Strategy Extraction for Transfer in AI Agents

Anonymous submission

Abstract

We propose an approach to knowledge transfer for improved lifelong learning in AI agents, using behavioural strategies as a form of transferable knowledge, influenced by the human cognitive ability to develop strategies. A strategy is defined as a partial sequence of actions that an agent can take to reach some predefined event of interest. This information acts as guidance or partial solutions that an agent can generalise and use to make predictions about how to handle unknown observed phenomena in future deep learning tasks. As a first step toward this goal, we present an approach for extracting strategies from an agent's existing knowledge that can be applied in multiple contexts. Our approach uses a combination of observed action frequency information, with local sequence alignment techniques, to find patterns of significance that form a strategy. We demonstrate our approach in two environments: Pacman; and a dungeon-crawling video game.

1 Introduction

Transfer learning research aims to minimise the re-training requirements for a system to operate in a new context when it has been previously trained in a similar context. The general approach is to transfer some knowledge structure that has been learned from a source task which, with minimal modification, can be used to solve a target task. This technique is common in classification or clustering tasks where a model that has been trained with some available data can be adapted for another similar domain where less data is required (Weiss, Khoshgoftaar, and Wang 2016). An extension of transfer learning is lifelong learning where an agent learns to solve a sequence of tasks over its lifetime. In such cases, the agent retains transferred knowledge which it can leverage (Chen and Liu 2018; Parisi et al. 2019).

Humans have the natural ability to develop strategies in one context that can be generalised and applied in other contexts, providing a useful starting point to form a complete solution in unfamiliar settings (Collins and Frank 2013). Suppose artificial agents have the capability to perform such strategy synthesis, allowing them to better utilise their existing knowledge to continuously learn and react to novel situations. In this work, we consider existing knowledge in the form of policies learnt through deep reinforcement learning (Zhu et al. 2020). This would greatly improve the agent's learning capabilities to handle a wide range of tasks with limited data where it can apply generalised strategies to guide behaviour. In this work, we propose a method of knowledge transfer in AI agents based on the human cognitive ability to develop strategies. As a first step, we seek to obtain transferable knowledge, in the form of strategies, from an agent's existing knowledge. In this paper, we form strategies by extracting information from game trajectories.

A strategy is an abstract task structure that represents a partial plan to achieve a goal in some environment. We adopt an intuitive definition of a strategy rather than the game theoretical definition referred to in multi-agent studies (Parsons and Wooldridge 2002). A partial plan refers to the usability of the solution–it may not completely solve a problem but it provides a base for a complete solution. Our definition encompasses plans with a partial ordering of actions as well as plans with potentially unnecessary actions.

The strategies we extract in this paper contain actions that are specific to a given game. Through abstraction, we can broaden the applicability of these strategies. In the game of Pacman, a strategy may be to "collect a power-up to defeat a ghost". This could be abstracted to "collect an item to defeat an enemy". Defeating enemies is a common objective that appears in a variety of games.

Contribution The key contribution of this paper is our unique approach to strategy extraction by treating the problem as a sequential pattern mining task. The novelty lies in the use of a sequence alignment technique known as the Smith-Waterman algorithm (Smith and Waterman 1981) which is more commonly known for its use in the comparison of DNA string sequences. We apply a modified Smith-Waterman algorithm combined with observed action frequencies across a dataset of trajectories to find patterns of significance that form a strategy. We consider case studies based on single-player video game environments to demonstrate the approach and develop performance benchmarks. Our evaluation serves as a promising first step towards efficient and robust generalisation and how to best utilise the generalised strategies in new tasks and complex domains. Strategy generalisation and transfer are left as future work.

2 Related Work

Strategy extraction is typically demonstrated in real-time strategy (RTS) games which provide a testbed for many dif-

ferent strategy-based tasks (Srovnal et al. 2004; Bosc et al. 2013; Chen et al. 2015; Een et al. 2015). There are a number of studies in sequence modelling and pattern mining that leverage information gathered from the analysis of state-action trajectories. Such techniques help to discover patterns in action trajectories related to specific goals. This information can inform our strategy construction.

Most existing work focuses on finding strategies with the purpose of player modelling for applications such as learning an opponent's strategy, understanding a player's behaviour and finding a winning strategy. The types of strategies mainly targeted are game or implementation specific and by themselves cannot be transferred to aid the learning of other games. Several extraction procedures extract strategies based on pattern frequency. For example, Chen et al. (2015) identify closed frequent patterns which are above some pre-specified threshold, Low-Kam et al. (2013) detect significant sequential patterns based on the frequency of pattern prefixes and Bosc et al. (2013) use a combination of pattern prefix and sequential pattern frequency. The types of strategies obtained, although useful, may return the simplest sequence of actions in order to accomplish the task as they do not provide any information at the action level in relation to each actions' significance to achieving the goal. This in turn makes it difficult to generalise the extracted strategy as it may include actions that are irrelevant.

Srovnal et al. (2004) use a text-based method, latent semantic analysis, to detect semantic information which they interpret as a game strategy. Inspired by this idea, our proposed approach uses the sequential pattern mining method known as *local sequence alignment* to find useful causal information which we can use to form strategies. Local sequence alignment is a variant of global sequence alignment which treats sequences as a whole. A notable method is the Smith-Waterman algorithm, first proposed in 1981 by F. Smith and M. S. Waterman after which it was named. In this work, we use this method, more commonly known for its application in bioinformatics to compare strings of nucleic acid or protein sequences, to compare game trajectories.

3 Preliminaries

We consider an agent that has been trained to play a game through reinforcement learning. A single-player game environment consists of a set of states S, actions A and rewards R. At each timestep, t, the player performs an action a_t from the set of available actions and receives some reward r_t. A game trajectory, τ is a finite sequence of consecutive actions, rewards and observed states of the environment before and after the action is taken, s_{t-1} and s_t respectively. A *subtrajectory* is a trajectory containing a subset of the elements from another trajectory, maintaining the same temporal ordering. A subtrajectory may not be equivalent to a subsequence; elements from the original trajectory can be dropped. For example, given a trajectory of the form a, b, c, a valid subtrajectory is a, c as well as a, b and b, c.

Environments

We demonstrate our method on two custom environments, implemented in OpenAI Gym (Brockman et al. 2016).

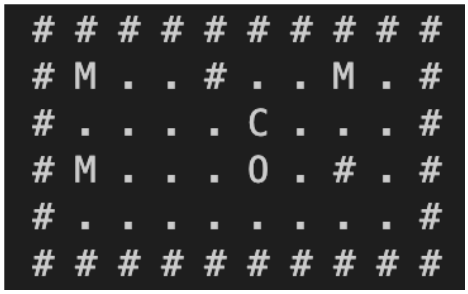

Figure 1: Example Pacman game state: C denotes Pacman's location; M a ghost; # a wall; O a power-up; and . a dot.

Pacman We developed a version of Pacman, as a 2D environment, as shown in Figure 1. An agent has the following actions available: *move, collect power-up/dot* and *kill a ghost*. An observation of the game state is a grid where each cell contains a numerical value representing a game component – walls, ghosts, dots, power-ups and Pacman. Pacman kills a ghost by collecting a power-up and then moving over the ghost within the next five movements.

We use Proximal Policy Optimisation (PPO) (Schulman et al. 2017) to learn a policy for playing Pacman. Rewards are received when the agent collects items (dots and power-ups). Additionally, the agent is rewarded for killing ghosts, with doubled rewards received when killing ghosts in succession. The map topology, and the starting locations of the ghosts, remain unchanged for every episode of training. Ghosts move around the map during the game.

Dungeon Crawler Dungeon Crawler is an exploration game where the goal of the agent is to navigate through a maze to collect a key and then escape through a door. The agent must avoid monsters in its search for the key or kill the monsters by collecting a weapon. Points are rewarded for collecting items (weapons and keys), killing monsters and escaping through the door. The available actions in this environment are *move, collect weapon (gun), collect weapon (sword), collect key, kill a monster* and *unlock door*.

We use PPO to learn a policy for playing Dungeon Crawler. The dungeon map is fixed for every training episode however, the items and monsters are randomly placed. An example of an initial game grid is shown in Figure 2. An observation of the game state contains the shortest-path distances between the agent's current location to various entities (weapons, monsters, the key and the door).

There are multiple events for which we expect to find strategies. The first is the key-door collection strategy. The second is the weapon-monster killing strategy. Multiple weapon types are available (e.g. 'gun' and 'sword'). Two valid strategies in this context may be "collect the gun to kill a monster", and "collect the sword to kill a monster".

Local Sequence Alignment

Suppose we have sequences $A = a_0, a_1, ..., a_m$ and $B = b_0, b_1, ..., b_n$ where m and n are the respective sequence lengths. Local sequence alignment can be applied to compare A and B and find regions where the two sequences are

Figure 2: Example Dungeon Crawler game state: o denotes the location of the player; Z a monster; X a wall; | the door; k the key; and g and s the gun and sword respectively

.

similar to each other. The result is one or more subsequences deemed to be most similar according to some metric.

The Smith-Waterman algorithm performs local sequence alignment by first forming a scoring matrix, populated by comparing the m elements of A with the n elements of B. The matrix dimensions are $(m+1) \times (n+1)$ and the first row and column are filled with zeros. The matrix cell (i, j) where $i \in [1, m+1]$ and $j \in [1, n+1]$ is assigned a score based on the comparison of A_i and B_j, and the values in adjacent matrix cells. In the original algorithm, these scores are computed as per Equation 1, where s, d and g are user-defined *match*, *mismatch* and *gap* scores respectively.

$$H_{AB}(i,j) = max \begin{cases} 0 \\ H_{AB}(i-1, j-1) + s, \text{ if} A_i = B_j \\ H_{AB}(i-1, j-1) + d, \text{ if} A_i \neq B_j \\ H_{AB}(i, j-1) - g \\ H_{AB}(i-1, j) - g \end{cases} \tag{1}$$

When filling out the values in the aforementioned matrix, Smith-Waterman keeps track of which adjacent cell was used to compute the score for cell (i, j). This score, $H_{AB}(i, j)$, will use at most one of $H_{AB}(i-1, j-1)$, $H_{AB}(i, j-1)$, and $H_{AB}(i-1, j)$. This knowledge is later used, through a *traceback* procedure, to form the output of the sequence comparison – a similar subsequence.

The traceback process starts from the highest-scoring cell, (i, j), including either A_i or B_j in our output subsequence. The method then recalls which adjacent cell was used to calculate the value in (i, j), and moves to that cell. For each diagonal movement to a cell (k, l), one of A_k and B_l is placed at the front of the evolving output subsequence. For vertical or horizontal movements, a *gap* token is added. This process continues until a cell with a zero is reached.

We adapt Smith-Waterman to perform pairwise comparisons on trajectories and return a subtrajectory. Given our motivation for comparing trajectories is to highlight important elements, we have made several changes to the original scoring function shown in Equation 1.

First, we maintain match and mismatch scoring since we want to maintain elements that exist in both trajectories and de-emphasize all others.

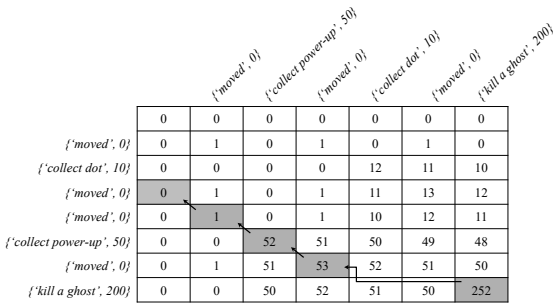

Figure 3: Example Smith-Waterman scoring matrix constructed using Equation 2 where is $s = 1$, $d = -1$ and $\mathcal{W} = max(1, reward(A_i), reward(B_j))$. Trajectory elements from A and B are displayed in terms of their actions and corresponding rewards. The directional arrows depict the recursive traceback process, highlighting the cells which are used to determine the output subtrajectory (shaded).

Second, we discard penalties (g) for the presence of gaps in the resulting subtrajectory. Smith-Waterman aims to find a similar *subsequence* when comparing two sequences, explicitly placing gap tokens between consecutive items in the result if they do not occur consecutively in the two sequences being compared. We are interested in identifying the important similarities (actions) between two trajectories, and their relative temporal ordering. We do not care whether gaps should be placed in the resulting subtrajectory, or how many, in the context of our strategy extraction objective.

Finally, we include a weighting, $\mathcal{W}$, capturing additional information about the elements being compared, beyond whether they match, to further influence which elements appear in our output subtrajectory. We describe in subsequent sections how we instantiate $\mathcal{W}$ when comparing trajectories. These changes are reflected in Equation 2.

$$H_{AB}(i,j) = \begin{cases} H_{AB}(i-1, j-1) + s \times \mathcal{W}, \text{ if} A_i = B_j \\ H_{AB}(i-1, j-1) + d \times \mathcal{W}, \text{ if} A_i \neq B_j \end{cases} \tag{2}$$

Once the scoring matrix is filled, we traceback through the matrix to discover which of its cells will be used to determine the output, following the procedure described earlier. When deciding on whether to include A_i or B_j in our resulting subtrajectory, after a diagonal movement to cell (i, j), we choose the action from the shorter trajectory.

Figure 3 shows an example of a scoring matrix that has been constructed for two Pacman trajectories, excluding state and reward information for simplicity, A ={"move", "collect dot", "move", "move", "collect power-up", "move", "kill a ghost"} and B ={"move", "collect power-up", "move", "collect dot", "move", "kill a ghost"}. In this example, $\mathcal{W} = max(1, reward(A_i), reward(B_j))$, where $reward(A_k)$ denotes the reward received by the agent after performing action A_k in the relevant trajectory. The output subtrajectory for this example is {"move", "collect power-up", "move", "kill a ghost"}, including actions from the shorter of the two trajectories, B.

The Smith-Waterman algorithm is $O(mn)$. Despite its high computational complexity as the length of the compared sequences increases, we prefer this method for three reasons. First, as a local, as opposed to global, sequence alignment method, it does not treat the sequences as a whole and allows for subsequence similarities to be detected. Second, it supports the addition of gaps in the output subsequences to replace one or more sequence elements. While we do not penalise the presence of gaps, or explicitly add gap tokens to our subtrajectories, gap allowance is important. In a trajectory, the actions that are important to a goal may not always occur consecutively. For example, in Pacman, we want to find strategies like "collect a power-up, kill a ghost" rather than explicitly list all the moves and dot collections that occur in between. Finally, the function and metrics used for element comparisons and score assignment are flexible allowing for customisation.

4 Method

In this section, we describe each stage of our proposed strategy extraction approach (Figure 4). Given a policy π for playing a game, $\mathcal{G}$, our approach first identifies *events of interest*. These events occur when playing $\mathcal{G}$, and represent goals or sub-goals that an agent may wish to achieve.

For an event of interest, e, our approach collects two sets of trajectories. The first set contains trajectories in which e occurs, denoted positive trajectories, and the second trajectories in which e does not occur, denoted negative trajectories. The lengths of the negative trajectories are normalised so that the distribution of trajectory lengths of the positive and negative sets match. The positive and normalised negative trajectories are then used to compute the likelihood of each available action being part of a strategy for achieving e. The likelihood for an action is computed by comparing the frequency with which it appears in the positive trajectories relative to the negative. Actions that appear more often in positive trajectories are more likely to be part of a strategy.

Pairs of positive trajectories will ultimately be compared using our Smith-Waterman adaptation. To reduce the complexity of these comparisons, we remove actions whose likelihood falls below a threshold from all positive trajectories. To select pairs of trajectories for comparison, we first cluster trajectories on the basis of the actions they contain. We sample pairs of trajectories from each cluster, performing pairwise comparisons using the adapted Smith-Waterman algorithm. The result for each cluster is a set of candidate strategies for achieving e. The purpose of clustering trajectories on the basis of their actions is to ensure that we identify different strategies for achieving the same e, when present.

Discovering Events of Interest

We first discover events that occur in the environment that the player could use a strategy to achieve. We use a method of detection based on rewards however, this can be replaced with more sophisticated approaches that do not rely solely on the reward function (Asadi and Huber 2007; Paul, van Baar, and Roy-Chowdhury 2019).

Using the policy π, we simulate the PPO agent in the environment for N episodes and gather trajectories from which events of interest are identified. The first $N/2$ episode trajectories are used to obtain an estimate of the average episode reward r_{avg}. The remaining $N/2$ episodes are simulated, searching for actions that receive a reward greater than r_{avg}. These actions form our *events of interest*.

Example 1 (Pacman) *We collect $N = 100$ trajectories to discover events of interest. Across the first half, $r_{avg} = 11$. Across the second half, actions that receive a reward greater than 11 include "collect power-up" (reward = 50) and "kill a ghost" (reward = 200, 400).*

Collecting Trajectory Data

For a selected e, trajectories are collected to form a dataset for strategy extraction. When the PPO agent is simulated in the environment, historical trajectories τ_H are saved. Positive trajectories, τ_P, are collected when the agent reaches an event of interest mid-simulation. The agent's trajectories from the beginning of the episode to the event of interest are saved. To collect negative trajectories, τ_N, we query τ_H for a random set of entire-episode trajectories, filtering out the trajectories which contain the event of interest.

Our approach relies on collecting the same number of positive and negative samples. Suppose this becomes too difficult; for example, if the policy π is trained sufficiently well with respect to a given event of interest such that all trajectories in τ_H are positive. A random agent or a basic heuristic could alternatively be used to guide agent behaviour when generating negative trajectories.

Example 2 (Pacman) *When $e =$"kill a ghost", one possible trajectory from τ_P is {"move", "collect dot", "move", "collect power-up", "move", "kill a ghost"}. A trajectory from τ_N is {"move", "collect dot", "move", "collect dot", "move"}, at which point Pacman is killed by a ghost.*

Normalisation

Depending on the environment and event of interest, there could be significant variation between the length distributions of τ_P and τ_N. For the chosen e, negative trajectories may be longer on average than positive trajectories. In such cases, this obscures the useful information that we wish to gather from comparing the two sets in later stages. Significantly longer negative trajectories influence our calculation of action likelihoods, as these are based on the relative frequency with which specific actions occur across the two sets.

To solve this problem, τ_N is normalised to ensure its length distribution matches that of τ_P. We randomly sample a length from the length distribution of τ_P, l_{τ_P}, and also sample a longer trajectory from τ_N, t_N. A subsequence of the desired length is extracted from t_N and added to the new normalised set, $\tau_N\prime$. We repeat this process until $\tau_N\prime$ contains the same number of trajectories as τ_P.

This normalisation step is not applied if the length distributions are equal, in terms of their respective mean and standard deviation, or if the negative trajectories are, on average, shorter than the positive trajectories. Figure 5 shows an example of applying our normalisation method on trajectories obtained from agent simulations in the Pacman environment.

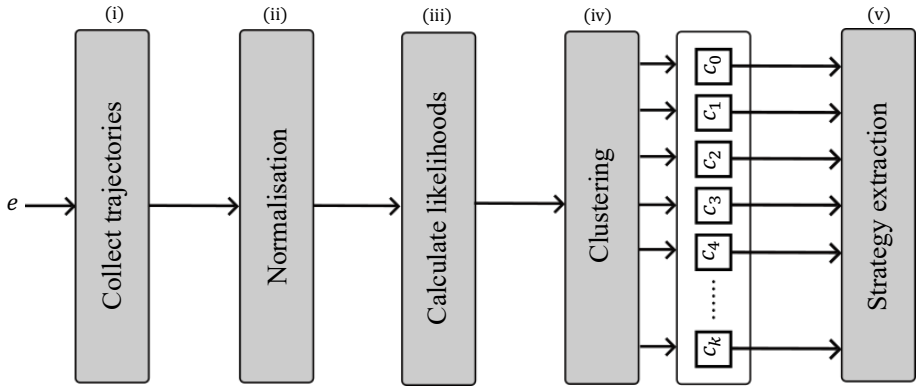

Figure 4: The stages of our strategy extraction approach for an event of interest e. Details are as follows: (i) generate positive and negative trajectories for e through simulation; (ii) using the positive and negative trajectory length distributions, normalise the negative trajectories; (iii) determine likelihood values for actions being part of a strategy; (iv) create clusters from the positive trajectories; and (v) for each cluster, perform strategy extraction to obtain candidate strategies through pairwise comparison.

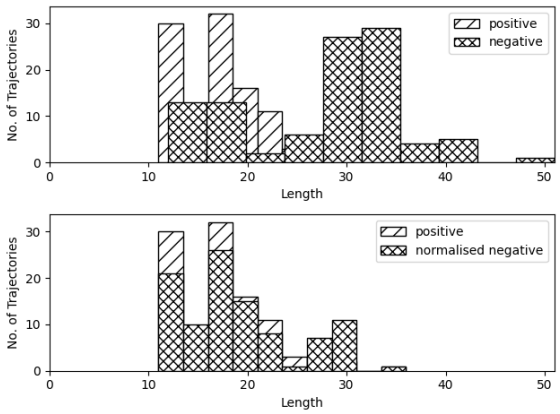

Figure 5: Before and after normalising the length distributions of the negative trajectories obtained from simulating the agent in a Pacman environment. Positive trajectories include the event "kill a ghost".

Example 3 (Pacman) *We expect a strategy for killing a ghost to include "collect power-up". This action is in* 100% *of trajectories in* τ_P*. We observe that episodes in which the agent does not kill a ghost involve long action sequences reflecting random exploration, item collection or ghost avoidance. As a result, we observe that "collect power-up" also appears in* 100% *of the negative trajectories. After normalisation, fewer of the negative trajectories contain this action.*

Calculating action likelihood

We have assumed that there is an equal likelihood for every action to appear in one or more strategies however realistically, this is very unlikely. By analysing groups of trajectories, we gain insights into the distribution of actions and recognise those which are more likely to be in a strategy. We derive likelihood values for each available action and use this information to disregard actions from the trajectories; simplifying them for subsequent calculations.

Algorithm 1 Find Strategies for an Event of Interest

Input: $\mathcal{C}$ (clusters), $\mathcal{L}$ (action likelihoods)
Output: Strategy set, $\mathcal{S}$

Let $\mathcal{S} = \emptyset$.
for $c \in \mathcal{C}$ **do**
  $t_c = ShortestTrajectory(c)$
  for $t_j \in c \setminus \{t_c\}$ **do**
    $result \leftarrow SmithWaterman(t_c, t_j, \mathcal{L})$
    $\mathcal{S} \leftarrow \mathcal{S} \cup \{result\}$
  end for
end for
return $\mathcal{S}$

To calculate the likelihood of an action, a, we compare its frequency of occurrence within τ_P and $\tau_N\prime$. If a occurs considerably more often in the positive samples than in the negative samples, its likelihood of appearing in the strategy is very high. Conversely, if it appears more in the negative than the positive, then commonsense reasoning tells us that a is unlikely to be part of the strategy. This is also true if the action appears approximately the same amount in both sets.

Likelihood is defined as the difference between occurrence frequencies of an action in each of τ_P and $\tau_N\prime$, denoted $f_{\tau_P}(a)$ and $f_{\tau_N\prime}(a)$ respectively. By default, all actions have a likelihood of 1. If an action only appears in the negative trajectories and not in the positive trajectories, its likelihood is set to 0. Equation 3 formalises the likelihood calculation for an action a.

$$l_a = \begin{cases} max(0, f_{\tau_P}(a) - f_{\tau_N\prime}(a)) & \text{if } a \text{ in } \tau_P \text{ and } \tau_N \\ 1 & \text{if } a \text{ not in } \tau_N \\ 0 & \text{if } a \text{ not in } \tau_P \end{cases} \tag{3}$$

Low-likelihood actions are then removed from τ_P using a predefined threshold. We denote these modified trajectories as $\tau_P\prime$ which is used for the remaining stages. A threshold of

Game $\mathcal{G}$	Event of Interest e	Strategies	Found(%)
Pacman	collect power-up	-	-
	kill a ghost	{*power-up, kill a ghost*}	100
	kill a ghost $\times 2$	{*power-up, kill a ghost, kill a ghost*}	100
	kill a ghost $\times 3$	-	-
Dungeon Crawler	collect key	{*collect gun, collect key*}	50
		{*collect sword, collect key*}	50
	kill a monster	{*collect gun, kill a monster*}	74
		{*collect sword, kill a monster*}	74
		{*collect key, kill a monster*}	28*
		{*collect gun, collect key, kill a monster*}	12*
	unlock door	{*collect key, unlock door*}	44
		{*collect key, kill a monster, unlock door*}	38*
		{*collect gun, collect key, kill a monster, unlock door*}	30*
		{*collect sword, collect key, kill a monster, unlock door*}	30*
		{*collect sword, unlock door*}	24
		{*collect sword, collect key, unlock door*}	22*
		{*collect gun, collect key, unlock door*}	20*
		{*collect gun, unlock door*}	18
		{*collect sword, kill a monster, unlock door*}	14
		{*collect gun, kill a monster, unlock door*}	6
		{*collect sword, collect gun, unlock door*}	4
		{*collect key, collect sword, collect gun, unlock door*}	2

Table 1: Strategies found for Pacman and Dungeon Crawler, and the frequency with which we find each strategy, across 50 runs of our approach. Across all runs, both τ_P and τ_N contain 100 trajectories. A '*' indicates a grouping of strategies that have the same actions, but with variations on the order of all but the final action. A '-' indicates no strategies were found for that event.

0.1 is used to obtain $\tau_P\prime$ across all experiments in this paper.

Example 4 (Pacman) *If the action "collect power-up" appears in* 100% *of positive trajectories and* 20% *in the negative trajectories; then its likelihood* $l_{power-up} = (100 - 20) \div 100 = 0.8$. *Computing likelihoods for the remaining actions, we obtain the following values: "move": 0, "collect dot": 0.1, "collect power-up":0.8, "kill a ghost": 1.*

Clustering

In this stage, we create clusters from the trajectories in $\tau_P\prime$ in order to group trajectories that are related. A new cluster is created for each action that occurs in a trajectory in $\tau_P\prime$, and all trajectories which contain this action are added to it. In this way, a trajectory may appear in more than one cluster. We denote the set of generated clusters, $\mathcal{C}$.

The aim of clustering is to ensure we uncover patterns when there are many different versions of trajectories that achieve the same e. For example, in the Dungeon Crawler environment for $e =$"kill a monster", $\tau_P\prime$ may include trajectories that contain one action a_1 and not another a_2 and vice versa. We can create two clusters; one for trajectories that contain a_1 and another for a_2. In the final stage, each cluster is considered separately to find all possible strategies.

Strategy extraction

In the final stage, the Smith-Waterman algorithm is used to perform multiple pairwise comparisons between trajectories in each cluster. Recall from Equation 2, the weighting $\mathcal{W}$. When comparing trajectory elements A_i and B_j, we use the weighting $\mathcal{W} = max(l_{A_i}, l_{B_j})$. Our choice to use likelihood in this way, rather than relying on rewards, is due to the large variance in the way rewards are assigned in different environments. If our method used rewards, the capability to extract appropriate strategies becomes dependent on the reward scheme of the game. We use likelihood values for strategy extraction to ensure the method is game-agnostic.

Algorithm 1 outlines our method for finding strategies for a given e. For each cluster, the shortest trajectory (i.e., with the least number of actions) is selected. A pairwise comparison between this trajectory, and all others in the cluster, is performed using the Smith-Waterman algorithm. The output of each pairwise comparison becomes a candidate strategy.

5 Experiments

We evaluate the performance of our proposed method in the Pacman and Dungeon Crawler games. For trajectory collection and determining events of interest, a PPO agent was trained using Masked PPO from *stable_baselines3*. Policies were trained on a maximum of 1 million timesteps, or until the agent exceeded a predefined threshold on average game reward during an evaluation phase occurring every 50 thousand timesteps. All experiments were performed on 3.2GHz Apple M1 with 16 GB RAM running Mac OS X 12.6.

To test the robustness of the extraction approach, we executed 50 runs for each environment under the same conditions and saved the resulting strategies, and total counts of how many times each strategy was found. The results for

Sample size	Action Likelihoods (average(min,max))	Run-time (s), average (min, max)	
		(i)	$(ii) - (v)$
10	"collect gun": 1, "collect sword": 0.94(0.4,1), "collect key": 0.91(0.3,1)	2.48 (1.30, 3.67)	0.0058 (0.0047, 0.0071)
50	"collect sword": 0.88(0.6,1), "collect gun": 0.81(0.56,1), "collect key": 0.06(0,0.16)	49.64 (9.62, 114.52)	0.032 (0.024, 0.039)
100	"collect sword": 78(0.6,1), "collect gun": 0.76(0.58,1), "collect key": 0.06(0,0.23)	129.01 (96.88 ,161.85)	0.052 (0.044, 0.058)
200	"collect gun": 70(0.67,0.78), "collect sword": 0.66(0.61,0.71), "collect key": 0.05(0,0.11)	107.23 (49.30, 175.24)	0.11 (0.092, 0.13)

Table 2: Average action likelihoods and method run times for $e =$ *"kill a monster"* in the larger *Dungeon Crawler* environment. Averages are calculated over 10 runs for each sample size. Run-times are reported in terms of the stages from Figure 4 - we record the average run-time for the trajectory collection stage and then the combined run-time for the remaining stages.

Event of Interest e	Predominant Strategies	Found (%)
collect key	{*collect gun, collect key*}	88
	{*collect sword, collect key*}	86
kill a monster	{*collect sword, kill a monster*}	98
	{*collect gun, kill a monster*}	94
unlock door	{*collect key, unlock door*}	34
	{*collect sword, unlock door*}	24

Table 3: Strategies found across 50 runs of the extraction approach for the larger *Dungeon Crawler* environment, using a sample size of 20 trajectories in τ_P and τ_N.

both games are shown in Table 1. The predominant strategies, those with the highest 'Found' percentage, are what we expected for each event of interest. In the Dungeon Crawler environment, we observed many possible strategies for the "unlock door" event of interest. In this environment, the agent performs best by both killing the monster and unlocking the door. Consequently, many positive trajectories for both events are likely to involve both of these outcomes.

To test the scalability of the approach, we applied it to a larger instance of *Dungeon Crawler*; an (11×41) grid with multiple (6) monsters and weapons (3 of each weapon type). In this environment, the trajectories are far more varied than in the smaller version, therefore we expect that the most appropriate strategy for events of interest will be harder to find. The results for this environment are shown in Table 3. We observed that strategies for rarer events are harder to find with the method, as it becomes more time-consuming to collect enough trajectories to adequately populate τ_P. We therefore reduced the number of trajectories collected for τ_P and τ_N, the sample size, from 100 to 20 trajectories.

Finally, we investigate the effect of the trajectory sample size on the strategies extracted. Table 3 shows that the expected strategies are still obtainable in the larger environment after reducing the sample size. We observe that the sample size has an influence on the likelihood values computed for each action. We expect that likelihood values for certain actions will become more accurate with an increased sample size. The results of this analysis are shown in Table 2. We also show the average time taken to run all the stages for each event of interest which increases as the sample size increases due to the trajectory collection stage.

Our results demonstrate that the proposed extraction approach is able to identify reasonable strategies for multiple events of interest. The performance of our approach is dependent on the ability to collect the same amount of positive and negative trajectories. In particular, in Table 3, for $e =$ *"unlock door"* the strategy {*collect key, unlock door*} is only found in 34% of runs. This result is due to the limitations of the trajectory collection stage where events of interest were eventually discarded if not enough positive trajectories could be collected within a specified time limit. The second key result is that the action likelihood values are influenced by the sample size of trajectories. This is most noticeable in Table 2, the likelihood of "collect key" to be part of a *"kill a monster"* strategy decreases dramatically as sample size increases from 10 to 50.

6 Conclusion

We have proposed an approach for the extraction of strategies from learned agent policies as a first step toward improving the ability of artificial agents to transfer and generalise their existing knowledge. We adapted the Smith-Waterman local alignment algorithm to find useful causal information from agent trajectories which we can use to form strategies. Our results when demonstrated on video game trajectories obtained using deep reinforcement learning, showcase the ability of this method to identify reasonable strategy candidates in different contexts. In future work, we will utilise the strategies obtained from this method and look at generalisation techniques to support lifelong learning across a wider class of tasks.

References

Asadi, M.; and Huber, M. 2007. Effective Control Knowledge Transfer through Learning Skill and Representation Hierarchies. In *IJCAI'07: Proceedings of the 20th International Joint Conference on Artifical intelligence*, 2054–2059.

Bosc, G.; Kaytoue, M.; Raïssi, C.; and Boulicaut, J. 2013. Strategic Pattern Discovery in RTS-games for E-Sport with Sequential Pattern Mining. *MLSA@ PKDD/ECML*, 11–20.

Brockman, G.; Cheung, V.; Pettersson, L.; Schneider, J.; Schulman, J.; Tang, J.; and Zaremba, W. 2016. OpenAI Gym. arXiv:1606.01540.

Chen, Z.; El Nasr, M. S.; Canossa, A.; Badler, J.; Tignor, S.; and Colvin, R. 2015. Modeling Individual Differences through Frequent Pattern Mining on Role-Playing Game Actions. In *11th Artificial Intelligence and Interactive Digital Entertainment Conference*. AAAI Press.

Chen, Z.; and Liu, B. 2018. *Lifelong Machine Learning: Second Edition*. Morgan & Claypool Publishers LLC.

Collins, A. G. E.; and Frank, M. J. 2013. Cognitive control over learning: creating, clustering, and generalizing task-set structure. *Psychological Review*, 120(1): 190–229.

Een, N.; Legg, A.; Narodytska, N.; and Ryzhyk, L. 2015. SAT-Based Strategy Extraction in Reachability Games. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*. AAAI Press.

Low-Kam, C.; Raïssi, C.; Kaytoue, M.; and Pei, J. 2013. Mining Statistically Significant Sequential Patterns. In *2013 IEEE 13th International Conference on Data Mining*, 488–497.

Parisi, G. I.; Kemker, R.; Part, J. L.; Kanan, C.; and Wermter, S. 2019. Continual lifelong learning with neural networks: A review. *Neural Networks*, 113: 54–71.

Parsons, S.; and Wooldridge, M. 2002. Game Theory and Decision Theory in Multi-Agent Systems. *Autonomous Agents and Multi-Agent Systems*, 5(3): 243–254.

Paul, S.; van Baar, J.; and Roy-Chowdhury, A. 2019. Learning from trajectories via subgoal discovery. *Advances in Neural Information Processing Systems*.

Schulman, J.; Wolski, F.; Dhariwal, P.; Radford, A.; and Klimov, O. 2017. Proximal Policy Optimization Algorithms. arXiv:1707.06347.

Smith, T. F.; and Waterman, M. S. 1981. Identification of Common Molecular Subsequences. *Journal of Molecular Biology*, 147(1): 195–197.

Srovnal, V.; Horák, B.; Bernatík, R.; and Snášel, V. 2004. Strategy Extraction for Mobile Embedded Control Systems Apply the Multi-agent Technology. In *Computational Science - ICCS 2004*, 631–637. Springer Berlin Heidelberg.

Weiss, K.; Khoshgoftaar, T. M.; and Wang, D. 2016. A survey of transfer learning. *Journal of Big Data*, 3(1): 1–40.

Zhu, Z.; Lin, K.; Jain, A. K.; and Zhou, J. 2020. Transfer Learning in Deep Reinforcement Learning: A Survey. arXiv:2009.07888.