Using Deep Reinforcement Learning to Build Intelligent Tutoring Systems

Ciprian Paduraru, Miruna Paduraru and Stefan Iordache
University of Bucharest, Romania

Keywords: Tutorial System, Reinforcement Learning, Actor-Critic, TD3, Games.

Abstract: This work proposes a novel method for building agents that can teach human users actions in various applications, considering both continuous and discrete input/output spaces and the multi-modal behaviors and learning curves of humans. While our method is presented and evaluated through a video game, it can be adapted to many other kinds of applications. Our method has two main actors: a teacher and a student. The teacher is first trained using reinforcement learning techniques to approach the ideal output in the target application, while still keeping the multi-modality aspects of human minds. The suggestions are provided online, at application runtime, using texts, images, arrows, etc. An intelligent tutoring system proposing actions to students considering a limited budget of attempts is built using Actor-Critic techniques. Thus, the method ensures that the suggested actions are provided only when needed and are not annoying for the student. Our evaluation is using a 3D video game, which captures all the proposed requirements. The results show that our method improves the teacher agents over the state-of-the-art methods, has a beneficial impact over human agents, and is suitable for real-time computations, without significant resources used.

1 INTRODUCTION

Our motivation to build AI agents (teachers) that can advise human users (students) stems from our experience in the gaming industry. What we observed in internal reports by analyzing user data is that many of them install a game demo or trial version on their device, but uninstall it after a short time before it brings revenue to the developer. When we went deeper to understand the behavior and reasons of the users, we discovered that many of the users stop playing the game because they do not understand the game dynamics or do not know how to play against other users online. The next attempt by game developers to fix this problem is usually to create some kind of text tutorial system that can help users understand the mechanics and improve their experience while playing. However, a handwritten tutorial proves to be limited as it cannot capture the multimodal characteristics, usage style, and learning curves of different users (Alameda-Pineda et al., 2018). Moreover, trying to implement such systems is usually very costly in terms of production costs.

The purpose of this work is to explore how we can build teacher agents that can advise human students to understand an application by giving the right suggestions to improve their behavior within the application, and at the same time without being annoying to the user. We understand that improving their behavior directly leads to a richer, more rewarding experience on the user side during the application. In our specific use case, we automate tutorial systems in game development processes. In terms of the methods used, in short, the online system uses a novel tutorial system that is automatically trained using reinforcement learning (RL) to provide live advice during the application runtime.

We also argue that the proposed methods can be adapted to other simulation environments and applications, not just computer games. Since we evaluated the method on an open-source 3D game, (Paduraru and Paduraru, 2019), we assume that it would also be suitable for many other types of applications with an extensive input/output space and logical complexity of behaviour, such as 3D computer-aided design, 3D modeling software, video or photo editors, etc. The reasons for this consideration are the following:

- The state of the game at any time is large in both dimension and logical complexity.
- The input and output spaces of the game are composed of both continuous and discrete types.

Paduraru, C., Paduraru, M. and Iordache, S.
Using Deep Reinforcement Learning to Build Intelligent Tutoring Systems.
DOI: 10.5220/0011267400003266
In *Proceedings of the 17th International Conference on Software Technologies (ICSOFT 2022)*, pages 288-298
ISBN: 978-989-758-588-3; ISSN: 2184-2833

The main contributions of this paper over existing work can be summarized as follows:

1. To the authors' knowledge, this is the first work that proposes a method for building a tutoring system using Deep Reinforcement Learning techniques. The previous works closest to ours, (Zhan et al., 2014), (Wang and Taylor, 2017), use a combination of reinforcement learning and heuristics to decide at what time to advise the user. They restrict the set of actions to discrete spaces, consider bounded environmental states, and decisions are only made instantaneously. Instead, we use modern deep learning techniques for several purposes: (a) building the architecture of the teacher model, (b) encoding the environmental states as embeddings, (c) removing the fixed heuristic thresholds and introducing them as learnable parameters in the model. As a result, our method is able to retain history information and make contextual decisions, rather than considering only instantaneous, and large environmental states with both discrete and continuous inputs/outputs.
2. Previous work does not focus on qualitative evaluation of trained policies (i.e., on real human users) and mostly evaluates the results quantitatively. Our work proposes a novel method to create an extended environment for observing student performance in accordance with the training goal, and adapting the suggestions based on this feedback. We also create a reward system within our RL methods that incorporates *human in the loop* to train teacher policies that are not disruptive to future students.
3. In the context of the gaming industry, to the authors' knowledge, this is the first work to propose a concrete automatic method to replace scripted tutorial systems in video game play. Our approach has the potential to not only reduce the cost associated with creating scripted systems for training human users, but also to dynamically adapt to the multimodality and high-dimensional state context of user behavior.

The paper is organized as follows. Section 2 gives an overview of some recent work in this area that had an important impact on our implementation decisions. The architecture and algorithms used are described in section 3. The evaluation is done in section 4, while a conclusion and some ideas for future work are given in the last section.

The framework built and presented in this paper is called *AHRec* (**A**gent to **H**uman **Rec**ommender system). Due to space constraints and double-blind review requirements, more details on the visual output database structure, pseudocode of the training algorithms and parameters used, discussions and limitations, along with a demonstration video can be found in our **supplementary material** available here: https://www.dropbox.com/sh/edc0hhlbdjb9zop/AADYBHGJV8mJP6ezca-lontza?dl=0. (reviewer note: these will be added and publicly available on Github outside the main paper after peer review). We also include an open-source implementation of our Teacher model with Tensorflow 2 backend and our test game environment at the anonymous Github account https://github.com/AGAPIA/BTreeGeneticFramework.

2 RELATED WORKS

One of the papers that addresses the same problem as ours is (Zhan et al., 2014), where the authors investigate different mechanisms to allow an AI trained with Q-learning to train human players on a classic Pac-Man video game. We follow their methods for limiting the number of recommendations per session within a given budget. The paper proposes a recommendation policy in the form of $\pi_d(S, B, I(\cdot), \rho(\cdot)) \rightarrow \{\pi_t, \varnothing\}$, where S captures the state of the student, B is the state of the advice budget, I is a function that evaluates the importance of the current state, while ρ specifies the rate at which advice is displayed. When a decision is made, similar to our work, the best action to the teacher's knowledge is given (π_t). The importance function is computed using a formula similar to that used in apprentice learning (Clouse and Utgoff, 1996), $I(s) = \max_a Q(s,a) - \min_a Q(s,a)$. The algorithms used by the authors then use different thresholds based on budget B, outcome I, and ρ to schedule the generation of teacher advice. There are five main improvements that our work addresses over this work: (a) Our method uses a full deep neural network to represent the teacher's state, using embedding layers. Previous work uses function approximation based on heuristics to represent the state. (b) Our method also considers an augmented state of the teacher that tracks the progress of the student and adapts to their needs over time, (c) we use a full RL method that uses Deep Learning to learn not only the importance of actions, the advice policy, but also the thresholds, rather than using simple heuristics that do not adapt or learn from existing real data, (d) we consider the order of actions in the decisions by using a recurrent neural network architecture; previous work only considers the current state when a decision is made, (e) Our actions can be proposed at both a low-level and high-level granularity to make them more explainable to human users.

In (Wang and Taylor, 2017), the authors study the problem of training an AI agent using the knowledge of a teacher agent (which could be a human or another AI agent) as a prior. In our work, we apply the inverse of this method by training a teacher agent using human expert demonstrations, which in turn can train new human students. The idea behind their work is to take a pre-trained teacher model that is used as a prior to bootstrap the values of states and action pairs at the beginning of the training of the student model. They prove that this initial prior can speed up the training process and make the student outperform the teacher faster. The *CHAT* method uses a confidence measure computed by one of the following three methods to detect the uncertainty in the teacher's recommendation: (a) *GPHAT* (Gaussian processes with variances and medians for actions learned from data), (b) a two-layer neural network that computes the confidence of actions in different states using a softmax regressor (the network is trained with user-supplied inputs), (c) *DecisionTreeHAT*, which uses a decision tree classifier where leaf nodes assign costs. The student considers the advice sent by the teacher with probability Φ. This value decays after each episode, being close to 1 at the beginning of the training process and decreasing to almost 0 when the agent can start acting independently to give it a chance to outperform the teacher. If the advice sent by the teacher to the student has a confidence value greater than a fixed threshold, then the student chooses the sent advice as the next action. If it does not, he performs his own guess according to his running training policy. The student is also allowed to explore with the classical decision factor ε (exploration vs. exploitation). Thus, there is a ε probability for exploration, Φ for transferring the action from the teacher to the student, and $1-\varepsilon-\Phi$ for acting according to its own current policy. The same DQN methods similar to (Zhan et al., 2014) are used as RL algorithms.

Even though the purpose was to train AI agents and not humans, their method can be transferred to improve the work in (Zhan et al., 2014) as we identified in this paper, and also be relevant for training human agents. In short, we tried to extend the work in (Zhan et al., 2014) by multiplying the importance of a state, $I(s)$, with the confidence value computed by the *CHAT* methods. The intuition behind this is to lower the importance of decisions when there is a large uncertainty behind the recommended actions. Their solution is also important to us because we reused the idea of a confidence value from their work. However, instead of setting a confidence threshold, when taking decisions, we instead set this as a learnable parameter within the models.

There are also several papers that approach the same core problem as we do, but for the educational field. In (Martin and Arroyo, 2004) and its sequel (Sarma and Ravindran, 2007), the authors build a graph between the skill set and the available hints to cluster and train simple policies for asking questions. In (Beck et al., 2000), (Malpani et al., 2011), and (Ausin et al., 2019), policies are trained to ask a set of questions that have predefined difficulties. When performing actions, the policy correlates correctness and response time from previous responses with the available set of questions to ask next. The methods used include recurrent neural networks (LSTMs) and Gaussian Processes, to infer immediate rewards from delayed rewards as a mechanism to improve policy outcomes. In (Peltola et al., 2019), a Bayesian multi-armed bandit method is used to build the model of a teacher. While the ideas from the educational field are valuable and provide insights for our methods, we believe that these approaches are not currently suitable for real-time applications such as computer games due to their limitation to the state and action space. Both are discrete and limited in the number of possible decisions. In addition, their training mechanisms consider human interaction in a way that is not useful for our purposes. Our proposal incorporates user feedback into episodes in an optional manner, rather than forcing interaction at each time step.

3 PROPOSED METHODS

3.1 Overview

There are two main actors used in our method: *Teacher* and *Student*. The *Teacher* agent can give suggestions at runtime, depending on its observations over *Student*'s behavior and the state of the environment in which the *Student* operates. The intuition is to let the *Teacher* agent observe the sequence of actions and states of the *Student* and then act in a way that both balances the utility of the suggestions and keeps the user in an entertaining mode without showing them too often or when not necessary.

The high-level flow for obtaining the policies that decide when and what suggestions to give is shown in Fig. 1. There are three policies that are trained for a *Teacher*.

1. A suggestion making policy π_T that can be sampled ($a_t \sim \pi_T$) to query *what is the action the Teacher would do in the Student's application state (environment state) S at time t*. In the first step, a *Teacher* suggestion policy is trained within the application using a policy-based RL technique

where rewards are learned from Generative Adversarial Imitation Learning (GAIL) (Ho and Ermon, 2016) by observing how real people (considering only the best available demonstrations) use the application. The methods used restrict the model to human comprehensible teaching methods (i.e., space of possible actions). The assumption is that the agent trained with the above settings will generally make the ideal decisions and still be able to handle human multimodal behavior by using a learned stochastic policy that maps the probability of actions from the given states. This is done in Step 1 in Fig. 1.

2. A policy that can be sampled to find out whether the *Teacher* should send or not send a suggestion at a given time step t: $a_t \sim \pi_{Advice_{v1}}$. This is an intermediate policy, as shown in Step 2, Fig. 1. It takes into account observations about how the trained *Student* performs (detailed in the next subsections), along with the context of the previously sent suggestions and feedback. As detailed in Section 3.2, in this step the *Student* does not explore the environment at all, but takes steps using only his current knowledge and the suggestions sent by the *Teacher*.
3. The final version of the suggestion making policy, π_{Advice}, is obtained in Step 3 by fine-tuning the parameters using human-sensitive feedback that lies behind the *Student* agents and provides qualitative information about the sequence of suggestions received.

Note that π_T is trained statically using demonstration data provided by experts, and remains fixed during the training of π_{Advice}. The second does not require human supervision at all, but human feedback can optionally be included if the user responds to various feedback questions while using the tutoring system. The π_{Advice} policy is trained with a class of Actor-Critic methods, specifically $TD3$ (Fujimoto et al., 2018). Given an embedding of the teacher's observations as a state, the policy outputs the probability of showing or choosing not to show a suggestion to *Student*, which is further sampled from π_T. Episodes consist of trajectories of the agent *Teacher* suggesting or not suggesting things at certain time steps. Each episode lasts until either a threshold for the number of suggestions is reached, MST, or a game session ends. The rewards, which consist of two components - the agent's performance relative to some trainable goals and any human sensitive feedback returned - are intuitively used to improve the policy's rewards over time by handling the correct moments at which suggestions should be made. Higher rewards mean higher performance for the *Student* agent's learning curve.

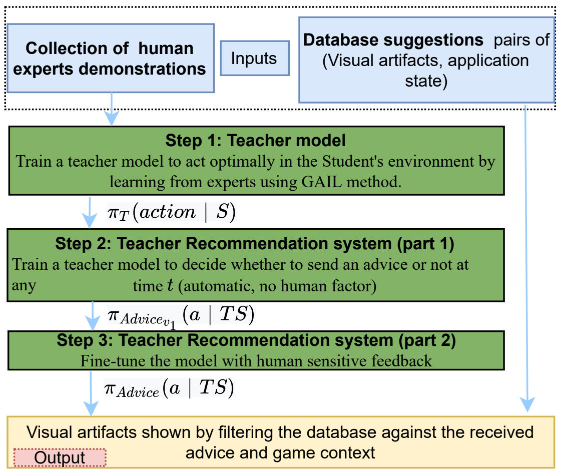
Collection of human experts demonstrations
Inputs
Database suggestions pairs of (Visual artifacts, application state)
Step 1: Teacher model
Train a teacher model to act optimally in the Student's environment by learning from experts using GAIL method.
$\pi_T(action \mid S)$
Step 2: Teacher Recommendation system (part 1)
Train a teacher model to decide whether to send an advice or not at any time t (automatic, no human factor)
$\pi_{Advice_{v1}}(a \mid TS)$
Step 3: Teacher Recommendation system (part 2)
Fine-tune the model with human sensitive feedback
$\pi_{Advice}(a \mid TS)$
Visual artifacts shown by filtering the database against the received advice and game context
Output

Figure 1: The flow to obtain the Teacher's policies (π_T in Step 1, then π_{Advice} from Step 2 and 3), and finally the visual output for the human user.

Negative feedback from human users (e.g., disruptive or incorrectly displayed hints) penalizes the *Teacher* agent's reward.

The output of the tutoring system, i.e., the suggestions of a trained *Teacher* to the *Student*, are conveyed in visual form. The transformation works as follows. Each visual suggestion contains a set of artifacts such as text, images, arrows, and the relative positions for each. Each suggestion is also accompanied by a description of the game state that explains, from the perspective of a human user, why that particular suggestion is important in the context of the game. For example, the *Teacher* might show a text and image to pick up a *healthbox* because the current user's life meter is low. A database of visual suggestions is then recorded along with their game state description. Then, at runtime, when the *Teacher* decides to send a suggestion, the database is filtered and a subset of compatible suggestions, CR, is kept for future evaluation. A similarity measure comparing the current game state with the suggestions in the CR set (absolute normalized difference between features such as health status, relative distances to enemies, and upgrade boxes) is used to decide which one to display. The step of building the database of visuals and compatible game states is currently done manually to ensure the plausibility of the displayed suggestions, but in the future we also consider automating this step.

In the rest of this section we describe in detail the implementation of this system.

3.2 Environment Setup

The *Extended Environment* (ES) component is divided into two subcomponents (Fig. 2):

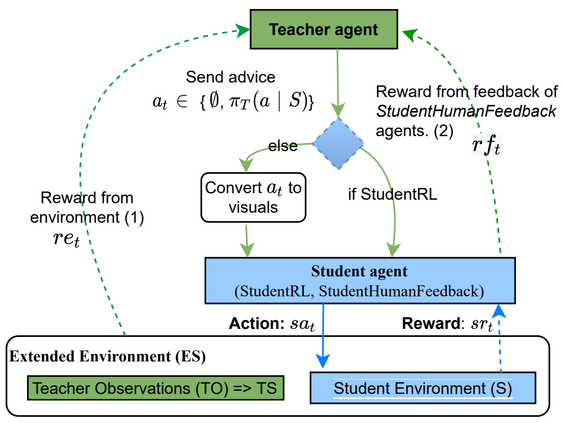
Teacher agent
Send advice
$a_t \in \{\emptyset, \pi_T(a \mid S)\}$
Reward from feedback of StudentHumanFeedback agents. (2)
rf_t
else
Convert a_t to visuals
if StudentRL
Reward from environment (1)
re_t
Student agent (StudentRL, StudentHumanFeedback)
Action: sa_t
Reward: sr_t
Extended Environment (ES)
Teacher Observations (TO) => TS
Student Environment (S)

Figure 2: The training process for obtaining *Teacher*'s suggestion policy π_{Advice} shown in Step 2 and 3 (Fig. 1) and explained in Section 3.

- **Student Environment** (S). Represents the part of the environment that the *Student* interacts with. In the general use case, this is the end-user application. State at time step t is denoted by S_t.
- **Teacher Observations** (TO_t). This is a backend component in the application that is not visible to the *Student* and collects observations about how the *Student* behaves over time in the application. In our case, since we are evaluating a game, we are interested in observing concrete things such as: did the player make progress in looking at the map, did he manage to hide when necessary, how many points did he score in a given situation. For example, in a word processing application, the system might monitor whether the user succeeds in finding the fonts, styles, and other basic components. We assume that at any given time, the application can make an assessment of how well the user is succeeding in understanding the application based on these observations.

The Student interacts with its own part of the environment (the blue colored boxes and arrows in Fig. 2) at a time t by performing an action sa_t and getting back a reward sr_t and a new state sa_{t+1}. This interaction is usually performed in every frame of the application.

The *Student* can be one of three types:

- StudentRL: This agent is trained using the same method as the *Teacher*, i.e. GAIL, and the same rewards. The agent itself has a locally learned policy that is only available during the training session with *Teacher*: $\pi_{S_t}(a|S)$. The action chosen at each frame is then read from this policy.
- StudentHumanFeedback: this is a human who takes the place of a *Student* and is able to give sensitive feedback on the suggestions received (e.g., on topics such as how disruptive he/she is because the *Teacher* makes too many suggestions, or focuses too much on the same category of actions or behaviors, etc.). This can also be illustrated in Fig. 2. Note that *StudentRL* cannot provide such sensitive feedback.
- StudentHuman: this agent does not provide any feedback or reward in training, and it is only used for final evaluation of our methods (Fig. 3).

The Teacher interacts with the other part of the environment (the green-colored boxes and arrows in Fig. 2) by deciding whether or not to send suggestions to *Student* at specified time steps. Note that these time steps are different from those of *Student*, since it has to analyze the situation over a longer period of time before making a decision. From another point of view, it does not make sense to send suggestions too often, as this could become disruptive for the *Student*. The *Teacher*'s proposal system state TS_t at a time step t is a concatenation of several features composed of the constructions explained and motivated below, as shown in Eq. 1. This is formed from the observations monitored by the *Teacher* (the TO_t values) and some other features:

$$TS_t = (Embedding(S_t), TO_t, Budget_t, HistShown_t) \tag{1}$$

- $Embedding(S_t)$: the embedding of multiple frames of the environment state where the *Student* agent is acting on between time points $[t-L+1, t]$. Details on how to obtain this are given in Section 3.4.
- $Budget$: an array indexed by the type of actions already proposed and how many of each were shown. It is important to understand that humans have little attention and patience compared to robots. With this in mind, our method tries to limit showing frequent but imprecise suggestions as much as possible, as opposed to showing fewer but precise suggestions.
- $HistShown$: an array of size L containing how many seconds ago the system showed the *Student* the last L suggestions. This is important because a human serving as *Student* can provide feedback when the suggestions become disruptive. So the purpose of this function is to automatically learn the right times to show the suggestions.

The training process (detailed in 3.3) models a policy that gives the probability of showing or not showing a suggestion at a given time step, given the state of the *Teacher*: $\pi_{Advice}(a_t|TS_t)$, where $a_t \in$

$\{\emptyset, \pi_T(a|S)\}$. The suggestion given always corresponds to the *Teacher*'s belief about the ideal action to do in the given *Student*'s environment state (S), i.e., $\pi_T(a|S)$. There are two types of rewards observed by the *Teacher*:

1. rewards that come from observing how the *Student* agent behaves in the environment according to the application's metrics (included in TO_t), re_t. This is useful for training the part of the network that models the importance of the current environment state S and thus suggests actions when needed.
2. rewards that come from the *StudentHumanFeedback* agents, rf_t, based on sensitive feedback from human users. This is in fact a kind of Active Learning for the *Teacher* agent (Rubens et al., 2016).

To separate the application specifics from the algorithm, we implemented several hooks in the proposed framework that allow users to make their own customizations for things like $Teacher's$ observations TO_t, environmental state S_t, and rewards. In the case of our evaluation application, and with the specific example of observations TO_t given at the beginning of this section, a possible example set of concrete rewards (also used in our evaluation) for re_t could be: $+1$ for each new map area covered, $+10$ for each enemy destroyed, $+20$ for each upgrade box used, and $+40$ for each successful taking cover or running against a stronger enemy. For rf_t, possible values could be -100 for each negative human feedback and $+50$ for a positive feedback. Also, in our evaluation, the *MST* constant was set to a maximum of 20 suggestions per episode (this can be adjusted by the end user; in our evaluation, we set this value so that the learning process would be efficient; a longer setting might result in too sparse rewards, as it would be difficult to learn from states and action pairs that lead to significant rewards).

3.3 Teacher Recommendation Model's Architecture, Training and Evaluation

The *Teacher*'s suggestion making policy π_{Advice} uses an $Actor-Critic$ class architecture (Konda and Tsitsiklis, 2000) in which the *Actor* decides the probability of sending or not suggestions to the *Student* based on its state TS_t at the evaluated time t. The *Critic* then decides how valuable the action chosen in the given state is. If the value sampled from the policy π_{Advice} is higher than a trainable parameter of the Bernoulli distribution threshold TH, then the *Teacher* chooses a suggestion, i.e., a sampled action from $\pi_T(a|S)$, based on its best knowledge of what to do in the current state of the *Student*'s environment. After a decision is made whether not to send a suggestion ($a_t = \emptyset$) or to send one ($a_t \sim \pi_T$), the progress of the *Student*'s agent is observed by the application for a period of time with respect to the desired metrics. Based on these observations, the system can compute the rewards returned by the *TeacherObservation* component of the *ExtendedEnvironment*. The intuition for optimizing policy π_{Advice} over time (according to the policy gradient class of algorithms) is the following: If the decision made (a_t) in the given state (TS_t) is good, its probability is increased by backpropagation, if not, it is decreased.

Since the environment outputs a continuous action space, after several benchmark studies with different algorithms, we concluded that the best results in our case were obtained by using a particular type of actor-critic algorithm: 'Twin Delayed Deep Deterministic' ($TD3$) policy gradient (Fujimoto et al., 2018). The decision was made after a comparison with other policy gradient class algorithms such as TRPO (Schulman et al., 2017a), PPO (Schulman et al., 2017b) and SAC (Haarnoja et al., 2018), with a *StudentRL* instance on the same benchmarks used in Section 4). Methodologically, the use of $TD3$ seems more appropriate in our use case, as the $Q-value$ could easily be overestimated with other methods, while the delay in policy updates helped our training in two ways: it made the training process more stable by reducing the error per update and improved the overall runtime performance. For training, we use a mini-batch of N transitions, using the importance sampling strategy with prioritized experience replay (Schaul et al., 2016).

The architecture of the full *Teacher* model is shown in Fig. 4. The runtime evaluation (inference) process is shown in Fig. 3. The interested user can refer to our supplementary material for the implementation details of the training and inference pseudocode.

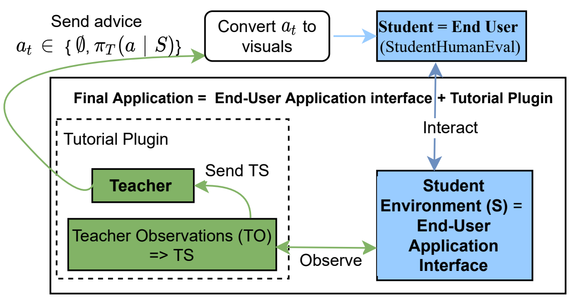
Send advice
$a_t \in \{\emptyset, \pi_T(a \mid S)\}$
Convert a_t to visuals
Student = End User (StudentHumanEval)
Final Application = End-User Application interface + Tutorial Plugin
Interact
Tutorial Plugin
Teacher
Send TS
Teacher Observations (TO) => TS
Observe
Student Environment (S) = End-User Application Interface

Figure 3: The components used at runtime of the application in evaluation mode, as described in section 3.3.

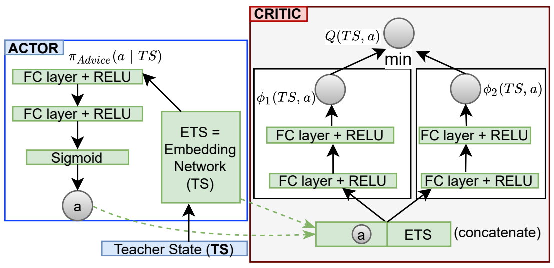

Figure 4: The Actor-Critic class architecture used by our framework using the TD3 algorithm, as detailed in Section 3.3.

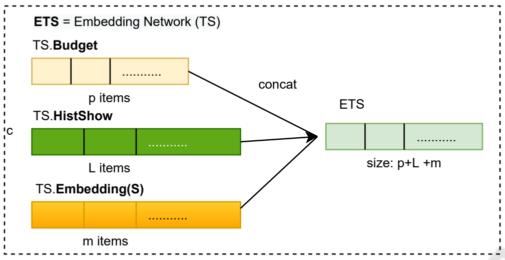

Figure 5: Showing how the *Teacher*'s state TS is used to obtain the embedding ETS by simply concatenating its input features.

3.4 States Embedding

The information obtained from the state of *Teacher* (TS) could be used raw, as it is. But according to the literature, in many situations it is better to have a neural network structure that learns the structure of the input data, converts it to a different dimensional space, reduces its dimension or upscales it (Ren et al., 2021), (Agarwal et al., 2021). This is the motivation for the implementation of the state embedding, ETS layer, which is included in the architecture of the Actor model in Fig. 4. Its value is obtained by concatenating several features from Eq. 1, as shown in Fig. 5. In our case, it plays an important role in both the model architecture and the results.

The low-level embedding of the *Student* environment state S (Fig. 6) involves a concatenation of 3 different entities: (1) an embedding of the nearest static objects around the user in the environment - this information is mainly used for cover point detection (2) the information from the last L frames about the user - positions, orientations and state (health, ammo, current upgrades) (3) similar information as for the main user, but for the user's nearest K enemies on the map. Each of these three embeddings use an internal architecture with two fully connected layers. Of interest for our particular use case is the attention layer (Vaswani et al., 2017) used. The motivation of its use here is to teach a model to weight the opponents in the game differently depending on their information embedded over a sequence of L frames analyzed.

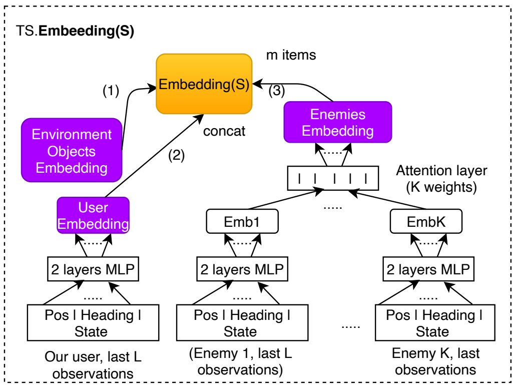

Figure 6: The hierarchical embedding of state S architecture. Note that this time, as detailed in the text, the embedding has trainable parameters.

4 EVALUATION

The first part of this section presents the setup used for training and evaluating the framework. It describes the environment under test and the metrics used. The second part analyzes the response to a few research questions about the efficiency and feasibility of the framework in practice. We adapt the Kirkpatrick's four levels for evaluating training (Newstrom, 2006) by conducting feedback collection from the evaluated users. This collection process is done: (a) automatic, through metrics evaluation, and (b) manual, through user interviews by text.

4.1 Application under Test

The function purpose of our methods was to be able to specify the suggestions at a higher level of granularity so that they are understandable to a human user, in both discrete and continuous action spaces, and at different levels of granularity, e.g., move to a specific location, take cover, shoot a specific unit, take a resource box, etc. We built our system on a 3D open-source game (Paduraru and Paduraru, 2019) using Unity Engine. The game has a customizable (NU) number of AI agents (tanks) and humans fighting on their own, with limited resources (life, health, ammo). The agents can get powerups (shield, better weapons), or replenish their resources. We used $NU = 10$ agents, with each episode lasting until there is only one tank left on the map, which generally takes ~ 2 minutes.

The database of visual suggestions VS is composed of 118 suggestions. The effort time needed to represent it took around ~ 58 hours. The manual process involved mostly the creation of texts and the decision of the relative positions and the orientations of the visual artifacts relative to the units.

4.2 Qualitative Metrics Used

After observing the problems new users had in understanding our application, we selected some metrics that could be automatically profiled and corrected using our tutoring system.
M1: What percentage of the map does a user manage to iterate on average during 10 episodes? (controls understanding).
M2: In a sequence of 100 episodes, how many times has the user managed to upgrade their entity (a tank game instance) by fetching crates from the map? (detecting upgrade boxes).
M3: How much time (on average) did the user survive in total during a sequence of 100 episodes? (ability to defend).
M4: How many total enemies has the user eliminated during a sequence of 100 episodes? (attack skill).

4.3 The Data Gathering and Training Process

To obtain the policy used for evaluating the *Student*'s environment state and making suggestions, $\pi_T(a_t|S)$, we used 4 selected people from an internal Quality Assurance (QA) department of a game company who knew the application very well. They recorded several uses of the application in an interval of 5 days, each with 8 hours of sessions. Their recordings were used to train the policy as described in the previous section with GAIL method (Ho and Ermon, 2016). The training last about 72 hours on a cluster of 4 Nvidia GTX 2080 Ti graphics cards.

Using the above trained policy as fixed and input for the next step, the training of the advising policy, $\pi_{Advice}(a_t|TS)$ was done using two steps: *Step (1):* The first policy version, $\pi_{Advice_{v1}}(a_t|TS)$, is obtained using a *StudentRL* by training for 120 hours , *Step (2):* Starting with the $\pi_{Advice_{v1}}(a_t|TS)$, we used our internal Quality Assurance department with a different set of 12 people, which did not use the application before, to play 4 sessions each of 8 hours to fine-tune the suggestions giving the final policy π_{Advice}.

4.4 Research Questions Evaluation

Having the policies trained as described in Section 4.3, and a set of metrics defined in Section 4.2, we are interested to evaluate three things:
R1: How well does *Teacher* help the users in understanding the application according to the goals set? This also tests the *adaptability* of the suggestions depending on the user's needs.
R2: From the users' perspective, how disruptive or useless are the suggestions sent?
R3: How costly in terms of resources is to run the advice system inside an application? Is it suitable for real-time applications ?

To respond to these questions, we benefit from doing playtests with 129 students in an academic partnership with University of Bucharest. We compared five different methods of training users in our game environment (note: all methods used the same visual resources available for training individuals inside our environment. We split the 129 students into 8 groups randomly divided as equal as possible and given out one the following methods:
Manual: A classic manual for the game, showing text and images describing controls, mechanisms, suggestions what to do in different situations. This is shown before entering the game sessions, but also available during play time.
Tutorial: A scripted tutorial before entering game sessions to put the users in a few scenarios and show them how to move through the map, attack, defend or use the upgrade boxes. After this tutorial that lasts $\sim$ 20 minutes, the player was let alone to play the game sessions with no other help.
AHRec: Using the trained *Teacher* to give adaptive suggestions using our proposed method.
PacMan: The method used by (Zhan et al., 2014).
PacHAT: The *CHAT* method from (Wang and Taylor, 2017) combined with (Zhan et al., 2014), according to our idea described in Section 2 to adapt it in training human users.
R1 Evaluation. For this evaluation, we profiled the results of metrics M1-M4 (Section 4.2) for all users within each of the groups. The results obtained were averaged over $\sim$ 80 game sessions played in 4 hours, with samples taken at each $30 minutes$ (the X-axis of the graphs below). Results are depicted in Fig. 7, 8, 9, 10 (Note that suffix *fa* stands for function approximation method, *emb* is for embedding method; *AHRec* by default uses embedding).

It is worth noting that all three methods based on AI techniques succeed in accelerating the training progress of the *Student* compared to the two classical methods. However, when using the proposed *AHRec* framework, we notice an improvement in the profiled metrics. We attribute these performance gains over the other two methods mainly to the contributions that we add and discuss in Section 1. Another thing to note is that creating the embedding using Deep Learning techniques plays an important role in the qualitative results for mapping the scene context via function approximation methods. We would expect this to be especially the case when mapping large environments

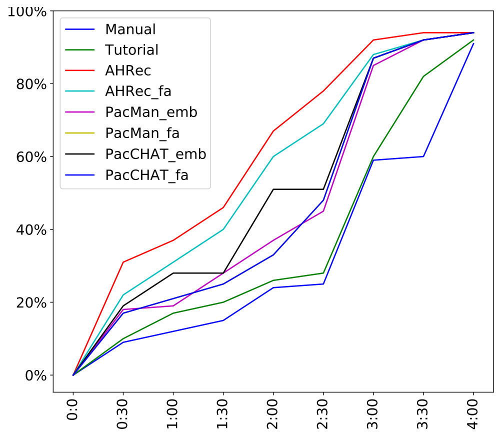

Figure 7: M1 evaluation. The Y-axis corresponds to the percent of map covered. All methods converge to fully coverage of the environment space (users understand the controls), with an advantage for the *AHRec* method which corrects the gaps in understand the controls in the game quicker.

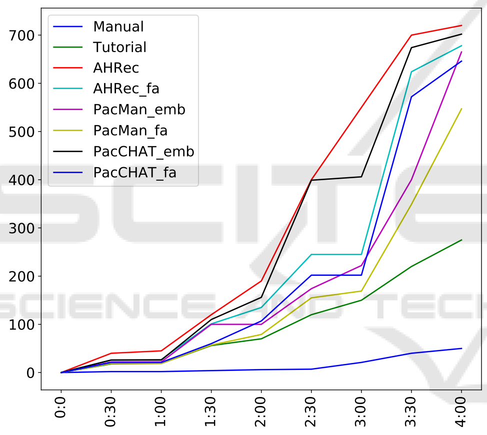

Figure 8: M2 evaluation. The Y-axis corresponds to the average number of used upgrade boxes. By using *AHRec*, the user learns quicker to use the upgrade boxes with the visual aids suggested at runtime that shows the utility of each in the right context (e.g. hint to take a health box when user's health is low).

such as the game in cause.

R2 Evaluation. In order to go more in depth in the qualitative evaluation, we tried to assess the sensitive feedback of the users when using the *AHRec* method. We conducted the experiments again with two groups of volunteers, each with 30 subjects, who received a game instance using the *AHRec* method. However, the first group used the intermediate $\pi_{Advice_{v1}}$ policy, while the second group used the final π_{Advice} policy. The purpose of this split is to understand the importance of fine-tuning the final policy with human feedback (*StudentHumanFeedback* agent, Section 3.1). There were two types of methods for collecting feed-

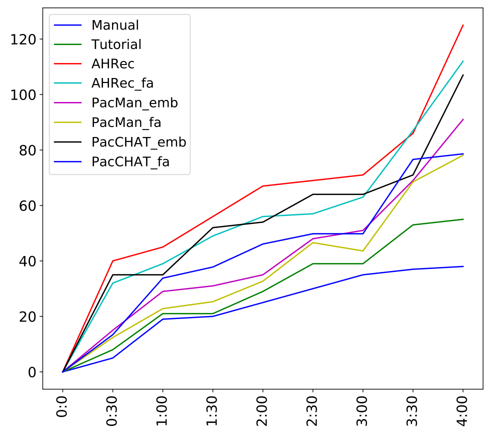

Figure 9: M3 evaluation. The Y-axis corresponds to the average number of avoiding being eliminated by an enemy. *AHRec* users group learned quicker how to defend themselves against either humans or AI agents.

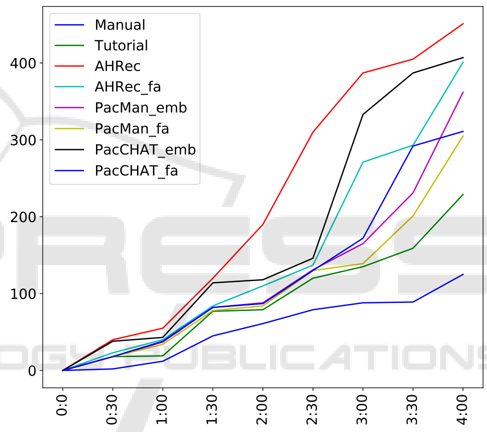

Figure 10: M4 evaluation. The Y-axis corresponds to the average number of destroyed entities. *AHRec* users group learned quicker how to attack and destroy other agents, either humans or AI agents.

back for each individual:

Live Feedback. After receiving a suggestion, the user had a non-intrusive UI component to evaluate the advice received as one of the tree options: (a) Useful, (b) Too repetitive (e.g. user could receive suggestions from same category suggestion too many times, but this does not fit its style of play), (c) Too many in a short interval of time.

Offline Feedback. After the 4 hours sessions each user could write a detailed feedback on different topics: (a) How do you rate the game in general? (b) What did you like about the tutoring system? (c) How would you compare it to *Manual* or *Tutorial*? (d) What would you change to make it better?

The number of suggestions generated in the 4-hour session depended heavily on observing the performance of individual users, but on average there were $\sim$ 59 (with a response rate of 71%) per user. As an aside, there were 4 users out of 60 (3 in the first group, 1 in the second) who completely disabled the tutoring system because they found it too distracting. The results shown in Table 1 indicate that the tutoring system generally provides good and helpful feedback on most aspects without being too disruptive when the fine-tuned policy is used.

Table 1: Live feedback evaluation results averaged over the number of responses received to each category during play sessions, using both the intermediate and the fine-tuned policy.

Live feedback response category	Percent of users	
	$\pi_{Advice_{v1}}$	π_{Advice}
Useful	38%	65%
Too many in a short time	33%	13%
Too repetitive	19%	19%
Users disabling the tutoring system	10%	3%

R3 Evaluation. For product-ready scenarios where applications such as games need to run as fast as possible and achieve high frame rates, it is important to profile the time required to infer our proposed model. Considering this, we profiled how much time is required on average per frame to run inference (CPU only, no GPU) on an Intel i5 9400 processor. The average evaluation per frame took $\sim 0.27\,milliseconds(ms)$, with variations of $\pm 0.12ms$. The memory footprint for the model was less than 24 MB. These results suggest that the methods described in this paper may be suitable for lower specification systems with limited resource budgets, without compromising the application frame rate.

5 CONCLUSIONS

In this paper, we presented a method to provide live suggestions to the user while using an application. These suggestions aimed to give the user a better understanding of the controls, interface, and dynamics, and to show them how to use the application to their advantage with less effort. Our method uses Reinforcement Learning techniques at its core. The evaluation conducted has shown that our tutoring system is simultaneously efficient for human users, not perceived as a nuisance, adapts to multimodal human behavior, and is fast enough to be deployed in real time with limited resource budget. Our future plans are to improve the method even further by automating the manual work for the database of suggestions made. One idea is to use some modern NLP techniques in this area, since text generation is the most expensive work for this step. Also, our method is currently being used in other games that are planned to be released, so we can further evaluate and improve our methods.

ACKNOWLEDGEMENTS

This research was supported by the European Regional Development Fund, Competitiveness Operational Program 2014-2020 through project IDBC (code SMIS 2014+: 121512).

REFERENCES

Agarwal, R., Machado, M. C., Castro, P. S., and Bellemare, M. G. (2021). Contrastive behavioral similarity embeddings for generalization in reinforcement learning. In *International Conference on Learning Representations*.

Alameda-Pineda, X., Ricci, E., and Sebe, N. (2018). *Multimodal Behavior Analysis in the Wild: Advances and Challenges*. Computer Vision and Pattern Recognition. Elsevier Science.

Ausin, M. S., Azizsoltani, H., Barnes, T., and Chi, M. (2019). Leveraging deep reinforcement learning for pedagogical policy induction in an intelligent tutoring system. In *EDM*.

Beck, J., Woolf, B. P., and Beal, C. R. (2000). Advisor: A machine learning architecture for intelligent tutor construction. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, page 552–557. AAAI Press.

Clouse, J. and Utgoff, P. (1996). On integrating apprentice learning and reinforcement learning.

Fujimoto, S., van Hoof, H., and Meger, D. (2018). Addressing function approximation error in actor-critic methods. *CoRR*, abs/1802.09477.

Haarnoja, T., Zhou, A., Abbeel, P., and Levine, S. (2018). Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor.

Ho, J. and Ermon, S. (2016). Generative adversarial imitation learning. In Lee, D. D., Sugiyama, M., Luxburg, U. V., Guyon, I., and Garnett, R., editors, *Advances in Neural Information Processing Systems 29*, pages 4565–4573. Curran Associates, Inc.

Konda, V. R. and Tsitsiklis, J. N. (2000). Actor-critic algorithms. In Solla, S. A., Leen, T. K., and Müller, K., editors, *Advances in Neural Information Processing Systems 12*, pages 1008–1014. MIT Press.

Malpani, A., Ravindran, B., and Murthy, H. (2011). Personalized intelligent tutoring system using reinforcement learning. In *FLAIRS Conference*.

Martin, K. and Arroyo, I. (2004). Agentx: Using reinforcement learning to improve the effectiveness of intelligent tutoring systems. volume 3220, pages 564–572.

Newstrom, J. (2006). Evaluating training programs: The four levels, by donald l. kirkpatrick. (1994). berrett-koehler. 229 pp. 6:317 – 320.

Paduraru, C. and Paduraru, M. (2019). Automatic difficulty management and testing in games using a framework based on behavior trees and genetic algorithms.

Peltola, T., Çelikok, M. M., Daee, P., and Kaski, S. (2019). Machine teaching of active sequential learners. In *NeurIPS*.

Ren, J., Zeng, Y., Zhou, S., and Zhang, Y. (2021). An experimental study on state representation extraction for vision-based deep reinforcement learning. *Applied Sciences*, 11(21).

Rubens, N., Elahi, M., Sugiyama, M., and Kaplan, D. (2016). *Active Learning in Recommender Systems*, pages 809–846.

Sarma, B. H. S. and Ravindran, B. (2007). Intelligent tutoring systems using reinforcement learning to teach autistic students. In *Home Informatics and Telematics: ICT for The Next Billion*, pages 65–78, Boston, MA. Springer US.

Schaul, T., Quan, J., Antonoglou, I., and Silver, D. (2016). Prioritized experience replay. *CoRR*, abs/1511.05952.

Schulman, J., Levine, S., Moritz, P., Jordan, M. I., and Abbeel, P. (2017a). Trust region policy optimization.

Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017b). Proximal policy optimization algorithms.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. u., and Polosukhin, I. (2017). Attention is all you need. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R., editors, *Advances in Neural Information Processing Systems 30*, pages 5998–6008. Curran Associates, Inc.

Wang, Z. and Taylor, M. E. (2017). Improving reinforcement learning with confidence-based demonstrations. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 3027–3033.

Zhan, Y., Fachantidis, A., Vlahavas, I. P., and Taylor, M. E. (2014). Agents teaching humans in reinforcement learning tasks.