Policy Fusion for Adaptive and Customizable Reinforcement Learning Agents

Alessandro Sestini
Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Firenze,
Florence, Italy
alessandro.sestini@unifi.it

Alexander Kuhnle
University of Cambridge,
Cambridge, United Kingdom
alexander.kuhnle@cantab.net

Andrew D. Bagdanov
Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Firenze,
Florence, Italy
andrew.bagdanov@unifi.it

***Abstract*—In this article we study the problem of training intelligent agents using Reinforcement Learning for the purpose of game development. Unlike systems built to replace human players and to achieve super-human performance, our agents aim to produce meaningful interactions with the player, and at the same time demonstrate behavioral traits as desired by game designers. We show how to combine distinct behavioral policies to obtain a meaningful "fusion" policy which comprises all these behaviors. To this end, we propose four different policy fusion methods for combining pre-trained policies. We further demonstrate how these methods can be used in combination with Inverse Reinforcement Learning in order to create intelligent agents with specific behavioral styles as chosen by game designers, without having to define many and possibly poorly-designed reward functions. Experiments on two different environments indicate that entropy-weighted policy fusion significantly outperforms all others. We provide several practical examples and use-cases for how these methods are indeed useful for video game production and designers.**

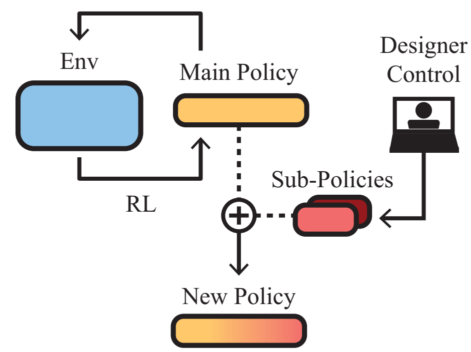


Fig. 1. Summary of our approach: by combining a previously-trained *main policy* with with one or more sub-policies, we can add local behaviors to the overall agent behavior, adapting it to evolving game design instead of discarding the previously trained agent policy and training a novel one from scratch. In this way, designers can create agents that reflect specific choices in easy and understandable ways.

I. Introduction

Non-player characters (NPCs) are a vital factor with respect to the quality of a video game, with the potential to elevate or ruin the player experience. Recent advances in Deep Reinforcement Learning (DRL) have managed to emulate and even supersede human players in the pursuit of achieving (super-)human-level performance, and examples of NPC agents trained with such techniques have been demonstrated for commercial video games [1], [2].

However, mass adoption of DRL by game designers still needs significant technical innovation to make them compatible with the realities of the modern game development process, and to build trust in these approaches [3]. Game AI designers need to be able to guide their exploration of the space of possible behaviors via quick design iterations, without having to tune opaque hyperparameters and study training analytics which are hard to interpret with respect to agent behavior. We are interested in training NPCs that exhibit specific behavioral styles or attitudes *chosen by designers*. However, achieving this using standard DRL training – i.e. via definition and shaping of a complex reward function – can be impractical [4] and expensive since it can require hundreds of thousands of episodes to learn useful policies from scratch. Techniques like Imitation Learning (IL) [5] and Inverse Reinforcement Learning (IRL) [6] can help game designers on the first point, by providing tools to articulate behaviors without requiring handcrafted rewards, however they do not address the sample efficiency problem inherent in DRL training. The policy fusion approaches we propose in this paper pair well in particular with Inverse Reinforcement Learning in that it is possible to train self-contained, *micro-behaviors* from expert (designer) demonstrations that are then *fused* with the main agent policy to adapt it with the new behavior.

We identify three situations that designers face when training game agents with DRL. In all of the following, we assume the existence of an agent trained with standard DRL which, however, does not reflect designer intent – and which therefore requires adaptation to:

- **Enhance performance**, when the agent trained with standard RL does not meet performance requirements – often due to a sparse or poorly crafted reward function. In standard RL, addressing this usually means tuning hyperparameters, modifying the training setup, adjusting the reward function, and then re-training the agent;
- **Add style**, when the agent does not reflect the qualitative behavior desired by designers. For example, an

978-1-6654-3886-5/21/$31.00 ©2021 IEEE

agent might be supposed to act more "sneaky" or more "aggressive" relative to the current agent behavior. In standard RL, this usually means adapting the reward function accordingly and re-train the agent; and
- **Adapt to new features**, when designers change a detail in the game mechanics such as adding a new object/ability/property/etc. Again, this typically requires re-training a previously trained agent from scratch. Moreover, one cannot rely on the fact that the previous training procedure and hyperparameters will continue to work well in this new version of the environment.

In this paper we tackle the aforementioned problems and propose methods to combine diversified policies in ways that avoid re-training. As illustrated in figure 1, instead of discarding the previously trained agent policy and training a novel one from scratch, we instead train a sub-policy to handle the intended behaviour aspect. For example, a sub-policy may be trained only to learn how to handle a novel object added by a designer. We then merge the main policy with the sub-policy via policy fusion, which requires no re-training and ideally results in an agent able to properly handle this new object while maintaining the overall skills of the main game policy. We also demonstrate how these approaches can efficiently be used in combination with IRL in order to train agents which reflect the intention of designers without requiring hand-crafted reward shaping. Experiments on two game environments show that our policy fusion approaches outperform fusion methods from the literature and that fused policies achieve the same – in some cases even better – results than re-training the main policy from scratch using engineered reward functions.

II. Related work

The potential of DRL in video games has been gaining interest from the research community. Here we review recent work most related to our contributions.

DRL *in* video games. Modern video games are environments with complex dynamics which can be useful testbeds for complex DRL algorithms. Results from AlphaStar [7] and OpenAI 5 [8] demonstrated how DRL can be used to create super-human agents in modern complex video-games, while results such as Ecoffet et al. [9] show that we can create super-human agents able to surpass human players in the ATARI games of the Arcade Learning Environment (ALE) [10]. However, our motivation is different in that we do not aim to create super-human agents, but rather to facilitate the use of DRL for NPCs as part of game design.

DRL *for* video games. At the same time, there is an increasing interest from the game development community in how to properly use DRL for video game production. As stated in the introduction, Jacob et al. [3] argued that industry does not need agents build to "beat the game", but rather to produce credible and human-like behaviors. Results such as [11]–[13] are other notable examples of applying DRL to commercial video games, as well as the DeepCrawl work [14], [15], a game environment specicially designed for studying the applicability of DRL in video game production. Within this context, Procedural Content Generation (PCG) has recently gained a lot of attention: works like [16]–[18] have noticed that diverse environment distributions are essential to adequately train and evaluate RL agents for video game production, as these kinds of environments enable generalization of agents when faced with design changes [15].

Inverse Reinforcement Learning. IRL refers to techniques that infer a reward function from human demonstrations, which can subsequently be used to train an RL policy. Adversarial Inverse Reinforcement Learning (AIRL) [19] is a state-of-the-art IRL method, which is enhanced in [20] to work with PCG environments. Other IRL methods different from AIRL, which were tested on simple and static environments, are [21]–[23]. There are only few examples of applying IRL for video game agents: Tucker et al. [24] try to use AIRL on ALE without success, while Source of Madness [1] used Imitation Learning to create diverse game agents in a commercial video game.

Ensemble methods for RL. The use of ensemble methods in RL refers to the practice of combining two or more RL algorithms to increase their performance. Wiering et al. [25] survey different methods to combine multiple RL algorithms. Other examples of ensemble methods in RL are [26]–[30]. However, all of these techniques combine models but with the same goal, instead our aim is to combine policies with different objectives, possibly even orthogonal ones.

Multi-objective learning is motivated by a similar goal, and attempts to combine different reward functions during training to create a complex agent [31]–[34]. Concurrently to our work, Aytemiz et al. [35] started to tackle a similar problem to ours with a multi-objective approach. These approaches, however, try to combine different objectives during training. Instead, our aim is to combine various policies without re-training of agents.

The *policy fusion* approaches we describe in this paper are distinct from ensemble methods and multi-objective learning. Our objective is to combine distributions of different policies after training, while ensemble methods combine decisions and multi-objective learning combine reward functions.

III. Background

IRL plays a critical role for training the sub-policies in our fusion approach. Applying IRL to video games and particularly PCG ones is difficult due dependence of IRL approaches on the number of demonstrations. Demonstration-Efficient Adversarial Inverse Reinforcement Learning (DE-AIRL) [20] addresses this problems and makes it possible to efficiently apply IRL to PCG. This technique is a modification of the Adversarial Inverse Reinforcement Learning (AIRL) [19] algorithm, a GAN [36] approach that alternates between training a discriminator $D_\theta(s,a)$ to distinguish between policy and expert trajectories and optimizing the trajectory-generating policy $\pi(a|s)$. The AIRL discriminator is given by:

$$D_\theta(s,a) = \frac{\exp\{f_{\theta,\omega}(s,a,s')\}}{\exp\{f_{\theta,\omega}(s,a,s')\} + \pi(a|s)}, \quad (1)$$

where $\pi(a|s)$ is the generator policy and $f_{\theta,\omega}(s,a,s') = r_\theta(s,a) + \gamma\phi_\omega(s') - \phi_\omega(s)$ is a potential base reward function which combines a reward function approximator $r(s,a)$ and a reward shaping term ϕ_ω. The objective of the discriminator is to minimize the cross-entropy between expert demonstrations $\tau^E = (s_0^E, a_0^E, \dots)$ and generated trajectories $\tau^\pi = (s_0^\pi, a_0^\pi, \dots)$, while the policy is optimized with respect to the learnt reward function:

$$\hat{r}(s,a) = \log(D_\theta(s,a)) - \log(1 - D_\theta(s,a)). \quad (2)$$

Intuitively, given a state we train the discriminator to recognize if this state comes from the expert dataset or the policy generator, while the aim of the policy generator is to fool the discriminator. The best way for the policy to fool the discriminator is to mimic as well as it can the demonstrations of the expert dataset, making it hard for the discriminator to recognize if a state comes from the dataset or the generator. In the end, the discriminator represents a good reward function for training a near-expert policy with DRL.

DE-AIRL is a modification of the training procedure of AIRL. First, it uses three extension to the original algorithm in order to increase stability:

- **Reward standardization**: the learnt reward model is standardized to have zero mean and a defined standard deviation;
- **Policy dataset expansion**: instead of using one discriminator training step followed by one policy optimization step, as the original algorithm, DE-AIRL performs K iterations of forward RL for every discriminator step; and
- **Fixed number of timesteps**: the terminal condition of the environment is removed, because these conditions encode information about the environment even when the reward function is not observable.

Moreover, to reduce the number of demonstrations needed when working with PCG environments, DE-AIRL introduces an artificially reduced environment, called SeedEnv, that consists of n levels sampled from the fully procedural environment, called ProcEnv. The levels sampled are then used to obtain n expert demonstrations:

$$\text{SeedEnv}(n) = \{L_1, \dots, L_n \mid L_i \sim \text{ProcEnv}\}$$
$$\text{Demos} = \{\tau^{L_i} \mid L_i \in \text{SeedEnv}(n)\}.$$

The reward function is trained via AIRL on the reduced SeedEnv environment instead of the fully-procedural one. In doing so, DE-AIRL forces the discriminator to focus on expert behavior instead of overfitting to levels characteristics, substantially reducing the number of expert demonstrations needed. A reward model trained in this way is able to generalize beyond the SeedEnv and can thus be used to train an agent on the original environment.

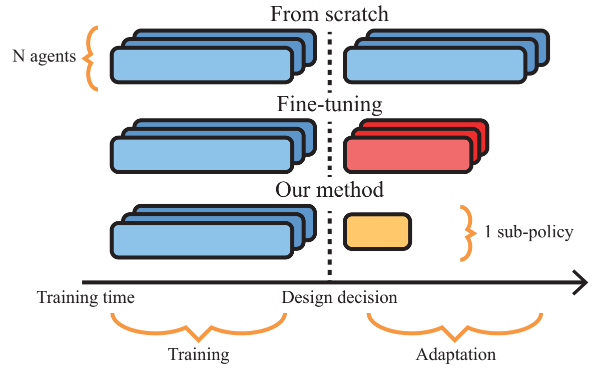


Fig. 2. Training time for different adaptation methods. The easiest approach is to simply re-train all agents from scratch, but this also requires the most time and assumes the training procedure and hyperparameters remain valid after design changes. Instead, with our policy fusion methods we need to only train a single sub-policy without touching the already-trained agents, requiring about a tenth of the time compared to re-training (see section V-C for more details).

IV. Policy Fusion Methods

When game designers want to create game agents with DRL for commercial video games, they must face a large amount of challenges. For example, they go through many iterative design choices that change the environment and force agents to adapt with it [37]; or they need to adapt the final agent behavior because it does not reflect the designers intention [3]; or they need to test new features added to the game under time constraints.

Suppose designers have an agent previously trained with DRL and it must be adapted to a game change. The simplest but most expensive solution is to change something in the training set-up – like reward function or environmental dynamics – and restart the training from scratch. Suppose that to train an agent from scratch will take T hours. Usually, a video game does not only have one agent, but multiple ones with different behaviors or characteristics. Hence, if we suppose that there are N agents, training all of them from scratch will take $N \times T$ hours every time a design decision is made.

Another possibility is to fine-tune the agents every time these decisions are made. To fine-tune an agent we need to change something in the training set-up and continue the DRL training of the previously-trained agent. Therefore, we must spend t_{ft} hours, with generally $t_{ft} \leq T$. However, since there are N agents we need to fine-tune all of them, resulting in a total of $N \times t_{ft}$ hours. Moreover, fine-tuning an agent after a design decision is not always trivial. It is known that DRL suffers from overfitting [18] that can render the process of fine-tuning very difficult. However, this can be mitigated with PCG [15]. Finally, both fine-tuning and training from scratch often require hyper-parameter tuning which may increase the overall training time.

Our objective is to reduce training time and to avoid the re-training of all agents every time we make a design decision. Our main idea is to train sub-policies that explain locally some type of behavior that designers want to teach to agents. Then, we need a policy fusion method that can combine the main policy with the new sub-policy without losing the skills from its main training. In this way, we get an agent that is now able to adapt to the new behavior. For example, suppose we have a well-trained agent but we add a new usable object to the game, which the agent never saw during training and therefore is unable to use it. Designers could now train a sub-policy which learns only how to best use that object. Then, they can combine the previously trained policy with the new sub-policy in order to "teach" the agent to properly use the new feature.

Suppose we spend t_{sp} hours to train the sub-policy, then most likely $t_{sp} \leq t_{ft} \leq T$. This method is completely independent from the number N of previously trained agents, because we need to only train 1 sub-policy and combine it with any of the previously trained agents. So, the total training time in this case is just t_{sp}. Figure 2 shows an example of the different training times.

We propose four different policy fusion methods for DRL to combine different policies. The first two are simple approaches often used for ensemble methods in RL [25], [30], while the other two are novel techniques proposed by us in this paper.

Suppose we have a main policy π_0 and a set of sub-policies π_k for $k = 1, \ldots, K$. Each policy takes as input a state s_t and returns a probability distribution over the same discrete action-space. As is common in the reinforcement learning literature, we will write the policy as $\pi_i(a|s_t)$ to indicate that it is a distribution over actions conditioned on state s_t.

To adapt the main policy to include the behaviors of the sub-policies, without the need for retraining, we propose the following fusions methods which result in a fusion policy π_f:

- **Mixture Policy (MP)**: the resulting policy is the average of the main and all sub-policies:

$$\pi_f(a|s_t) = \frac{1}{K+1} \sum_{k=0}^{K} \pi_k(a|s_t). \tag{3}$$

- **Product Policy (PP)**: the resulting policy is the product of the main and all sub-policies:

$$\pi_f(a|s_t) = \frac{1}{Z} \prod_{k=0}^{K} \pi_k(a|s_t), \tag{4}$$

  where Z is the normalization constant required to make π_f a probability distribution.

- **Entropy-Threshold Policy (ET)**: we compute the entropy of all policies at state s_t and find the sub-policy k^* with minimum entropy:

$$\mathcal{H}_k = -\ln \frac{1}{|A|} \sum_a \pi_k(a|s_t) \ln \pi_k(a|s_t) \tag{5}$$

$$k^* = \operatorname*{argmin}_{k=1,\ldots,K} \mathcal{H}_k, \tag{6}$$

  Where $|A|$ is the cardinality of the state space. Then, if $\mathcal{H}_{k^*}$ is less then $\mathcal{H}_0$ plus threshold ϵ, we perform the action following the sub-policy, otherwise we perform the action following the main policy:

$$\pi_f(a|s_t) = \begin{cases} \pi_{k^*}(a|s_t) & \text{if} \quad \mathcal{H}_{k^*} < \mathcal{H}_0 + \epsilon \\ \pi_0(a|s_t) & \text{otherwise} \end{cases} \tag{7}$$

- **Entropy-Weighted Mixture Policy (EW)**: the resulting policy is a weighted average of the main policy and the minimum-entropy sub-policy identified using equation 6:

$$\pi_f(a|s_t) = \mathcal{H}_{k^*} \times \pi_0(a|s_t) + (1 - \mathcal{H}_{k^*}) \times \pi_{k^*}(a|s_t). \tag{8}$$

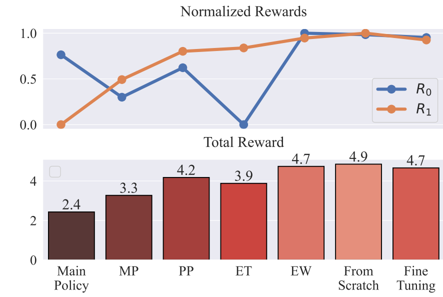


Fig. 3. Results of the different fusion methods in the MiniWorld environment. The plot shows the normalized rewards R_0 and R_1 using the methods described in section IV. The plot below shows the total reward $R_0 + R_1$ achieved by the respective combined policies. Numbers are averages over 1,000 episodes.

V. Experiments

We performed experiments on two different environments to compare the policy fusion methods. In the first experiment, we combine two independent policies trained with hard-coded reward functions in order to understand the performance of each fusion method. For this, we used the MiniWorld [38] environment, a minimalist 3D interior environment simulator for reinforcement learning and robotics research. In the second set of experiments, we used the DeepCrawl environment [14] to evaluate how policy fusion methods can be used in combination with Inverse Reinforcement Learning (IRL).

A. *Results on MiniWorld*

For our first experiment we use the `PickUpObjs` variant of MiniWorld. In this environment, there is a single large room in which the agent must collect objects of two types: red boxes and green balls. A maximum of 5 objects are spawned in random positions. The observation space of this environment is a single RGB image of size $(80, 60, 3)$.

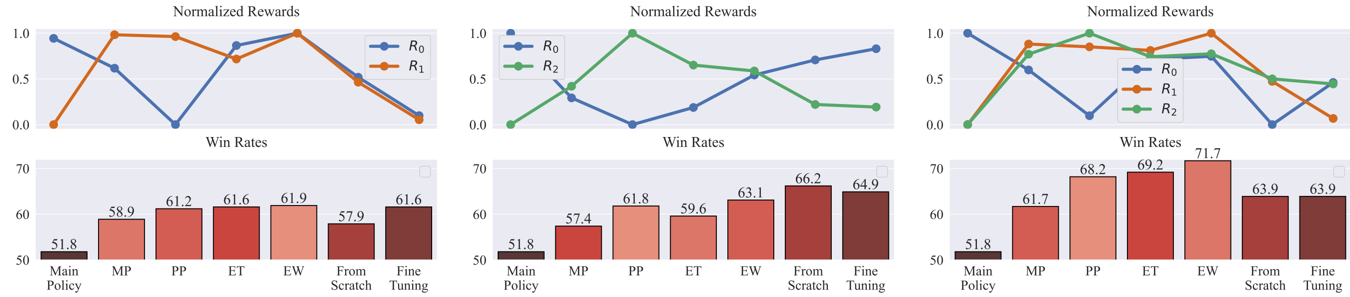
We train two policies with different, hard-coded reward functions. Our main policy π_0 is trained to collect all the red boxes, with reward:

$$R_0 = +1 \text{ for collecting a red box.} \tag{9}$$

A single sub-policy π_1 is then learned with the aim of collecting all the green balls:

$$R_1 = +1 \text{ for collecting a green ball.} \tag{10}$$

The two policies use the same network architecture, consisting of three, stride 2 convolutional layers of size 5, 3 and



(a) (b) (c)

Fig. 4. Results for the *enhancing performance* use-case. In all figures, the top plot shows the normalized rewards for each fusion method: R_0 refers to the original environment reward, R_1 refers to the reward used for training sub-policy π_1, and R_2 to the reward used for training sub-policy π_2. The bottom plot shows the win rate of the combined agents versus the base agent π_0. (a) Results for the combination $\pi_0 + \pi_1$. (b) Results for $\pi_0 + \pi_2$. (c) Results for $\pi_0 + \pi_1 + \pi_2$. Numbers are averages over 1,000 episodes.

3, respectively, and with 32, 32, and 64 channels, respectively. These are followed by two fully connected layers of size 256 and a final fully connected layer followed by a softmax to represent the action distribution. The action space consists of 5 discrete actions: turn left, turn right, move forward, move backward and pick up.

After training the two policies with Proximal Policy Optimization (PPO) [39], we combine them using the methods proposed in section IV. As baselines, we also train a policy from scratch using the combination of $R_0 + R_1$, as well as a policy fine-tuned for the combined reward R_0+R_1 after being trained first with only R_0. Whereas the fusion methods do not involve further learning, the two baseline policies represent "upper bounds" which are trained to optimize the combined reward. For this first experiment, both the main policy and the sub-policy were trained for 1000 episodes, while the baseline was trained for 3000 episodes.

Figure 3 shows the results of these experiments. As the plots illustrate, each policy fusion method improves upon the overall performance of the agent with respect to the combined reward. However, our proposed fusion approach EW outperforms all other methods and achieves the same performance level of the policies trained from scratch or fine-tuned.

This first experiment shows that our proposed policy fusion methods can indeed combine different policies to achieve more complex behavior. The results indicate that, while all yield some improvement, the EW method is by far the best overall, followed by PP. This is a trend we observed in all subsequent experiments.[1]

B. *Results on DeepCrawl*

DeepCrawl is a Roguelike game built for studying the applicability of DRL techniques in video game development [14]. The visible environment at any time is a grid of 10×10 tiles. Each tile can contain the agent, an opponent, an impassable object, or collectible loot. The structure of the map and object locations are procedurally generated at the beginning of each episode. Collectible loot and actors have attributes whose values are randomly chosen as well. The action space consists of 19 different discrete actions: 8 movement actions, 8 ranged attack actions, 2 magic actions and 1 loot action. The environment reward function is defined as:

$$R_0 = -0.01 + \begin{cases} -0.1 & \text{for an impossible move} \\ +10.0 * \mathrm{HP} & \text{for the win} \end{cases}, \tag{11}$$

where HP refers to the normalized agent HP remaining at the moment of defeating an opponent. Agents are trained against an enemy that always makes random moves. For a complete description of the environment and the training set-up, refer to the original paper [14].

As before, we used PPO to train agents in this environment, and they all use the architecture proposed in [15]. We train an agent with the hard-coded reward function in equation 11 as the main policy π_0. The training of this agent reflects the "Ranger" training described in [15]. Subsequently, we train one or more sub-policies and merge those with the main policy. Each of the following sub-policies are trained with DE-AIRL, using the reward approximators of [20]. The majority of demonstrations for DE-AIRL come from a human expert. In the next section we describe the experiments we conducted, each of which are instances of a use-case outlined in section I.

Enhancing performance. Suppose that training with the hard-coded reward function does not result in competitive agents. For example, they do not use certain objects in the map which designers believe would improve their win rate. This can be caused by a badly-designed reward function or a sub-optimal training setup.

To explore this use case, we created two different sub-policies π_1 and π_2 for our first experiment. The first is trained with DE-AIRL to collect and use a specific object in the map, while the second is trained from demonstrations to get all loot which increases agent statistics. We then combine $\pi_0 + \pi_1$ and $\pi_0 + \pi_2$, and let the combined agents fight against the base agent π_0. Pitting agents against the base agent allows us to have some quantitative measure about the different policy

[1] Code to replicate these experiments is at https://tinyurl.com/fusion-rl

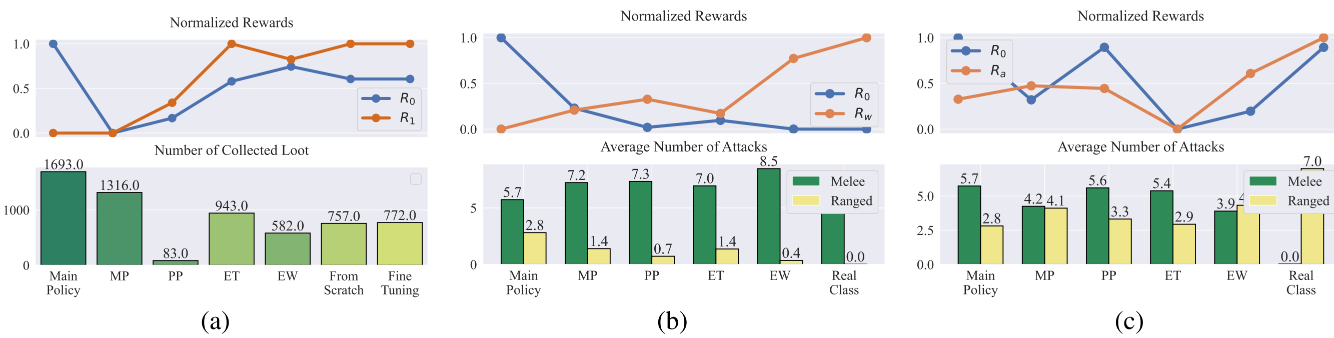


Fig. 5. Results for the *adding style* use-case. In all figures, the top plot shows the individual normalized rewards for the different fusion methods: R_0 refers to the original environment reward, and R_* refers to the reward used to train sub-policy π_*. The bottom plot shows some qualitative statistics which define the "style" of the agent. (a) Combination $\pi_0 + \pi_1$, where π_1 is a sub-policy trained to avoid loot in the map. (b) Combination $\pi_0 + \pi_w$, where π_w is a sub-policy trained to act like a "warrior". (c) Combination $\pi_0 + \pi_a$, where π_a is a sub-policy trained to act like an "archer". Numbers are averages over 1,000 episodes.

fusion methods, as our aim here is to create agents that increase their win rate.

As figure 4a and 4b show, combining the main policy with π_1 and π_2 increases the win rate of the main agent for all fusion methods, with EW being the best, followed by PP. Moreover, all methods decrease the original reward function of the environment. This indicates that the reward is probably not optimally designed in order to yield the most competitive agents (in terms of win rate).

The experiment shows that combining policies can increase the win rate of an agent by more than 10%. Among the methods, EW is the one that decreases the original reward the least while at the same time improving the sub-policy reward. In the π_2 experiment, the EW method even outperforms training from scratch and fine-tuning. This is probably due to the difficulty of combining a hard-coded reward function with a learned one, since these rewards have different scales.

Since both sub-policies individually already increase the competitiveness of the main agent, we also tried to combine them all: $\pi_0 + \pi_1 + \pi_2$. As we expected, figure 4c shows that the combination of all three policies using EW results in an even more competitive agent which wins more than 70% of the games against the base policy π_0. Furthermore, EW clearly outperforms the other methods, including training from scratch and fine-tuning.

Adding style. This use case is about training NPCs that exhibit certain behavioral styles specified and controlled by developers. For example, designers may want an agent to act more "sneakily" or more "aggressively". It is very difficult to computationally specify subjective styles, and expect that the main agent π_0 may likely not reflect the qualitative behavior a designer wants. Since we are not interested in the *competitiveness* of the agent here, but rather being able to control the behavioral aesthetics of NPCs, we conduct a qualitative analysis of agent behavior.

We first train a sub-policy π_1 with DE-AIRL which *avoids* loot in the map while fighting against the opponent. This is an interesting experiment for two reason: on the one hand, we add a new style to the main behavior, while, on the other hand, we are trying to limit the agent avoid a behavior it has already learned during training of π_0. Figure 5a shows that EW method continues to outperform the others, as it decreases the main reward the least while simultaneously augmenting it with the style demonstrated by the designers. Other interesting observations here are that MP does not work at all, while the PP method does add the intended style to the behavior but deviates a lot from the main policy.

For the second experiment in this use-case changed the combat style of π_0 to emulate the style of the other two agent classes in [14] – Warrior and Archer. These two classes are distinguished from each other by the type of attacks they perform in combat: the Warrior relies on melee attacks, while the Archer only on ranged attacks. In contrast, the Ranger, our main policy, performs both melee and ranged attacks. We train Warrior and Archer sub-policies π_w and π_a from demonstrations provided by the pre-trained agents and then combine them with the main policy: $\pi_0 + \pi_w$ and $\pi_0 + \pi_a$. The results are shown in figures 5b and 5c. Indeed, with $\pi_0 + \pi_w$ we are able to modify the Ranger behavior act as a Warrior. With $\pi_0 + \pi_a$ we can change the behavioral style of the Ranger *towards* the Archer, i.e., to perform more ranged than melee attacks, but the combination does not perfectly emulate the Archer behavior. Again, in both cases the EW method outperforms all other fusion methods.

Adapting to new features. suppose an agent π_0 was trained on a certain version of the environment, E_0. Later, after design decisions, some aspect of the game has changed. For example, maybe a new usable item in the map was added, to arrive at a new version of the environment E_1. At this point, designers require an agent π_1 which is aware of and able to properly exploit this new feature. They have two choices: re-train from scratch all previously-trained agents, or train a sub-policy in order to teach it how to use only the new object, and then augment π_0 with this sub-policy.

For our first experiment in this use-case, we added a new usable object to the game. We designed this object so that an NPC using it will have an advantage agents that do not. This way, use of the new object will be reflected in the win rate

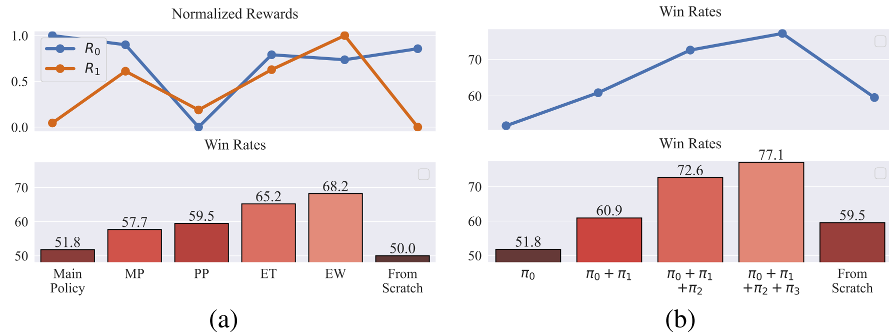


Fig. 6. Results for the *adapting to new features* use-case. (a) The top plot shows normalized rewards for the different policy fusion methods: R_0 refers to the original environment reward and R_1 refers to the reward used for training sub-policy π_1. The bottom plot shows the win rate of the combined agent versus the base agent. All experiments use the E_1 version of the game. (b) Both plots show the win rate of the combined agent versus the plain agent with different combinations of sub-policies with π_0. Numbers averages over 1,000 episodes.

of π_1 over π_0. We train a sub-policy to exploit this feature and combine it with π_0 using policy fusion. This requires an "adapter" which hides the new object from π_0, so to avoid confusion of encountering a new object never seen during training. As baseline, we train an agent from scratch in E_1 but with the original DeepCrawl reward defined in equation 11. Figure 6a shows the results: the combined agent is perfectly capable of using the new object, with EW outperforming all other fusion methods. An interesting observation here is that training from scratch does not learn to make use of the new feature: it just ignores the object and reaches the same level of performance as π_0. This demonstrates that training from scratch in the face of design changes is not a trivial task: we may need to tune hyperparameters or even adapt the reward function to train an agent able to incorporate the new feature into its behavior. But, as we mentioned before, engineering a well-designed reward function for very complex behavior is not easy, and training from scratch takes more time than training only a sub-policy. Our approach enables quick and efficient adaptation to new features in a game environments, which facilitates testing without having to wait for agents to re-train.

Our final experiment simulates multiple, iterative design changes: since we know that fusion methods can adapt to a single change in the environment dynamics, what if we add two more features? For this, we first added a new usable object which increases the statistics of agents that collect it, and then we added an instant death tile. Clearly, an agent that adapts to these changes will have an advantage. For evaluation we compare two agents in an environment with both new features present. The first agent π_0 was trained before these features were added. The second agent is π_0 plus all the sub-policies trained how to use the new features. The baseline is an agent re-trained from scratch on the environment with both changes. Since we have established that EW is the best of the considered fusion methods, we only used this technique here. Figure 6b shows the results for different combinations of sub-policies and π_0. Indeed, we can even combine even all 4 policies and achieve better results than the re-training baseline. We believe that after some hyperparameter tuning, the baseline will likely be able to perform better than our method. However, this would come at a very high cost in term of human effort and time.

C. *Training Times*

Taking the last experiment of V-B as illustrative example, training the main policy takes about 66 hours, while training a single sub-policy requires about 6 hours. Using policy fusion thus requires only an additional 6 hours for each adaptation, whereas it takes another 66 hours per adaptation to re-train main policy from scratch. Our method is about 10 times faster than the standard approach. All training was performed on a NVIDIA RTX 2080 SUPER GPU with 8GB RAM.

VI. Conclusions

Training intelligent agents in complex environments using Deep Reinforcement Learning is difficult and time-consuming, and moreover requires specialized knowledge of both the domain and state-of-the-art deep learning techniques. In this paper we presented several policy fusion methods that can *combine* policies with the aim of adapting or modifying behavior in the face of game design changes – all without requiring retraining of agents to cope with these changes. Our experiments clearly show that the Entropy-Weighted Mixture (EW) fusion technique significantly outperforms the others methods, and in some cases even surpasses the agent re-trained from scratch or fine-tuned using the combined reward function. Of all methods, EW strikes the best balance of maintaining the original policy behavior while simultaneously augmenting it with the sub-policy's new "style".

Our experiments also showed how these fusion methods can be used in combination with Inverse Reinforcement Learning to create varied and complex behavior without defining new reward functions, which contrasts to the prevailing perception that IRL is difficult to exploit in high-dimensional state spaces [37].

Creating credible and meaningful NPC behavior is a difficult task. We think that our work is a first step towards more controlled behavior design by blending interpretable fusion and

DRL training. In future work, we are planning to investigate more sophisticated policy fusion methods which give more control over agent behavior. For instance, we are currently studying ways to learn which policy is the best to use in a particular state. Finally, we are interested in comparing multi-objective techniques for specifying attitudes (e.g. [35]) to our fusion methods.

REFERENCES

[1] "Source of Madness," https://sourceofmadness.com, accessed: 2021-03-02.

[2] "MotoGP21," https://motogpvideogame.com/it/, accessed: 2021-03-02.

[3] M. Jacob, S. Devlin, and K. Hofmann, "It's unwieldy and it takes a lot of time. Challenges and opportunities for creating agents in commercial games," in *16th AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, 2020.

[4] D. Amodei, C. Olah, J. Steinhardt, P. Christiano, J. Schulman, and D. Mané, "Concrete problems in AI safety," *arXiv preprint arXiv:1606.06565*, 2016.

[5] M. Bain and C. Sammut, "A framework for behavioural cloning." in *Machine Intelligence 15*, 1995, pp. 103–129.

[6] A. Y. Ng and S. Russell, "Algorithms for inverse reinforcement learning," in *International Conference on Machine Learning*, 2000.

[7] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev *et al.*, "Grandmaster level in StarCraft II using multi-agent reinforcement learning," *Nature*, vol. 575, no. 7782, pp. 350–354, 2019.

[8] OpenAI, C. Berner, G. Brockman, B. Chan, V. Cheung, P. Debiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. Pachocki, M. Petrov, H. P. de Oliveira Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, and S. Zhang, "Dota 2 with large scale deep reinforcement learning," *arXiv preprint 1912.06680*, 2019.

[9] A. Ecoffet, J. Huizinga, J. Lehman, K. O. Stanley, and J. Clune, "First return, then explore," *Nature*, vol. 590, no. 7847, pp. 580–586, Feb 2021. [Online]. Available: https://doi.org/10.1038/s41586-020-03157-9

[10] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *Journal of Artificial Intelligence Research*, vol. 47, pp. 253–279, 2013.

[11] O. Delalleau, M. Peter, E. Alonso, and A. Logut, "Discrete and continuous action representation for practical RL in video games," *Proceedings of AAAI-20 Workshop on Reinforcement Learning in Games*, 2019.

[12] E. Alonso, M. Peter, D. Goumard, and J. Romoff, "Deep reinforcement learning for navigation in AAA video games," *arXiv preprint arXiv:2011.04764*, 2020.

[13] M. Pleines, F. Zimmer, and V. Berges, "Action spaces in deep reinforcement learning to mimic human input devices," in *2019 IEEE Conference on Games (CoG)*, 2019, pp. 1–8.

[14] A. Sestini, A. Kuhnle, and A. D. Bagdanov, "Deepcrawl: Deep reinforcement learning for turn based strategy games," in *Proceedings of AIIDE Workshop on Experimental AI in Games*, 2019.

[15] ——, "Deep policy networks for NPC behaviors that adapt to changing design parameters in roguelike games," *Proceedings of AAAI Workshop on Reinforcement Learning in Games*, 2021.

[16] K. Cobbe, C. Hesse, J. Hilton, and J. Schulman, "Leveraging procedural generation to benchmark reinforcement learning," *arXiv preprint arXiv:1912.01588*, 2019.

[17] S. Risi and J. Togelius, "Procedural content generation: from automatically generating game levels to increasing generality in machine learning," *arXiv preprint arXiv:1911.13071*, 2019.

[18] N. Justesen, R. Torrado, P. Bontrager, A. Khalifa, J. Togelius, and S. Risi, "Illuminating generalization in deep reinforcement learning through procedural level generation," *Proceedings of NIPS Workshop on Deep Reinforcement Learning*, 2018.

[19] J. Fu, K. Luo, and S. Levine, "Learning robust rewards with adverserial inverse reinforcement learning," in *International Conference on Learning Representations*, 2018.

[20] A. Sestini, A. Kuhnle, and A. D. Bagdanov, "Demonstration-efficient inverse reinforcement learning in procedurally generated environments," *Proceedings of AAAI Workshop on Reinforcement Learning in Games*, 2021.

[21] D. Brown, W. Goo, P. Nagarajan, and S. Niekum, "Extrapolating beyond suboptimal demonstrations via inverse reinforcement learning from observations," in *International Conference on Machine Learning*. PMLR, 2019, pp. 783–792.

[22] P. F. Christiano, J. Leike, T. Brown, M. Martic, S. Legg, and D. Amodei, "Deep reinforcement learning from human preferences," in *Advances in Neural Information Processing Systems*, 2017.

[23] B. Ibarz, J. Leike, T. Pohlen, G. Irving, S. Legg, and D. Amodei, "Reward learning from human preferences and demonstrations in Atari," in *Advances in Neural Information Processing Systems*, 2018.

[24] A. Tucker, A. Gleave, and S. Russell, "Inverse reinforcement learning for video games," in *Proceedings of NIPS Workshop on Deep Reinforcement Learning*, 2018.

[25] M. A. Wiering and H. Van Hasselt, "Ensemble algorithms in reinforcement learning," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 38, no. 4, pp. 930–936, 2008.

[26] S. Faußer and F. Schwenker, "Ensemble methods for reinforcement learning with function approximation," in *International Workshop on Multiple Classifier Systems*. Springer, 2011, pp. 56–65.

[27] A. Hans and S. Udluft, "Ensembles of neural networks for robust reinforcement learning," in *2010 Ninth International Conference on Machine Learning and Applications*. IEEE, 2010, pp. 401–406.

[28] X. B. Peng, G. Berseth, and M. Van de Panne, "Terrain-adaptive locomotion skills using deep reinforcement learning," *ACM Transactions on Graphics (TOG)*, vol. 35, no. 4, pp. 1–12, 2016.

[29] B. Xu, X. Hu, X. Tang, X. Lin, H. Li, D. Rathod, and Z. Filipi, "Ensemble reinforcement learning-based supervisory control of hybrid electric vehicle for fuel economy improvement," *IEEE Transactions on Transportation Electrification*, vol. 6, no. 2, pp. 717–727, 2020.

[30] X. B. Peng, M. Chang, G. Zhang, P. Abbeel, and S. Levine, "Mcp: Learning composable hierarchical control with multiplicative compositional policies," *arXiv preprint arXiv:1905.09808*, 2019.

[31] N. Vithayathil Varghese and Q. H. Mahmoud, "A survey of multi-task deep reinforcement learning," *Electronics*, vol. 9, no. 9, p. 1363, 2020.

[32] S. Huang and S. Ontañón, "Action guidance: Getting the best of sparse rewards and shaped rewards for real-time strategy games," *arXiv preprint arXiv:2010.03956*, 2020.

[33] T. Brys, A. Nowé, D. Kudenko, and M. Taylor, "Combining multiple correlated reward and shaping signals by measuring confidence," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 28, no. 1, 2014.

[34] T. Brys, A. Harutyunyan, P. Vrancx, M. E. Taylor, D. Kudenko, and A. Nowé, "Multi-objectivization of reinforcement learning problems by reward shaping," in *2014 international joint conference on neural networks (IJCNN)*. IEEE, 2014, pp. 2315–2322.

[35] "Designer-centered reinforcement learning," https://www.microsoft.com/en-us/research/blog/designer-centered-reinforcement-learning/, accessed: 2021-03-02.

[36] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Advances in Neural Information Processing Systems*, 2014.

[37] Y. Zhao, I. Borovikov, F. D. M. Silva, A. Beirami, J. Rupert, C. Somers, J. Harder, J. Kolen, J. Pinto, R. Pourabolghasem *et al.*, "Winning isn't everything: Enhancing game development with intelligent agents," *IEEE Transactions on Games*, 2020.

[38] M. Chevalier-Boisvert, "gym-miniworld environment for openai gym," https://github.com/maximecb/gym-miniworld, 2018.

[39] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.