Discovering Meaningful Labelings for RTS Game Replays via Replay Embeddings

Pavan Kantharaju
College of Computing and Informatics
Drexel University
Philadelphia, PA
pk398@drexel.edu

*Santiago Ontañón
College of Computing and Informatics
Drexel University
Philadelphia, PA
so367@drexel.edu

***Abstract*—Real-Time Strategy (RTS) games are an interesting environment to study challenging AI problems, such as real-time adversarial planning and opponent modeling. In this paper we focus on approaches that make use of replay data, which usually encode domain expert knowledge of gameplay. Some of these approaches use supervised learning to learn player/agent strategy models and thus rely on these replays being annotated with specific strategies or other labels. However, replays do not usually contain labels for these strategies. The problem we address in this paper is the automatic discovery of meaningful labeling of replays in RTS games. We address this problem by learning action and replay embeddings via recursive neural network models such as LSTMs. These embedded replays can then be clustered to discover labelings by using the clusters as the labels. We show that we can learn embeddings and discover labelings for replays that are correlated with meaningful information from those replays.**

***Index Terms*—Real-Time Strategy Games, Deep Learning, Unsupervised Learning**

I. Introduction

Real-Time Strategy (RTS) games such as Starcraft and Age of Empires are a very successful genre of video games. From an AI point of view, they provide an interesting research challenge as they require real-time adversarial planning with partial observability over vast state and action spaces. For that reason, they additionally provide a strong testbed for studying real-time challenging AI problems, such as adversarial planning or opponent modeling, and AI subfields such as reinforcement learning [1] and AI planning [2] have used them as testbeds.

Many AI approaches to play RTS games make use of replay data (sequences of game state / action pairs seen over the course of a game session) [3]–[5]. These replays are usually generated from professional gameplay or game-playing agents and encode domain expert knowledge of gameplay. Some of these approaches, such as supervised learning ones for player/agent strategy modeling, rely on replays being annotated with specific strategies. Examples of this work include that of Combinatory Categorial Grammar (CCG)-based planning [6] and plan recognition [7]. However, replays do not usually contain labels for these strategies. This makes it challenging to apply supervised learning to replay modeling. This problem is further exacerbated by the fact that labeling replays accurately requires domain-expert knowledge. In particular, these works assumed that player strategies extracted from replays were primarily related to the agent, where training data (i.e. strategies) used the corresponding agent as a proxy label. Even though these proxy labels are reasonable, they may not be meaningful.

The problem we address in this paper is thus the automatic discovery of meaningful labeling of replays in RTS games. In order to address this problem, we use unsupervised deep learning techniques to automatically construct embeddings that "summarize" replays. In particular, our approach applies word embedding techniques from Word2Vec [8] and sequence modeling approaches such as LSTMs [9] to learn embeddings for symbolic actions extracted from replays. We then use these action embeddings along with state features to train an LSTM with Attention [10] for encoding replays. These encoded replays can then be used with off-the-shelf clustering techniques (such as k-medoids) to discover labelings by using the clusters as the labels. We demonstrate that we can learn meaningful embeddings and discover labelings for replays by assessing the information within clusters using metrics from information theory, providing a deeper understanding of these replays. μRTS[1] is used as our evaluation domain, as large collections of replays from past editions of the μRTS competition are readily available.

This paper is structured as follows. Section II provides background on μRTS and word embeddings. Section III provides a description of our approach on learning replay embeddings. Section IV provides some initial experimental results on clustering the replay embeddings. Section V provides some related work and finally, we conclude in Section VI.

II. Background

This section describes the μRTS testbed and provides a brief background on word embeddings.

A. μRTS

μRTS is a minimalistic Real-Time Strategy game designed to evaluate AI research in an RTS setting [11]. Figure 1

Work partially funded by DARPA (Contract No. HR001119C0128)
*Currently at Google

[1]https://github.com/santiontanon/microrts

978-1-7281-4533-4/20/$31.00 ©2020 IEEE

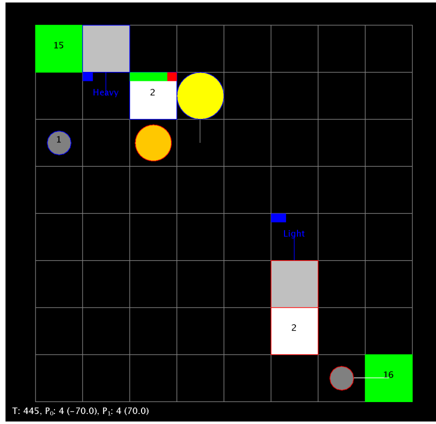


Fig. 1. Screenshot of μRTS gameplay

shows two scripted agents playing against each other in μRTS. Compared to complex commercial RTS games such as *StarCraft*, μRTS still maintains those properties of RTS games that make them complex from an AI point of view (i.e. durative and simultaneous actions, real-time combat, large branching factors, and full or partial observability). In this paper, μRTS games are deterministic, and fully observable. μRTS has been used in previous work to evaluate RTS AI research, [2], [12], [13] and has also been used in AI competitions[2].

B. Word Embeddings

Machine learning algorithms such as LSTMs and Transformers [10] are widely used in Natural Language Processing to solve a variety of language tasks such as textual entailment and question-answering. These algorithms do not directly work on words, but usually on continuous vector representations of words. A naive vector representation of words is one-hot vectors, but these vectors do not capture information about a word such as syntax and semantics. A *word embedding* is a d-dimensional vector that both represents a word and captures such information. Word embeddings can be static, where the embedding is constant regardless of context, or contextualized, where the embedding is a function of all words in a sentence or sequence. There are a multitude of ways to compute word embeddings, such as ELMo [14] and BERT [15] (contextualized), or GloVe [16] (static).

Our work focuses on Word2Vec, which is a set of shallow neural network architectures that learn to transform words to static word embeddings. In this work, we use Word2Vec on tokens from symbolic actions instead of words. There are two architectures in Word2Vec, *continuous bag of words* and *skip gram*, each with their own separate objective functions. In this work, we focus on *Skip Gram with Negative Sampling* [8]. Skip Gram aims to predict tokens within a context window $x_{t-k} \dots x_{t+k}$ given token x_t using the following objective:

$$\frac{1}{T}\sum_{t=1}^{T}\sum_{-k\leq j\leq k; j\neq 0} log(p(x_{t+j}|x_t))$$

where k is the context window size, x_t is the current token, x_{t+j} is the context token, and the probability $p(x_{t+j}|x_t)$ is computed using the softmax function. Negative Sampling extends the Skip Gram learning objective with noise words sampled from a noise distribution, where the objective is to additionally distinguish actual words from these noisy words. In this work, $k = 1$ and the number of noisy words is 5.

[2]https://sites.google.com/site/micrortsaicompetition/home

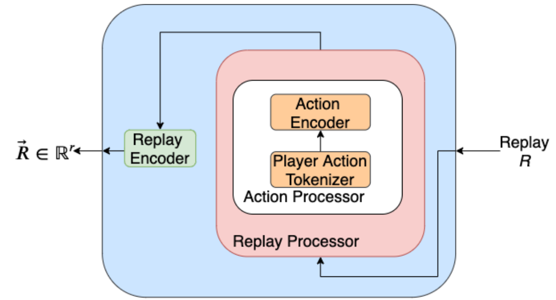


Fig. 2. Replay Encoding Module (REM)

TABLE I
μRTS UNIT ACTION TYPES AND THEIR SYMBOLIC REPRESENTATION

Action Name	Parameters	Symbolic Representation
Harvest	Worker, Direction	**harvest**(worker, direction)
Return	Worker, Direction	**return**(worker, direction)
Attack	Unit, X-Coordinate, Y-Coordinate	**attack**(unit)
Produce	Unit, Direction, Unit Type	**produce**(unit, direction, unit type)
Move	Unit, Direction	**move**(unit, direction)

III. Approach

Our approach aims to learn meaningful labelings for RTS replays by applying clustering analysis on replay embeddings. Past work has looked at encoding replays from Starcraft using hand-crafted feature vectors [17], [18]. Here, we assume that each replay represents a single player strategy and encode RTS replays from μRTS using recursive neural networks.

Figure 2 provides the architecture for the Replay Encoding Module (REM). The two main components are the *replay processor* and *replay encoder*. The replay processor takes in an RTS replay, and constructs an embedding of the state-action pairs in the replay. The replay encoder then takes this processed replay and constructs an r-dimensional embedding.

A. Replay Description

When dealing with replays in μRTS, we distinguish two separate concepts: *unit* actions and *player* actions. The first type, unit actions, are those performed by individual units in game (workers, military units, etc.). In the default settings of μRTS there are six different types of units: *worker*, *barracks*, *base*, *heavy*, *light*, and *ranged*. Moreover, in order to generate action and replay embeddings using neural networks, we want to represent these actions as sequences of symbolic tokens.

TABLE II
STATE FEATURES FOR μRTS

Feature Type	Feature Indices	Description
Player-Specific	0-5	# of Each Unit Type
	6-11	Health of Each Unit Type
	12	Worker Resources
	13	Base Resources
General	14	Height × Width of Map
	15	# of non-player resources

Table I defines the set of unit actions types that are used in this work and the symbolic representation we use for them, syntactically defined as **action name**($parameters$). The sequence of tokens used to represent this symbolic representation is what we will ultimately use for generating action and replay embeddings. For example, for the action name **Harvest** which requires a worker unit and a direction to harvest, the symbolic representation is **harvest**($worker, direction$) (as seen in Table I). We note that some parameters are not in the symbolic representation of the action, since we ignore them in this work. For example, the **Attack** action contains two numeric parameters (X-Coordinate and Y-Coordinate), but the symbolic action ignores these coordinates and only uses the *Unit* parameter. The **idle** action is ignored in this work.

The second type of actions are *player actions*. In μRTS, a player action is the set of unit actions that a player issues to all of the units it controls at a given time step. Player actions are syntactically similar to unit actions. Specifically, we ordered a set of unit actions lexicographically based on their unit action name, and set the **action name** as the concatenation of the unit actions names, and the *parameters* as the aggregation of the parameters from each unit action. We used this representation as it coincides with the syntax of a unit action. An example set of unit actions is {**attack**($light_1$), **produce**($base_1, down, worker$)}, and the corresponding representation we use for a player action would be **attack-produce**($light_1, base_1, down, worker$), where $base_1$, $light_1$ are units, $down$ is a direction, and $worker$ is a unit type.

Given these, we define a *replay* as a sequence of game state and player action pairs seen over the course of a game session $R = [(s_0, a_0), \ldots, (s_n, a_n)]$, where $1 \leq t \leq n$ is a game frame, s_t is the current game state and a_t are player actions done by a player. In this work, we assume all replays are extracted from 2-player games. As such, two replays will be constructed per game, one for each player. Replays can be generated through gameplay with either humans or agents. We generate replays using several game-playing agents for μRTS.

B. Replay Processor

We start by defining the replay processor, which takes a replay R and re-represents it (R') for consumption by our neural network architecture. In particular, we encode states and actions differently. States are encoded using a set of hand-crafted features, and actions are encoded using an *action processor*, which utilizes a combination of Word2Vec and Bi-LSTMs. The state features are used to provide context for the action embeddings (i.e. in what state was the action applied). Finally, the state and action vectors are concatenated. During training, the replay processor will convert a dataset D_{REM} of replays passed into REM into D'_{REM}.

Table II provides the 16 state features, which can be categorized into *player-specific* and *general* features. For the *player-specific* features, there are 6 features related to the number of units for each unit types, 6 related to total health of units for each unit type, 1 for the number of resources held by a worker, and 1 for the number of resources held by a base. For the *general* features, there is 1 for the map size, and 1 for the number of non-player resources. Map size is height times width to keep dimensionality at a power of 2.

The replay processor constructs $R' = [\vec{\phi_1}, \ldots, \vec{\phi_n}]$, where $\vec{\phi_t} = [f_t; \vec{a_t}]$ represents the concatenation of f_t and $\vec{a_t}$ ($R' \in D'_{REM}$ during training). Here, $\vec{a_t} \in \mathbb{R}^z$ is an embedding of the action a_t constructed by the action processor and $f_t \in \mathbb{R}^{16}$ is a normalized feature vector of the game state. This processed replay is then passed into the replay encoder.

1: **procedure** PLAYERACTIONTOKENIZER(a)
2: Syntax of a: **action name**(*parameters*)
3: $a_{name} \leftarrow$ player action name of a
4: $a_{param} \leftarrow$ parameters of a
5: $T \leftarrow$ list of unit action names extracted from a_{name}
6: **for** $p \in a_{param}$ **do**
7: **if** p is a unit **then**
8: $q \leftarrow$ Unit type of p
9: **if** p is a unit type **then**
10: $q \leftarrow p$ + "type"
11: **if** p is a direction **then**
12: $q \leftarrow p$
13: $T \leftarrow append(T, q)$
14: Return T

Fig. 3. Player Action Tokenizer

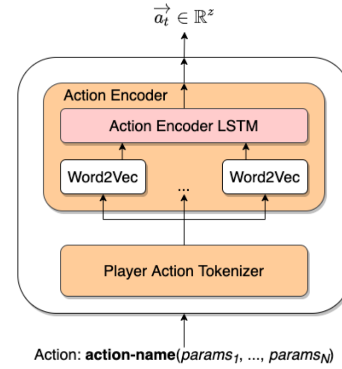


Fig. 4. Action Processor

C. Action Processor

The purpose of the action processor is to transform each of the symbolic player actions from each replay ($a_1 \ldots a_n$) into a z-dimensional embedding. Figure 4 provides a visualization of the action processor. To construct these embeddings, the action

processor first tokenizes each of the player actions (*player action tokenizer*), and then uses this tokenization to construct the embedding (*action encoder*).

Player Action Tokenizer: The player action tokenizer transforms a replay R into $R' = [(s_0, T_0), \dots, (s_n, T_n)]$, where T_t is a list of tokens for player action a_t constructed by the algorithm in Figure 3. Specifically, the player action tokenizer takes as input a player action a and constructs a list of tokens T as follows. First, the action tokenizer splits the player action into its action name and parameters. Next, the player action name is further split into a list of unit action names, and added to T. Then, the tokenizer extracts each token from the parameters, adds them to T, and returns T. We note that some of these parameter tokens are unique objects in the game (such as $worker_1$) and those not found during training will not be encodable. The tokenizer addresses this by replacing these unique objects with their type and replacing unit types with the unit type + "type."

For example, suppose we have the player action represented by the following symbolic representation (see Section III-A): **attack-produce**$(down, base_1, light_1, worker)$. The algorithm in Figure 3 will first split the action into a_{name} = **attack-produce** and $a_{param} = \langle light_1, base_1, down, worker\rangle$. Next, the algorithm splits a_{name} into **attack** and **produce**, and adds them to T ($T = [\textbf{attack}, \textbf{produce}]$). Next, each parameter is traversed and converted into tokens. In this example, we add $light$, $base$, $down$, and $workertype$ to T yielding $T = [\textbf{attack}, \textbf{produce}, light, base, down, workertype]$.

Action Encoder: The list of tokens T_t for player action a_t is viewed as a sequence of words in a sentence. Thus, a single Bidirectional LSTM (Bi-LSTM) is used for action encoding, where a z-dimensional embedding is constructed by averaging the final cell states $c_{f,m} \in \mathbb{R}^z$ and $c_{b,m} \in \mathbb{R}^z$, where $m = |T_t|$. These final cell states can be viewed as containing a "summary" of the player actions from the two directions of traversal (forward for $c_{f,m}$, backward for $c_{b,m}$).

To use T_t for encoding a player action, the action encoder converts each token in T_t into a d-dimensional embedding. This is done using the *Skip Gram with Negative Sampling* (SGNS) [8] architecture from Word2Vec. Word2Vec is trained using a list of sequences of tokens $[\mathcal{T}_1, \dots, \mathcal{T}_k, \dots, \mathcal{T}_{|D_{REM}|}]$, where each sequence $\mathcal{T}_k$ is constructed from $R_k = [(s_0, T_0), \dots, (s_n, T_n)]$ $(1 \leq k \leq |D_{REM}|)$. For each R_k, $T_0 \dots T_n$ are extracted and a special end token **END** referring to the end of the sequence is added to each token list. Next, the tokens in $T_0 \dots T_n$ are aggregated into an ordered list of tokens $\mathcal{T}_k$, which represent all tokens found in R_k. Finally, an **END** token is added to the tail of $\mathcal{T}_k$.

Given the trained Word2Vec network, an action embedding is constructed by first embedding each token $q_j \in T_t$ using Word2Vec. This results in a sequence of token embeddings $E = [\vec{q_1}, \dots, \vec{q_m}]$, where $\vec{q_j} \in \mathbb{R}^d$. E is then passed into the Bi-LSTM to construct a z-dimensional action embedding. We note that during training, a d-dimensional embedding of the **END** token is added to the tail of E: $[\vec{q_1}, \dots, \vec{q_m}, \vec{END}]$.

The action encoder is trained to predict the next token embedding $\vec{q_{j+1}}$ given $\vec{q_1} \dots \vec{q_j}$. In particular, we train the model to minimize the loss of the cosine similarity between the predicted embedding $\vec{q_{j+1}}'$ and the next embedding $\vec{q_{j+1}}$. To do this, we use an additional linear layer above the Bi-LSTM. This linear layer is passed over the hidden states from the Bi-LSTM, where each hidden state is a concatenation of the hidden states from each LSTM $h_t = [h_{f,t}; h_{b,t}]$. This linear layer transforms the hidden state $h_t \in \mathbb{R}^{2z}$ into $\mathbb{R}^d$. Finally, a z-dimensional action embedding is constructed by averaging the final cell states $c_{f,m} \in \mathbb{R}^z$ and $c_{b,m} \in \mathbb{R}^z$, where $m = |T_t|$.

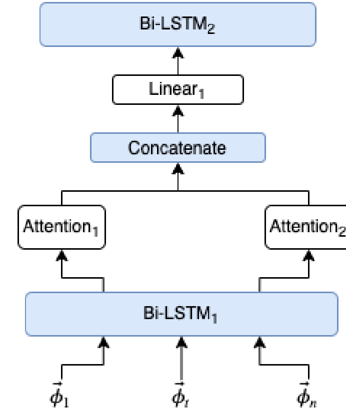


Fig. 5. Replay Encoder Neural Network Architecture

D. Replay Encoder

Similar to the action encoder, the replay encoder encodes whole replays into r-dimensional vectors. A key challenge with replays is their length, where a single replay can be 3000 or more state, action pairs. As such, a single Bi-LSTM will not be sufficient for constructing replay embeddings. Therefore, we augment the Bi-LSTM with two attention layers and an additional Bi-LSTM that constructs the final replay embedding. Figure 5 provides the replay encoder, which consists of 2 Bi-LSTMs (*Bi-LSTM*$_1$ and *Bi-LSTM*$_2$) and 2 Self-Attention layers (*Attention*$_1$ and *Attention*$_2$).

More specifically, the replay encoder works as follows. First, the input $[\vec{\phi_1}, \dots, \vec{\phi_n}]$ ($\vec{\phi_t} \in \mathbb{R}^{z+16}$) from a processed replay R constructed by the replay processor described in Section III-B is passed into a Bi-LSTM (*Bi-LSTM*$_1$ in Figure 5), which results in a sequence of hidden states $h_1 \dots h_t \dots h_n$, where $h_t = [h_{f,t}; h_{b,t}]$ is a concatenation of the two LSTM hidden states at time t. As such, each $h_t \in \mathbb{R}^{2r}$. Next, this sequence is passed into two self-attention networks. The attention layers output two $\mathbb{R}^{n \times r}$ matrices, and when concatenated together, yield a $\mathbb{R}^{n \times 2r}$ matrix. This is then passed into a linear feed-forward layer ($Linear_1$ in Figure 5), yielding a sequence of n vectors of size r. Note that each of these vectors correspond to the state-action embedding pairs $\vec{\phi_t}$. These n vectors are then passed into a second Bi-LSTM (*Bi-LSTM*$_2$ in Figure 5), where a r-dimensional replay embedding is constructed by averaging the final cell states $c_{f,m} \in \mathbb{R}^r$ and $c_{b,m} \in \mathbb{R}^r$ from *Bi-LSTM*$_2$.

Given $R \in D'_{REM}$, we train the replay encoder to minimize the loss in predicting the next state-action embedding $\vec{\phi}_{t+1} \in R$ given $\vec{\phi}_1 \dots \vec{\phi}_t$ using the cosine similarity as the loss function. As such, the training input for the replay encoder is $[\vec{\phi}_1, \dots \vec{\phi}_n, \vec{\phi_{END}}]$, where $\vec{\phi_{END}} = [f_n; \vec{END}]$ represents the end of the replay. Training the replay encoder is similar to that of the action encoder, where an additional linear layer is used above Bi-LSTM$_2$. Specifically, a linear layer is passed over the hidden states from the Bi-LSTM, transforming the hidden state $h_t \in \mathbb{R}^{2r}$ into $\mathbb{R}^{z+16}$.

IV. Experiments

The objective of our experiments is to demonstrate that the embeddings learned by REM can be used to discover meaningful labelings from replays for the RTS game μRTS. To this end, we present results using three separate datasets of μRTS game replays, where the players were different AI agents. We generate labelings for each of the three datasets and then compare those labelings against a collection of known features such as which agent was playing, and in which map they were playing, among others, in order to try to understand what is it that the generated labels represent.

A. Experiment Setup

We use off-the-shelf clustering algorithms to discover labels. Particularly, we use k-medoids clustering, defined as follows:

- *Number of cluster:* varied
- *Initial Medoids:* Randomly chosen with a seed
- *Maximum number of iterations for clustering:* 10000
- *Distance metric:* Squared Euclidean distance

We also employ several additional publicly-available libraries. Specifically, we use the *gensim* library [19] for Skip Gram with Negative Sampling with the configuration described in Section II-B (5 noise words, context window of 1, and embedding size of 16). We also use the Bi-LSTM, Attention, and linear layer from Tensorflow 2.1.0.

Both the action and replay encoders use the cosine similarity loss function with Adam [20] as the optimizer (learning rate $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$). Additionally, both are trained using a batch size of 32 and construct vectors of size 32. We trained the action encoder for 1 epoch and the replay encoder for 20 epochs. The replay encoder input length is restricted to 512 due to memory constraints.

We use a series of μRTS replay datasets for these experiments. We note that all replays are generated from 2-player, fully-observable games. Our neural networks are trained on Training Dataset, and evaluated and used to generate labels for Datasets 1, 2 and 3, described below:

Training Dataset: The training dataset comes from the CoG 2019 μRTS competition replay data[3]. The competition consisted of three tracks (*standard*, *non-deterministic*, and *partially-observable*), where each track involved a 5-iteration round robin tournament played on 8 open maps plus 4 hidden maps. In this work, we use replay data from the 12 agents in the *standard* track and the 8 open maps. We use the first three iteration for training REM, which contains 6336 replays.

[3]https://sites.google.com/site/micrortsaicompetition/competition-results/2019-cog-results

TABLE III
μRTS Maps for *Dataset 3*

Map Name	Map Size	Max # of Game Iterations
(4)Fortress.scxA	128×128	12000
GardenOfWar64x64	64×64	8000
melee14x12Mixed18	14×12	4000
barricades24x24	24×24	5000
basesWorkers16x16noResources	16×16	4000
itsNotSafe	15×14	4000
letMeOut	16×8	4000
chambers32x32	32×32	6000

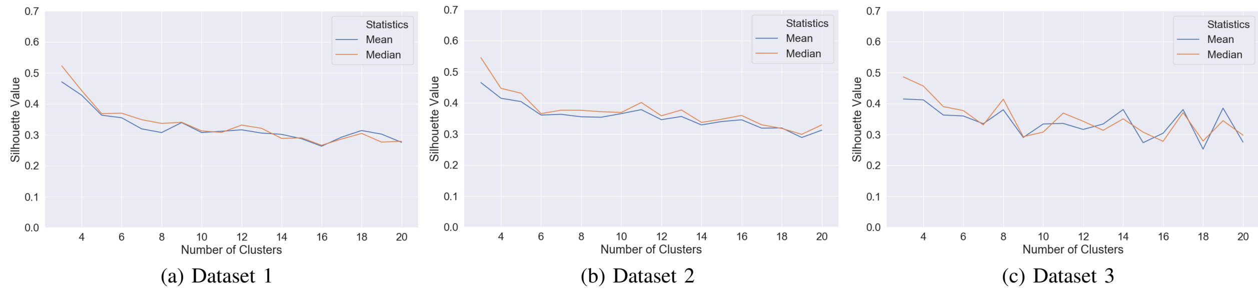
Dataset 1: The next dataset consists of the last two iterations of the standard track of the CoG 2019 μRTS competition replay data, and contains 4224 replays.

Dataset 2: The second testing replay dataset contains 1936 replays and consists of a single iteration round-robin tournament with 11 built-in game-playing agents playing on the 8 open maps of the CoG 2019 μRTS competition. The built-in scripted agents were *POLightRush*, *POHeavyRush*, *PORangedRush*, *POWorkerRush*, *EconomyMilitaryRush*, *EconomyRush*, *HeavyDefense*, *LightDefense*, *RangedDefense*, *WorkerDefense*, *WorkerRushPlusPlus*, and each agent played against each other as both player 1 and player 2. We only required a single iteration tournament as most of these agents are deterministic. Additionally, we note that *POLightRush* and *POWorkerRush* were also used in the CoG 2019 μRTS competition.

Dataset 3: The final replay dataset contains 1296 replays and is an adaptation of the second one. In particular, use the same built-in agents except *POLightRush* and *POWorkerRush*, and 8 new maps not found in the first and second dataset. As such, all agents and maps in this dataset are not in the training dataset. Table III provides information on these 8 new maps. We note that *(4)Fortress.scxA* is the largest map of all three datasets and several of the maps in Table III are not perfect squares. This is important as 7 of the 8 maps in the training dataset are perfect squares. As such, we expect replays in this dataset to be more challenging to encode. All other maps in Table III are from the hidden maps of the 2019 and 2018 tournament. Agents played on these maps until one won or the maximum number of game iterations elapsed.

We next provide a description of the metrics we use to evaluate the resulting labelings. The first metric, *silhouette value*, is used to evaluate the quality of the generated clusters. Intuitively, the silhouette value for a given data point measures the similarity of points in its own cluster and those in other clusters. This similarity is computed using a distance metric (in our work, this is euclidean distance), and ranges from $[-1, 1]$, where -1 means the data point is in the wrong cluster and 1 means the data point is close to its own cluster and far from others. We compute the mean and median silhouette values using the silhouette values from each data point.

The second metric, Variation of Information (VI) [21], is used to evaluate the relationship between the information

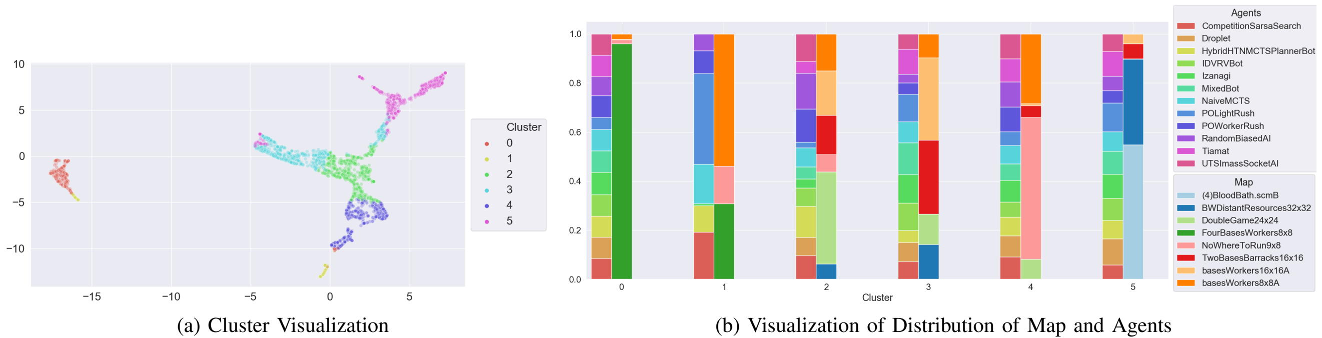


(a) Dataset 1 (b) Dataset 2 (c) Dataset 3

Fig. 6. Number of Clusters vs Silhouette Value for Datasets 1, 2 and 3

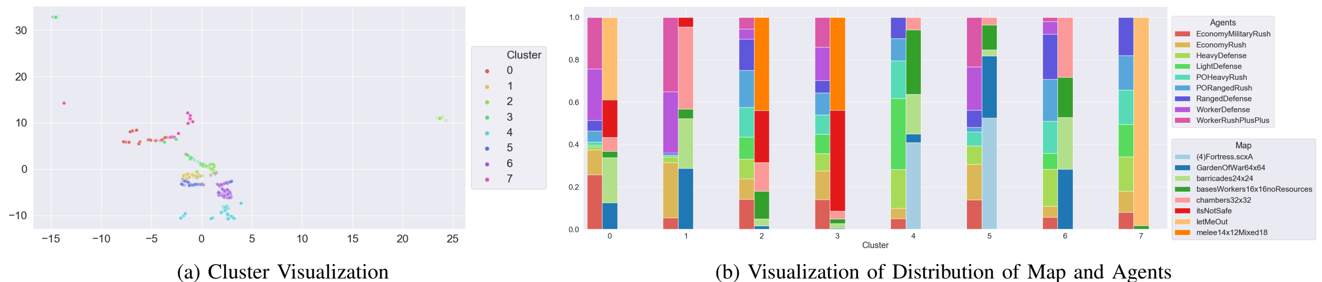
(a) Dataset 1 (b) Dataset 2 (c) Dataset 3

Fig. 7. Variation of Information for Dataset 1, 2, and 3 (Lower=Better)



(a) Cluster Visualization (b) Visualization of Distribution of Map and Agents

Fig. 8. Visualization of Clustering and Distribution of Map and Agents for Dataset 1



(a) Cluster Visualization (b) Visualization of Distribution of Map and Agents

Fig. 9. Visualization of Clustering and Distribution of Map and Agents for Dataset 3

contained in the replay embeddings and clusters. VI measures the distance between two partitions of a dataset. In this work, the first partition is the clusterings and the second partition is different pieces of information extracted from the replays, including *agents*, *agent opponents*, *map*, and *map size*. Here, the smaller the distance, the more closely related the clusters are to a particular piece of information. More formally, this is computed as follows $VI(X,C) = H(X,C) - I(X,C)$, where X and C are discrete random variables representing information and the clusters, $H(X,C)$ is the joint entropy,

and $I(X, C)$ is mutual information.

B. Experiment Results

Figures 6a, 6b and 6c provide the silhouette values from k-medoids clustering for varying number of clusters (3 to 20) for Datasets 1, 2 and 3. Overall, we see that the silhouette value decreases as the number of clusters increases. A silhouette value of 0 implies that clusters are overlapping. However, we see that the silhouette values are higher than 0.25, which implies that k-medoids was able to separate the replay embeddings. Further, we notice that Dataset 3's curve is not a smooth decline like Datasets 1 and 2. This is a result of clusters with higher than uniform number of points having higher silhouette score than other clusters. This caused some number of clusters to have higher silhouette scores. These graphs can be used to determine a natural number of clusters for k-medoids. Specifically, the best number of clusters is the minimum set of clusters where adding additional clusters will not improve the clustering (i.e. no need for additional partitioning of the data). This would thus be the elbow point of the graph. The natural number of clusters is 6 for Datasets 1 and 2, and 8 for Dataset 3.

Next, we look at the information learned by the embeddings. Figures 7a, 7b and 7c provide the variation of information (VI) for Datasets 1, 2, and 3 on types of information extracted from the replays, including *agents*, *agent opponents*, *map*, and *map size*. Here, lower VI is better as it corresponds to types of information being closely related to the clusters. Overall, we see that VI increases with the number of clusters. This makes sense as more clusters would result in different clusters having similar information (e.g. multiple clusters containing the map *(4)Fortress.scxA*), thereby making it difficult to relate a type of information to a cluster. We also see that the clusters correspond more to map or map size than agents or their opponents. This implies that, despite their specific behavior, agents play based on the size and structure of a map (or at least that this has a larger influence in the replay embeddings). We also see that the agent itself has a larger influence in the labeling than the opponent, which is expected.

Figures 8a and 8b provide a visualization of clustering and distribution of agent (left bar) and map (right bar) in each cluster for Dataset 1. Each colored bar represents some fraction of the cluster. For example, in cluster 5 of Figure 8b, approximately half of the cluster corresponded to the *(4)BloodBath.scmB* map (right bar, light-blue). Here, we use UMAP [22] to reduce the dimensionality of the embeddings for cluster visualization. We see that cluster 0 (red) is an entire island of points and the remaining clusters separate a large island of points. Specifically, cluster 0 corresponds to all agents playing on the map *FourBasesWorkers8x8* according to Figure 8b. This clustering makes sense as the map doesn't allow for many elaborate strategies except to rush the opponent due to its size and structure. We also see that cluster 2 (green) separates cluster 3 (light-blue). Cluster 3 is correlated with agents playing on the maps *BWDistantResources32x32*, *DoubleGame24x24*, *TwoBasesBarracks16x16*, *basesWorkers16x16A*, and *basesWorkers8x8A*. We believe that the larger set of points refers to *TwoBasesBarracks16x16*, *basesWorkers16x16A*, and *basesWorkers8x8A* while the other points refer to the remaining maps. This makes sense as the maps in the former set mostly differ in size or entities on the map. Specifically, the difference between *TwoBasesBarracks16x16* and *basesWorkers16x16A* is that there is an additional base and two barracks for each player in the former.

Figures 9b and 9b provides a visualization of clustering and distribution of agent and map in each cluster for Dataset 3. Here, we see that cluster 1 and 5 represent playing on larger maps with a worker-type strategy while clusters 4 and 6 refer to non-worker type strategies. We define a worker type strategy as one that only requires worker units to execute. Non-worker type strategies require constructing barracks and offensive units such as ranged, light or heavy. Thus, it makes sense that worker-type strategies have their own clusters separate from agents that construct offensive units.

These results indicate that both the replay embeddings and clusters contain meaningful information about the replays. In particular, the replay embeddings encode a lot of map information and some agent information. This is illustrated in the visualizations of agent and map distributions in Figures 8b and 9b. We also note that the replay embeddings capture similar information irregardless of the agents and maps used for generating replays (as can be seen with the results in Dataset 3, which use a completely disjoint set of maps). This demonstrates that our approach is robust and should be applicable to replays with unseen agents and maps.

V. Related Work

Learning action embeddings using deep learning is not a new idea. For example, Tennenholtz and Mannor [23] present Act2Vec, an approach that uses contextual information (action histories) to learn action embedding. Specifically, the action being converted and the action context are provided as input to Word2Vec (specifically, Skip Gram Model with Negative Sampling). These learned action embeddings are then applied in the context of RL. Our work also learns action embeddings using Word2Vec, but uses those embeddings for replay encoding. Chandak et al. [24] presents a supervised technique for learning action embeddings in the context of Reinforcement Learning (RL). Specifically, they simultaneously learn a policy for getting an action embedding, and a mapping function from embedding space to discrete actions.

Additionally, RTS games in general have been used as testbeds for problems such as reinforcement learning [1], [25], planning [2], [26], and plan recognition [27]. There has also been some prior work on labeling replays and applying them to strategy prediction. Weber [17] modeled opening strategies from Starcraft using hand-crafted feature vectors and used a set of rules to label these openings. Synnaeve and Bessiere [18] similarly uses hand-crafted feature vectors for various Starcraft openings and determined labels for them using clustering. Our work discovers labelings through clustering and encodes replays using deep learning.

Finally, our work is also related to the general area of player modeling, which aims to describe properties of players such as strategies or knowledge (see Machado et al. [28] for an overview of the area). Many works [29]–[31] apply machine learning over gameplay features extracted from logs to learn these player models. Our approach instead learns features from replays to then subsequently find labelings for replays.

VI. Conclusion

The problem we addressed in this paper is the automatic discovery of meaningful labeling of replays for the RTS game μRTS. Our contribution is the Replay Encoding Module (REM), which applies unsupervised deep learning techniques to automatically construct embeddings that "summarizes" a replay. Encoded replays are then used with k-medoids to discover labelings for the replays using the clusters as the labels. Our experiments demonstrate that we can learn meaningful labelings and embeddings that capture map and agent information from the replays.

For future work, we are interested in studying other deep learning architectures such as Transformers for replay encoding. We are also interested in relaxing the assumption that each replay corresponds to a single strategy as it is possible that agents many execute multiple, possible interleaved strategies within a replay. Finally, we also want to look at ways to manipulate the type of information in the embeddings and how to apply these learned labelings for playing μRTS.

References

[1] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, J. Oh, D. Horgan, M. Kroiss, I. Danihelka, A. Huang, L. Sifre, T. Cai, J. P. Agapiou, M. Jaderberg, A. S. Vezhnevets, R. Leblond, T. Pohlen, V. Dalibard, D. Budden, Y. Sulsky, J. Molloy, T. L. Paine, C. Gulcehre, Z. Wang, T. Pfaff, Y. Wu, R. Ring, D. Yogatama, D. Wünsch, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, K. Kavukcuoglu, D. Hassabis, C. Apps, and D. Silver, "Grandmaster level in StarCraft II using multi-agent reinforcement learning," *Nature*, vol. 575, no. 7782, pp. 350–354, 2019.

[2] S. Ontañón and M. Buro, "Adversarial Hierarchical Task Network Planning for Complex Real-Time Games," in *Proc. of the 24th International Joint Conference on Artificial Intelligence*, 2015, pp. 1652–1658.

[3] G. Synnaeve and P. Bessiere, "A Bayesian Model for Plan Recognition in RTS Games Applied to StarCraft," in *Proc. of 7th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2011, pp. 79–84.

[4] G. Synnaeve, Z. Lin, J. Gehring, D. Gant, V. Mella, V. Khalidov, N. Carion, and N. Usunier, "Forward modeling for partial observation strategy games-a starcraft defogger," in *Advances in Neural Information Processing Systems*, 2018, pp. 10 738–10 748.

[5] S. Ontañón, K. Mishra, N. Sugandh, and A. Ram, "Learning from demonstration and case-based planning for real-time strategy games," in *Soft Computing Applications in Industry*. Springer, 2008, pp. 293–310.

[6] P. Kantharaju, S. Ontañón, and C. W. Geib, "microCCG, a CCG-based Game-Playing Agent for microRTS," in *Proc. of the 2018 IEEE Conference on Computational Intelligence and Games*. IEEE, 2018, pp. 1–8.

[7] ——, "Scaling up CCG-Based Plan Recognition via Monte-Carlo Tree Search," in *Proc. of the IEEE Conference on Games 2019*, 2019.

[8] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proc. of the 26th International Conference on Neural Information Processing Systems*, 2013, pp. 3111–3119.

[9] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[10] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, 2017, pp. 5998–6008.

[11] S. Ontañón, "The combinatorial multi-armed bandit problem and its application to real-time strategy games," in *Proc. of the 9th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2013, pp. 58–64.

[12] A. Shleyfman, A. Komenda, and C. Domshlak, "On combinatorial actions and CMABs with linear side information," in *Frontiers in Artificial Intelligence and Applications*, vol. 263. IOS Press, 2014, pp. 825–830.

[13] J. R. Marino, R. O. Moraes, C. Toledo, and L. H. Lelis, "Evolving action abstractions for real-time planning in extensive-form games," in *Proc. of the 2018 AAAI Conference on Artificial Intelligence*, 2018.

[14] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," in *Proc. of 2018 North American Chapter of the Association for Computational Linguistics*, 2018.

[15] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.

[16] J. Pennington, R. Socher, and C. D. Manning, "Glove: Global vectors for word representation," in *Proc. of 2014 Conference on Empirical Methods in Natural Language Processing*. Citeseer, 2014.

[17] B. G. Weber and M. Mateas, "A data mining approach to strategy prediction," in *2009 IEEE Symposium on Computational Intelligence and Games*. IEEE, 2009, pp. 140–147.

[18] G. Synnaeve and P. Bessiere, "A Bayesian Model For Opening Prediction in RTS Games With Application To Starcraft," in *Proc. of 2011 IEEE Conference on Computational Intelligence and Games*. IEEE, 2011, pp. 281–288.

[19] R. Rehurek and P. Sojka, "Software Framework for Topic Modelling with Large Corpora," in *Proc. of the LREC 2010 Workshop on New Challenges for NLP Frameworks*. Valletta, Malta: ELRA, may 2010, pp. 45–50.

[20] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," *Proc. of the 3rd International Conference on Learning Representations*, 2015. [Online]. Available: http://arxiv.org/abs/1412.6980

[21] M. Meilă, "Comparing clusterings by the variation of information," in *Learning theory and kernel machines*. Springer, 2003, pp. 173–187.

[22] L. McInnes, J. Healy, and J. Melville, "UMAP: Uniform Manifold Approximation and Projection for Dimension Reduction," *ArXiv e-prints*, feb 2018.

[23] G. Tennenholtz and S. Mannor, "The Natural Language of Actions," *Proc. of the International Conference on Machine Learning*, 2019. [Online]. Available: http://arxiv.org/abs/1902.01119

[24] Y. Chandak, G. Theocharous, J. Kostas, S. Jordan, and P. S. Thomas, "Learning Action Representations for Reinforcement Learning," *Proc. of the 36th Interational Conference on Machine Learning*, 2019.

[25] S. Xu, H. Kuang, Z. Zhi, R. Hu, Y. Liu, and H. Sun, "Macro action selection with deep reinforcement learning in starcraft," in *Proc. of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 15, no. 1, 2019, pp. 94–99.

[26] S. Ontañón, "Combinatorial multi-armed bandits for real-time strategy games," *Journal of Artificial Intelligence Research*, vol. 58, no. 1, pp. 665–702, 2017.

[27] F. Kabanza, P. Bellefeuille, F. Bisson, A. R. Benaskeur, and H. Irandoust, "Opponent Behaviour Recognition for Real-Time Strategy Games," in *Workshop of 24th AAAI Conference on Artificial Intelligence*, 2010, pp. 29–36.

[28] M. C. Machado, E. P. C. Fantini, and L. Chaimowicz, "Player modeling: Towards a common taxonomy," in *Proc. of the 16th International Conference on Computer Games*. IEEE, 2011, pp. 50–57.

[29] P. Harrison and D. Roberts, "Identifying Patterns in Combat that are Predictive of Success in MOBA Games," in *Proc. of the 2014 Foundations of Digital Games*, 2014.

[30] H. P. Martínez, K. Hullett, and G. N. Yannakakis, "Extending Neuro-Evolutionary Preference Learning through Player Modeling," in *Proc. of the 2010 IEEE Conference on Computational Intelligence and Games*. IEEE, 2010, pp. 313–320.

[31] J. Valls-Vargas, S. Ontanón, and J. Zhu, "Exploring Player Trace Segmentation for Dynamic Play Style Prediction," in *Proc. of the 11th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2015.