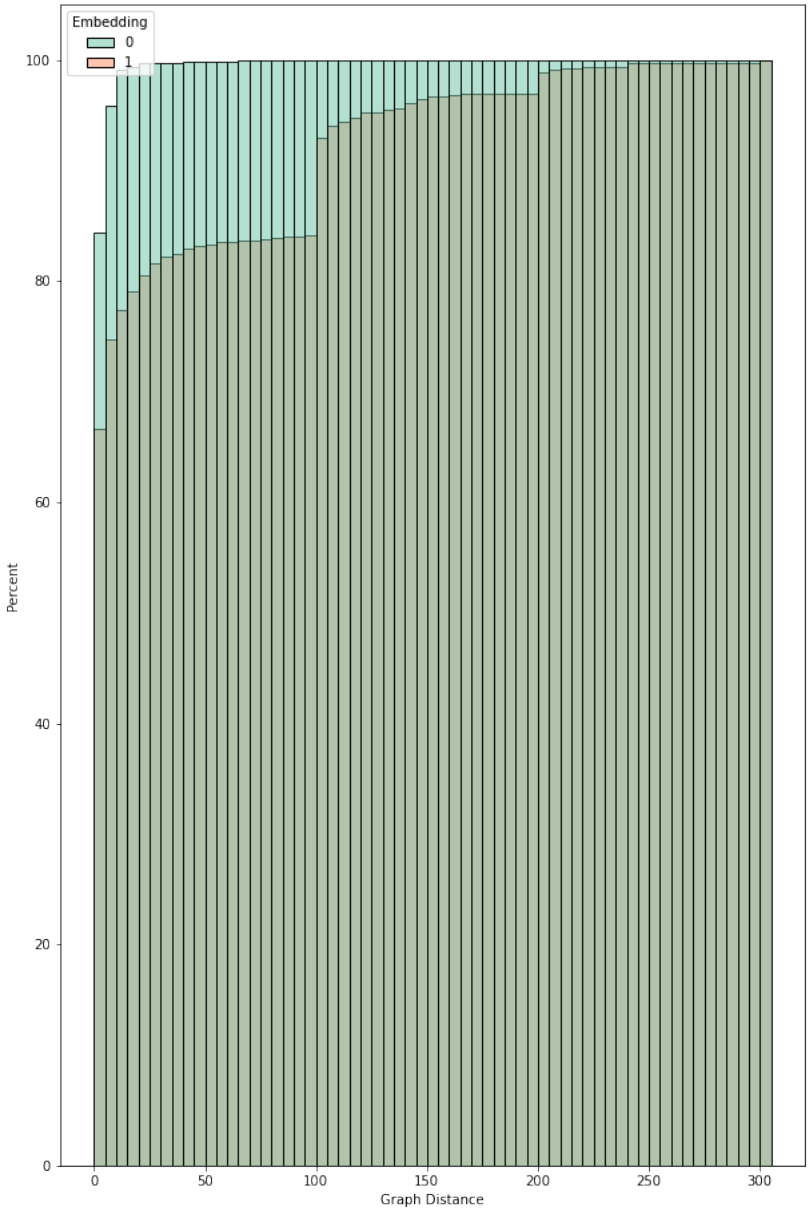

Figure 4.10: (PV-DBOW) Cumulative histogram of the percentage of graphs from validation run and their graph distance from graph from training set which was closest according to cosine similarity in case of embedding(Embedding=1) and directly using graph distance(Embedding=0)

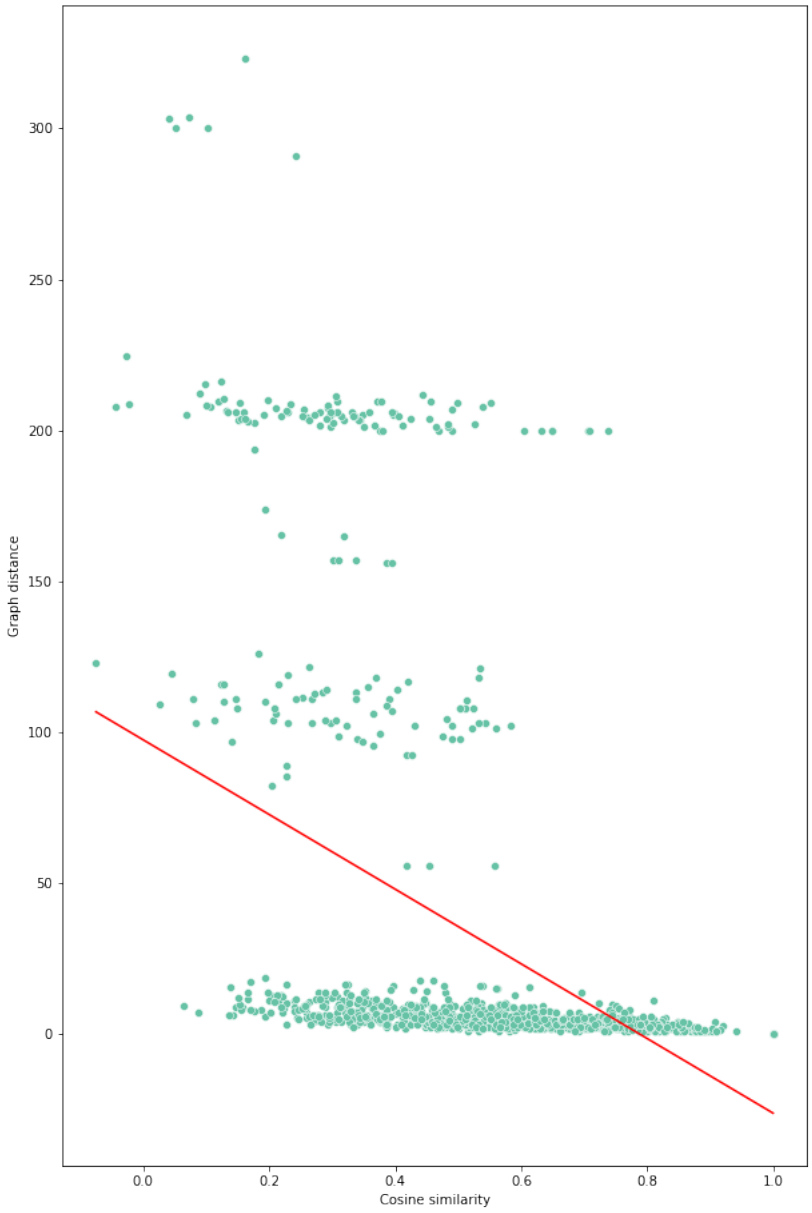

Figure 4.11: (PV-DBOW) Cosine similarity and graph difference. The Red line is fitted with linear regression.

In summary, while embedding is not perfect it is promising as it displays correct behaviors like the significant correlation between graph distance and cosine similarity.

4.2. Ontology and Planner test

In this section we will test models which were trained for purposes of Ontology and Planner modules. This testing takes place after completing training on 2 million timesteps. Also we test this modules both on semantic embedding and on RAM.

4.2.1. Testing on Semantic embedding

State prediction test

To test state prediction we made a new run. This run had 3875 timesteps. After using our state prediction neural network we got 31.5% R2 score. If we train and test the neural network from scratch on the new trajectory we get 75.3% R2 score. However, if we use 5-fold cross-validation on this new network our score from the best split is 2.5% R2. If we lower the number of neurons of this new neural network to just 128 then results are 17.9% R2 score for the best split of 5-fold cross-validation and 49.9% if we train and test on the whole trajectory. If we train our old network with new data we get 69.3% R2.

Reward prediction test

We use the same run as in the previous subsection. This run had 21 positive rewards, 14 negative rewards, and 3840 neutral rewards. Because this data is very imbalanced we are using balanced accuracy. After using our reward prediction model we get 34.6% balanced accuracy. If we train and test SVM from scratch on the new trajectory we get 83.2% balanced accuracy. If we use 5-fold cross-validation on this new network our balanced accuracy from the best split is 72.1%. If we update our old SVM with new data we get 36.2% R2. We should also analyze confusion matrices. When we look at confusion matrix of our original model (see Fig. 4.12). We can see that almost all transitions were classified as neutral. We can see that none of the positive rewards were classified as negative rewards and vice versa. This means that there is a good separation between these classes. After update (see Fig. 4.14) we can see that more positive rewards were correctly classified. When we look at the confusion matrix of the model trained on the whole trajectory from the beginning (see Fig. 4.13) we see that it was easier to classify positive rewards than negative rewards. We get 64% accuracy for negative rewards and 86% for

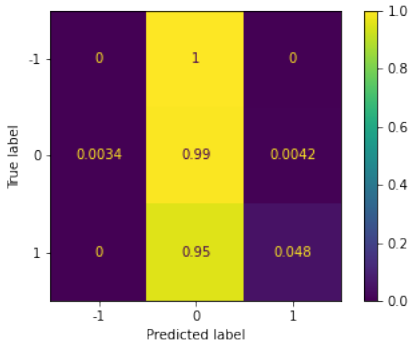

Figure 4.12: Confusion matrix of original reward prediction model. It is row wise normalised so all rows sum to 1.

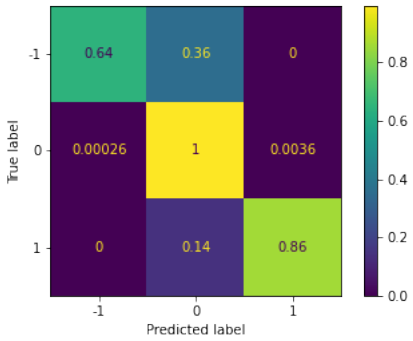

Figure 4.13: Confusion matrix of reward prediction model trained from beginning. It is row wise normalised so all rows sum to 1.

positive rewards.

Graph description

After finishing training our node Birch tree got 2184 centroids and our edge Birch tree got 6969 centroids. If we try to classify states and transitions of our 3875 timesteps long test run we get 960 unique node labels and 1213 unique edge labels. So we compressed states by a factor of 4 and transitions by a factor of around 3. If we try to fit Birch trees from the beginning we get 321 unique node labels and 744 unique edge labels. The reason for this difference is that as the number of samples grows it is easier for the Birch tree to form new subclusters. Another metric is the reward stored by the edge centroids. Here 1.6% of centroids had positive rewards and 2.5% had negative rewards. We can also look directly at our graph (see Fig. 4.15). Here red color represents negative rewards, green positive rewards, and yellow neutral rewards. positions of nodes were determined by the first two Principal Components which together explain only 10.8% of variance which may be the reason why red and green nodes are not separable in the

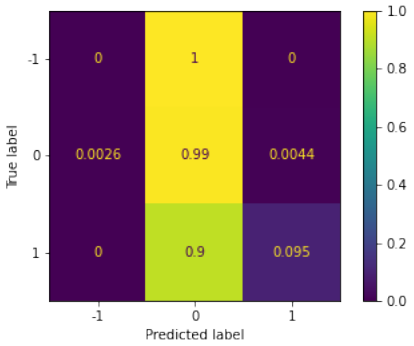

Figure 4.14: Confusion matrix of original reward prediction model after update. It is row wise normalised so all rows sum to 1.

picture. The diameter of this graph is 20 and the radius is 9.

4.2.2. Testing Planner on RAM

RAM State prediction test

To test state prediction on RAM we made a new run. This run had 9022 timesteps. After using our state prediction neural network we got 97.3% R2 score. If we train and test the neural network from scratch on the new trajectory we get 97.4% R2 score. However, if we use 5-fold cross-validation on this new network our score from the best split is 98.1% R2. If we lower the number of neurons of this new neural network to just 128 then results are 97.8% R2 score for the best split of 5-fold cross-validation and 97.2% if we train and test on the whole trajectory. If we train our old network with new data we get 98.6% R2.

RAM Reward prediction test

We use the same run as in the previous subsection. This run had 12 positive rewards, 21 negative rewards, and 8989 neutral rewards. Because this data is very imbalanced we are using balanced accuracy. After using our reward prediction model we get 33.3% balanced accuracy. If we train and test SVM from scratch on the new trajectory we get 77.3% balanced accuracy. If we use 5-fold cross-validation on this new network our balanced accuracy from the best split is 77.3%. If we update our old SVM with new data we still get only 33.3% R2. When we look at confusion matrix of our original model (see Fig. 4.17). We can see that all transitions were classified as neutral. When we look at the confusion matrix of the model trained on the whole trajectory from the beginning (see Fig. 4.16) we see that it was easier to classify positive rewards than negative rewards. We get 57% accuracy for negative rewards and 75% for positive rewards.

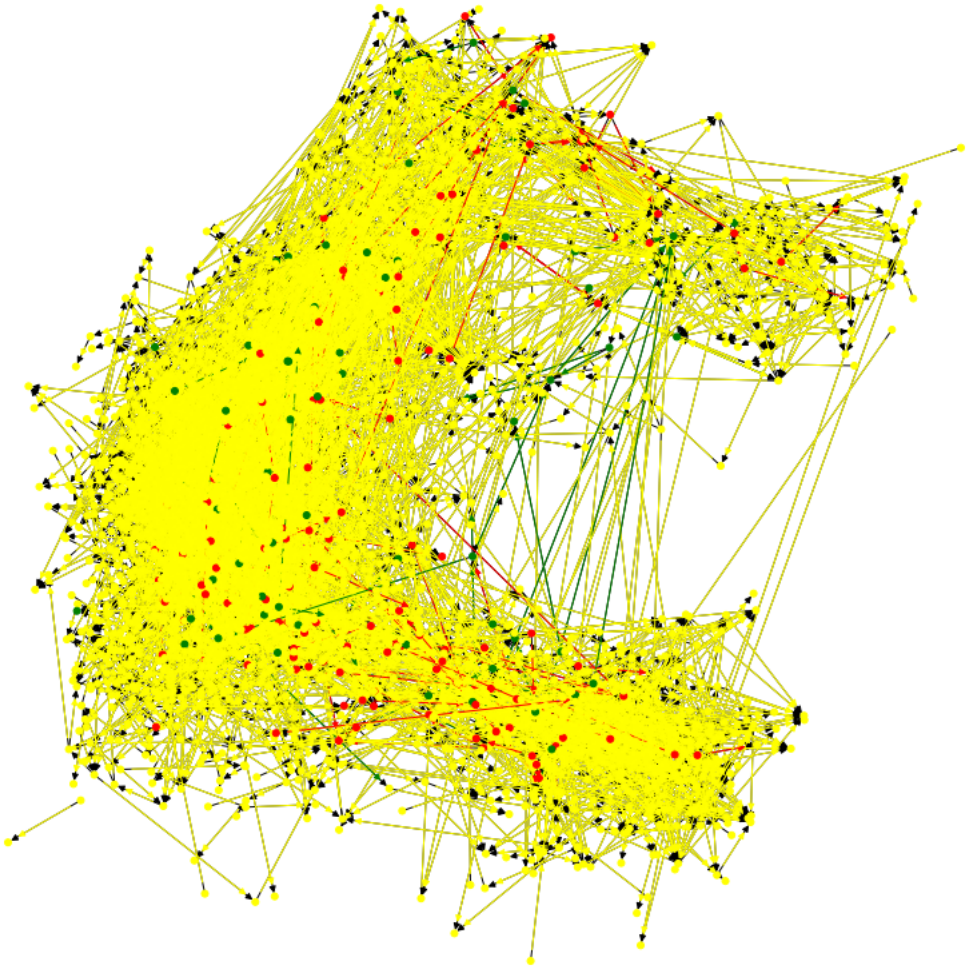

Figure 4.15: Graph generated using Birch trees. Here red color represents negative rewards, green positive rewards and yellow neutral rewards. positions of nodes were determined by first two Principal Components which together explain 10.8% of variance

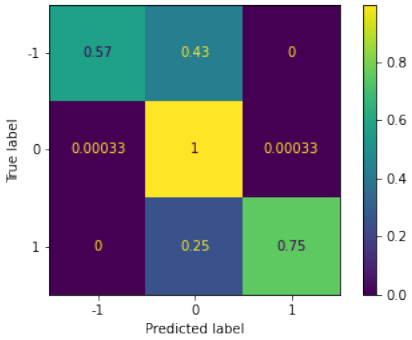

Figure 4.16: Confusion matrix of RAM reward prediction model trained from beginning. It is row wise normalised so all rows sum to 1.

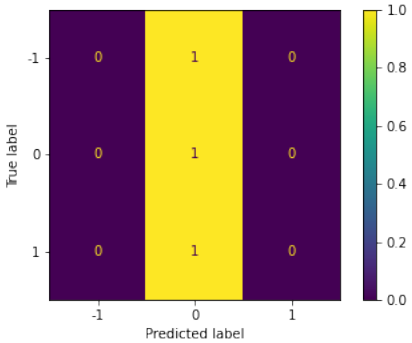

Figure 4.17: Confusion matrix of original RAM reward prediction model after and before update. It is row wise normalised so all rows sum to 1.

RAM State map description

After finishing training our node Birch tree got 185 centroids and our edge Birch tree got 1966 centroids. If we try to classify states and transitions of our 9022 timesteps long test run we get 146 unique node labels and 802 unique edge labels. If we try to fit Birch trees from the beginning we get 79 unique node labels and 373 unique edge labels. The reason for this difference is that as the number of samples grows it is easier for the Birch tree to form new subclusters. Another metric is the reward stored by the edge centroids. Here 1.8% of centroids had positive rewards and 3.1% had negative rewards. We can also look directly at our graph (see Fig. 4.18). Here red color represents negative rewards, green positive rewards, and yellow neutral rewards. positions of nodes were determined by the first two Principal Components which together explain 66.4% of variance. We can easily see separation between red and green nodes. The diameter of this graph is 17 and the radius is 8.

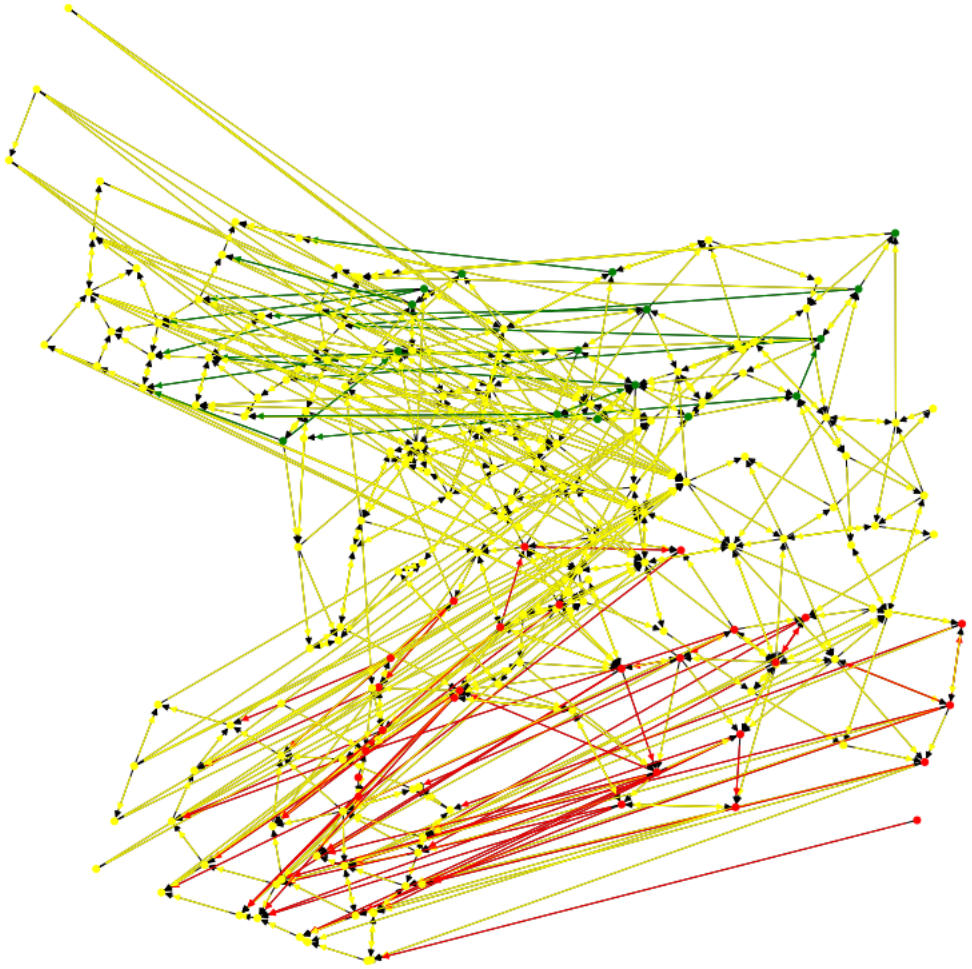

Figure 4.18: Graph generated using RAM Birch trees. Here red color represents negative rewards, green positive rewards and yellow neutral rewards. positions of nodes were determined by first two Principal Components which together explain 66.4% of variance

4.3. Final Results on pong

There will be five main experiments: Learning on RAM only, Learning without preprocessing (except basic one described in earlier section), learning after object detection by Otsu threshold, learning after object classification by Birch and finally after embedding by graph2vec. In case of learning after object detection by Otsu threshold and object classification by Birch we give labeled image as input. We will also try to use only actions proposed by the Planner.

As we can see on Fig. 4.19 the fastest converging was one based on Otsu detection, while models based on semantic embedding and RAM were lagging behind. These models also displayed very high variance. The reason why numbers of episodes is different among different models is that we were training on set amount of frames and even matches takes longer than ones where one side wins overwhelmingly. To get better grasp at average performance it is better to look at Fig. 4.20. There we can see that model based on semantic embedding outperforms RAM based one. However when we look at table 4.1 which is summarising last 10 runs semantic model displays very high variance. Another key aspect is that models which got access to generated goals by our reasoners got better results than ones without it. However when we run these models using only proposed actions by planner we could not score any points. It seems that although planner could not play Pong by itself it still helped agents to score points.

Model	Mean	Max	Min	Std
baseline	20.8	21	20	0.42
otsu	20.1	21	18	1.19
birch	20.4	21	18	0.96
semantic	13.6	20	-10	9.60
semantic map	14.1	21	3	6.26
RAM	-6.4	4	-12	4.55
RAM map	-0.6	8	-6	4.76

Table 4.1: Models results on last 10 runs.

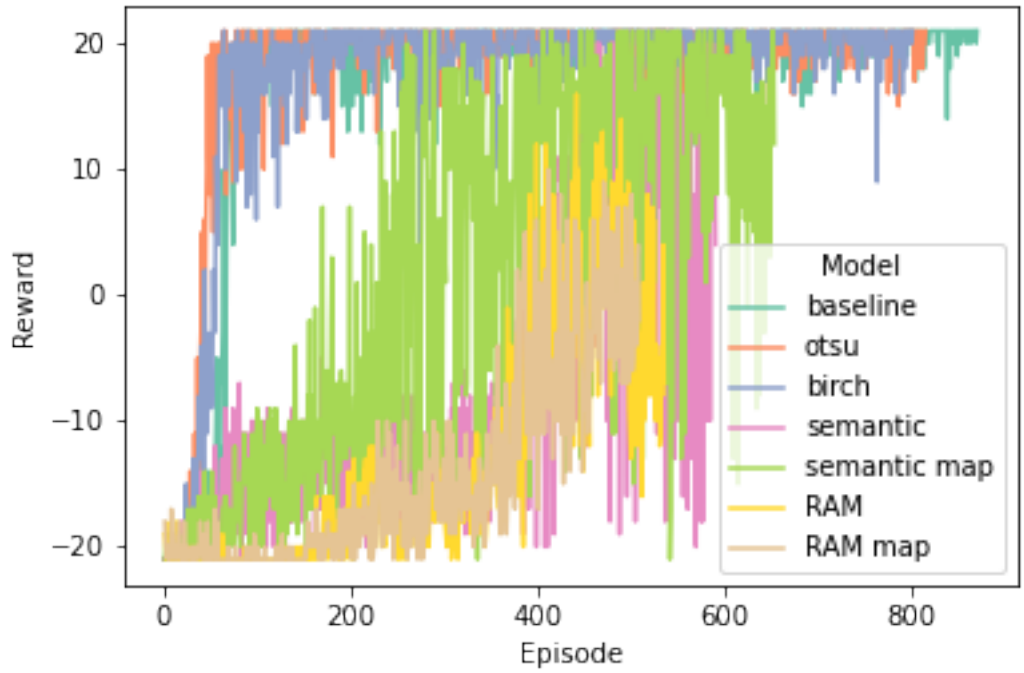

Figure 4.19: Episode reward for given run.

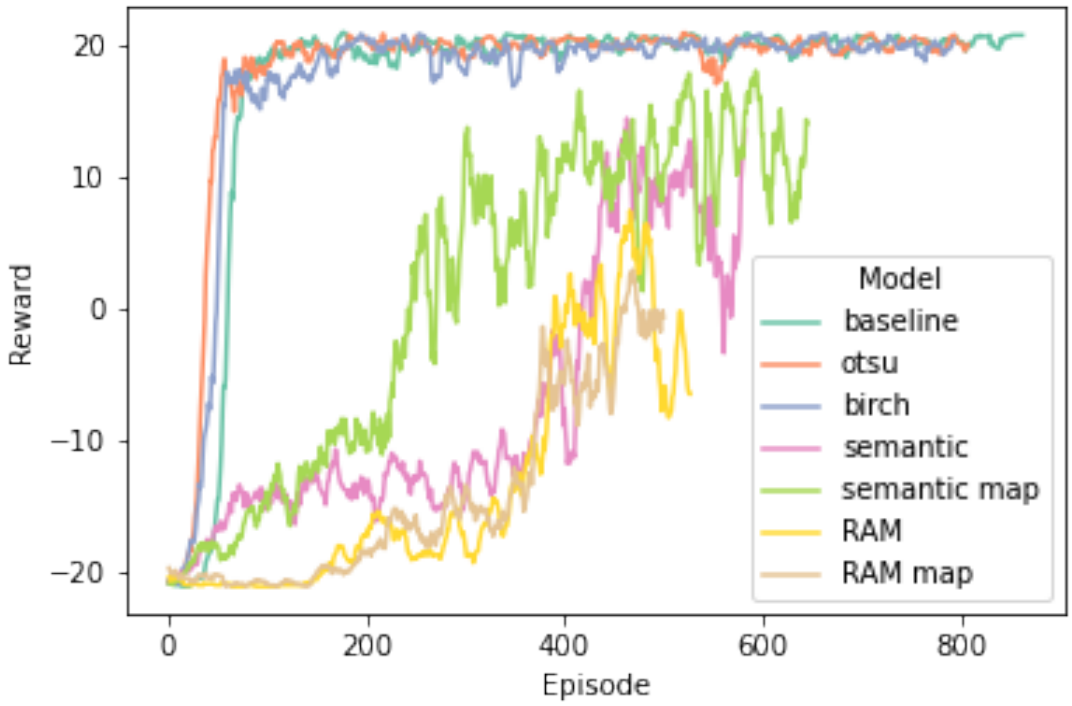

Figure 4.20: Mean episode reward averaged for last 10 runs.

5. Summary

In the end we managed to train model based on semantic embedding which can play Pong however it has high variance and converge more slowly then ones which are using convolutions. We also found out that by giving our model labeled images it converges faster but simpler object detection was better than after subsequent object classification. We also used reasoning to generate goals for our agents. Unfortunately we did not use descriptive logic as during implementation it turned out that using graph was much easier. Another insight was that although prediction of both states and rewards was far from perfect and planner could not play by itself it still helped agents. The greatest obstacle to this thesis which was not directly overcome but rather avoided was the fact that Gensim [45] implementation of doc2vec [29] DO NOT support incremental learning (see github issues [50], [9]). Solving this issue could be topic of master or engineering thesis on its own.

Bibliography

[1] Abien Fred Agarap. "Deep learning using rectified linear units (relu)". In: *arXiv preprint arXiv:1803.08375* (2018).

[2] Ahmed Akakzia et al. *Grounding Language to Autonomously-Acquired Skills via Goal Generation*. 2021. arXiv: 2006.07185 [cs.AI].

[3] Yolanda Blanco-Fernández et al. "Semantic Reasoning: A Path to New Possibilities of Personalization". In: *The Semantic Web: Research and Applications*. Ed. by Sean Bechhofer et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 720–735. ISBN: 978-3-540-68234-9.

[4] Léon Bottou. "Stochastic Gradient Descent Tricks". In: *Neural Networks: Tricks of the Trade*. 2012.

[5] Hongyun Cai, Vincent W. Zheng, and Kevin Chen-Chuan Chang. *A Comprehensive Survey of Graph Embedding: Problems, Techniques and Applications*. 2017. DOI: 10.48550/ARXIV.1709.07604. URL: https://arxiv.org/abs/1709.07604.

[6] G.B. Coleman and H.C. Andrews. "Image segmentation by clustering". In: *Proceedings of the IEEE* 67.5 (1979), pp. 773–785. DOI: 10.1109/PROC.1979.11327.

[7] Corinna Cortes and Vladimir Vapnik. "Support-vector networks". In: *Machine learning* 20.3 (1995), pp. 273–297.

[8] Edsger W Dijkstra. "A note on two problems in connexion with graphs". In: *Numerische mathematik* 1.1 (1959), pp. 269–271.

[9] *Doc2Vec: when we have string tags, build_vocab with update removes previous index*. https://github.com/RaRe-Technologies/gensim/issues/3162. (Visited on 10/06/2022).

[10] J. C. Dunn†. "Well-Separated Clusters and Optimal Fuzzy Partitions". In: *Journal of Cybernetics* 4.1 (1974), pp. 95–104. DOI: 10.1080/01969727408546059. eprint: https://doi.org/10.1080/01969727408546059. URL: https://doi.org/10.1080/01969727408546059.

[11] Manfred Eppe, Phuong D. H. Nguyen, and Stefan Wermter. "From Semantics to Execution: Integrating Action Planning With Reinforcement Learning for Robotic Causal Problem-Solving". In: *Frontiers in Robotics and AI* 6 (2019), p. 123. ISSN: 2296-9144. DOI: 10.3389/frobt.2019.00123. URL: https://www.frontiersin.org/article/10.3389/frobt.2019.00123.

[12] Malik Ghallab et al. "PDDL - The Planning Domain Definition Language". In: (Aug. 1998).

[13] Birte Glimm et al. "Hermit: An Owl 2 Reasoner". In: *Journal of Automated Reasoning* 53 (Oct. 2014). DOI: 10.1007/s10817-014-9305-1.

[14] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

[15] Thomas R. Gruber and Gregory R. Olsen. "An Ontology for Engineering Mathematics". In: *KR*. 1994.

[16] Nicola Guarino, Daniel Oberle, and Steffen Staab. "What Is an Ontology?" In: May 2009, pp. 1–17. DOI: 10.1007/978-3-540-92673-3_0.

[17] David Ha and Jürgen Schmidhuber. "World Models". In: (Mar. 2018). DOI: 10.5281/zenodo.1207631.

[18] Hirotaka Hachiya and Masashi Sugiyama. "Feature Selection for Reinforcement Learning: Evaluating Implicit State-Reward Dependency via Conditional Mutual Information". In: *ECML/PKDD*. 2010.

[19] Austin W. Hanjie, Victor Zhong, and Karthik Narasimhan. *Grounding Language to Entities and Dynamics for Generalization in Reinforcement Learning*. 2021. arXiv: 2101.07393 [cs.CL].

[20] Bernhard Hengst. "Hierarchical Reinforcement Learning". In: *Encyclopedia of Machine Learning*. Ed. by Claude Sammut and Geoffrey I. Webb. Boston, MA: Springer US, 2010, pp. 495–502. ISBN: 978-0-387-30164-8. DOI: 10.1007/978-0-387-30164-8_363. URL: https://doi.org/10.1007/978-0-387-30164-8_363.

[21] Ningyuan Teresa Huang and Soledad Villar. "A Short Tutorial on The Weisfeiler-Lehman Test And Its Variants". In: *ICASSP 2021 - 2021 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, June 2021. DOI: 10.1109/icassp39728.2021.9413523. URL: https://doi.org/10.1109%5C%2Ficassp39728.2021.9413523.

[22] Nidal Ismael, Mahmoud Alzaalan, and Wesam Ashour. "Improved Multi Threshold Birch Clustering Algorithm". In: *International Journal of Artificial Intelligence and Applications for Smart Devices* 2 (May 2014), pp. 1–10. DOI: 10.14257/ijaiasd.2014.2.1.01.

[23] Xin Jin and Jiawei Han. "K-Means Clustering". In: *Encyclopedia of Machine Learning*. Ed. by Claude Sammut and Geoffrey I. Webb. Boston, MA: Springer US, 2010, pp. 563–564. ISBN: 978-0-387-30164-8. DOI: 10.1007/978-0-387-30164-8_425. URL: https://doi.org/10.1007/978-0-387-30164-8_425.

[24] Shivaram Kalyanakrishnan et al. *An Analysis of Frame-skipping in Reinforcement Learning*. 2021. DOI: 10.48550/ARXIV.2102.03718. URL: https://arxiv.org/abs/2102.03718.

[25] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. 2014. DOI: 10.48550/ARXIV.1412.6980. URL: https://arxiv.org/abs/1412.6980.

[26] Nils Kriege, Fredrik Johansson, and Christopher Morris. "A survey on graph kernels". In: *Applied Network Science* 5 (Jan. 2020). DOI: 10.1007/s41109-019-0195-3.

[27] Markus Krötzsch, Frantisek Simancik, and Ian Horrocks. *A Description Logic Primer*. 2013. arXiv: 1201.4089 [cs.AI].

[28] Maxat Kulmanov et al. "Semantic similarity and machine learning with ontologies". In: *Briefings in Bioinformatics* 22 (Oct. 2020). DOI: 10.1093/bib/bbaa199.

[29] Quoc V. Le and Tomas Mikolov. *Distributed Representations of Sentences and Documents*. 2014. DOI: 10.48550/ARXIV.1405.4053. URL: https://arxiv.org/abs/1405.4053.

[30] Douglas B. Lenat and R. V. Guha. *Building Large Knowledge-Based Systems; Representation and Inference in the Cyc Project*. 1st. USA: Addison-Wesley Longman Publishing Co., Inc., 1989. ISBN: 0201517523.

[31] Bin Liu et al. "Combining Ontology and Reinforcement Learning for Zero-Shot Classification". In: *Knowledge-Based Systems* 144 (Dec. 2017). DOI: 10.1016/j.knosys.2017.12.022.

[32] Jelena Luketina et al. "A Survey of Reinforcement Learning Informed by Natural Language". In: *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*. International Joint Conferences on Artificial Intelligence Organization, July 2019, pp. 6309–6317. DOI: 10.24963/ijcai.2019/880. URL: https://doi.org/10.24963/ijcai.2019/880.

[33] Laurens van der Maaten, Eric Postma, and H. Herik. "Dimensionality Reduction: A Comparative Review". In: *Journal of Machine Learning Research - JMLR* 10 (Jan. 2007).

[34] Sidharth Mishra et al. “Principal Component Analysis”. In: *International Journal of Livestock Research* (Jan. 2017), p. 1. DOI: 10.5455/ijlr.20170415115235.

[35] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013. DOI: 10.48550/ARXIV.1312.5602. URL: https://arxiv.org/abs/1312.5602.

[36] Annamalai Narayanan et al. *graph2vec: Learning Distributed Representations of Graphs*. 2017. DOI: 10.48550/ARXIV.1707.05005. URL: https://arxiv.org/abs/1707.05005.

[37] Mahamed Omran, Andries Engelbrecht, and Ayed Salman. “An overview of clustering methods”. In: *Intell. Data Anal.* 11 (Nov. 2007), pp. 583–605. DOI: 10.3233/IDA-2007-11602.

[38] *OpenAI Gym — Atari games, Classic Control, Robotics and more*. https://gym.openai.com/. (Visited on 10/16/2021).

[39] Nobuyuki Otsu. “A Threshold Selection Method from Gray-Level Histograms”. In: *IEEE Transactions on Systems, Man, and Cybernetics* 9.1 (1979), pp. 62–66. DOI: 10.1109/TSMC.1979.4310076.

[40] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. URL: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf.

[41] *Principal Components Analysis Explained for Dummies*. https://programmathically.com/principal-components-analysis-explained-for-dummies/. (Visited on 12/10/2021).

[42] *Protege*. https://protege.stanford.edu/. (Visited on 12/10/2021).

[43] *Python, an interpreted high-level general-purpose programming language*. https://www.python.org/. (Visited on 12/10/2021).

[44] *RDF/XML*. https://www.w3.org/TR/rdf-syntax-grammar/. (Visited on 12/10/2021).

[45] Radim Rehurek and Petr Sojka. “Gensim–python framework for vector space modelling”. In: *NLP Centre, Faculty of Informatics, Masaryk University, Brno, Czech Republic* 3.2 (2011).

[46] *Resource Description Framework (RDF) Model and Syntax Specification*. https://www.w3.org/TR/1999/REC-rdf-syntax-19990222/. (Visited on 12/10/2021).

[47] Benedek Rozemberczki, Oliver Kiss, and Rik Sarkar. "Karate Club: An API Oriented Open-source Python Framework for Unsupervised Learning on Graphs". In: (2020). DOI: 10.48550/ARXIV.2003.04819. URL: https://arxiv.org/abs/2003.04819.

[48] Shaker El-Sappagh et al. "SNOMED CT standard ontology based on the ontology for general medical science". In: *BMC Medical Informatics and Decision Making* 18 (Aug. 2018). DOI: 10.1186/s12911-018-0651-5.

[49] Erez Schwartz et al. *Language is Power: Representing States Using Natural Language in Reinforcement Learning*. 2020. arXiv: 1910.02789 [cs.CL].

[50] *Segmentation fault using build_vocab(..., update=True) for Doc2Vec*. https://github.com/RaRe-Technologies/gensim/issues/1019. (Visited on 10/06/2022).

[51] Pinky Sitikhu et al. *A Comparison of Semantic Similarity Methods for Maximum Human Interpretability*. 2019. DOI: 10.48550/ARXIV.1910.09129. URL: https://arxiv.org/abs/1910.09129.

[52] John Sowa. "Knowledge Representation: Logical, Philosophical, and Computational Foundations". In: Jan. 2000.

[53] *SPARQL Query Language for RDF*. https://www.w3.org/TR/rdf-sparql-query/. (Visited on 12/10/2021).

[54] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: an introduction*. Second edition. Adaptive computation and machine learning series. The MIT Press, 2018. ISBN: 9780262039246.

[55] Stefanie Tellex et al. "Robots That Use Language". In: 2020.

[56] *Terse RDF Triple Language*. https://www.w3.org/TR/turtle/. (Visited on 12/10/2021).

[57] *The Organization Ontology W3C Recommendation 16 January 2014*. https://www.w3.org/TR/vocab-org/. (Visited on 12/10/2021).

[58] Emre Ugur and Justus Piater. "Bottom-up learning of object categories, action effects and logical rules: From continuous manipulative exploration to symbolic planning". In: *2015 IEEE International Conference on Robotics and Automation (ICRA)*. 2015, pp. 2627–2633. DOI: 10.1109/ICRA.2015.7139553.

[59] Manuela M. Veloso. "Planning and Learning by Analogical Reasoning". In: *Lecture Notes in Computer Science*. 1994.

[60] *ViZDoom Doom-based AI Research Platform for Reinforcement Learning from Raw Visual Information*. http://vizdoom.cs.put.edu.pl/. (Visited on 10/16/2021).

[61] David Walden. “THE BELLMAN-FORD ALGORITHM AND ”DISTRIBUTED BELLMAN-FORD”. In: (Jan. 2008).

[62] Shenghui Wang and Rob Koopman. “Clustering articles based on semantic similarity”. In: *Scientometrics* 111.2 (Feb. 2017), pp. 1017–1031. ISSN: 1588-2861. DOI: 10.1007/s11192-017-2298-x. URL: http://dx.doi.org/10.1007/s11192-017-2298-x.

[63] *Web Ontology Language (OWL)*. https://www.w3.org/OWL/. (Visited on 12/10/2021).

[64] Deli Yu et al. *Towards Accurate Scene Text Recognition with Semantic Reasoning Networks*. 2020. arXiv: 2003.12294 [cs.CV].

[65] Nida M. Zaitoun and Musbah J. Aqel. “Survey on Image Segmentation Techniques”. In: *Procedia Computer Science* 65 (2015). International Conference on Communications, management, and Information technology (ICCMIT’2015), pp. 797–806. ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2015.09.027. URL: https://www.sciencedirect.com/science/article/pii/S1877050915028574.

[66] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. “BIRCH: An Efficient Data Clustering Method for Very Large Databases”. In: *SIGMOD Rec.* 25.2 (June 1996), pp. 103–114. ISSN: 0163-5808. DOI: 10.1145/235968.233324. URL: https://doi.org/10.1145/235968.233324.

List of Figures

List of tables