Deep policy networks for NPC behaviors that adapt to changing design parameters in Roguelike games

Alessandro Sestini,[1] **Alexander Kuhnle,**[2] **Andrew D. Bagdanov**[1]

[1]Dipartimento di Ingegneria dell'Informazione, Università degli Studi di Firenze, Florence, Italy
[2]Department of Computer Science and Technology, University of Cambridge, United Kingdom
{alessandro.sestini, andrew.bagdanov}@unifi.it, alexander.kuhnle@cantab.net

Abstract

Recent advances in Deep Reinforcement Learning (DRL) have largely focused on improving the performance of agents with the aim of replacing humans in known and well-defined environments. The use of these techniques as a game design tool for video game production, where the aim is instead to create Non-Player Character (NPC) behaviors, has received relatively little attention until recently. Turn-based strategy games like Roguelikes, for example, present unique challenges to DRL. In particular, the categorical nature of their complex game state, composed of many entities with different attributes, requires agents able to learn how to compare and prioritize these entities. Moreover, this complexity often leads to agents that overfit to states seen during training and that are unable to generalize in the face of design changes made during development. In this paper we propose two network architectures which, when combined with a *procedural loot generation* system, are able to better handle complex categorical state spaces and to mitigate the need for retraining forced by design decisions. The first is based on a dense embedding of the categorical input space that abstracts the discrete observation model and renders trained agents more able to generalize. The second proposed architecture is more general and is based on a Transformer network able to reason relationally about input and input attributes. Our experimental evaluation demonstrates that new agents have better adaptation capacity with respect to a baseline architecture, making this framework more robust to dynamic gameplay changes during development. Based on the results shown in this paper, we believe that these solutions represent a step forward towards making DRL more accessible to the gaming industry.

1 Introduction

In the gaming industry, Artificial Intelligence (AI) systems that control Non-Player Characters (NPCs) represent a vital component in the quality of games, with the potential to elevate or break the player experience. Recently, examples of NPC agents trained with Deep Reinforcement Learning (DRL) techniques have been demonstrated for commercial video games, however mass adoption by game designers requires significant technical innovation to build trust in these approaches (Jacob, Devlin, and Hofmann 2020). A step in this direction was the DeepCrawl prototype (Sestini, Kuhnle, and Bagdanov 2019), a Roguelike game where all NPCs were moved by DRL algorithms.

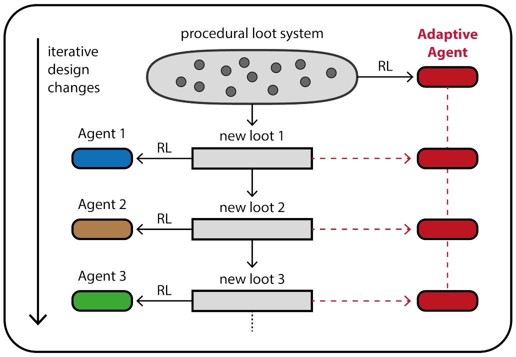
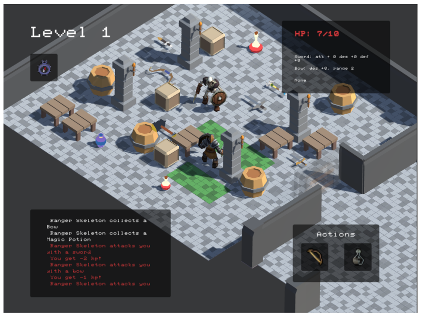


Figure 1: A summary of our approach. The left side illustrates the original DeepCrawl framework in which the agent must be retrained every time the loot distribution changes (e.g. for balancing the overall game). Our approach is shown on the right: with the new adaptive architectures we can create agents which can learn from a new procedural loot system. Agents trained in this way are able to adapt to a changing loot distribution without the need to retrain.

In this paper we address the challenges of *adaptation* and *scaling*, described by the aforementioned authors, and also encountered in DeepCrawl. DRL algorithms are extremely sensitive to design changes in the environment, since they fundamentally change the way agents "see" the game world around them. Even seemingly minor changes can force a complete retraining of all agents. This is mostly due to the categorical nature of the input state space which makes the network overfit to the specific entities seen during training, leaving it without the capacity to generalize to unseen states. Collectible objects in DeepCrawl and their effect on the game, for example, must be predefined by developers, and are represented by unique integer IDs and not by their effect on the player. This can be an important problem during game development: if developers want to change parameters, for example to balance gameplay, they require agents which can



HP: 7/10

Sword: atk + 0 des +0 def +0

Bow: des +0, range 2

None

Ranger Skeleton collects a Bow
Ranger Skeleton collects a Magic Potion
Ranger Skeleton attacks you with a sword
You get -2 hp!
Ranger Skeleton attacks you with a bow
You get -1 hp!
Ranger Skeleton attacks you

Figure 2: Screenshot of the DeepCrawl game. For more information about gameplay elements see section 3 and the original DeepCrawl paper (Sestini, Kuhnle, and Bagdanov 2019).

handle these modifications and do not require retraining. This makes it difficult to adapt an existing agent to new scenarios, resulting in inappropriate agent behavior when NPC agents are used in environments for which they were not designed.

Moreover, with the NPC model architecture of the original DeepCrawl work it is not possible to extend the set of available loot or loot types without completely retraining agents from scratch. This is largely due to specific DeepCrawl network architecture: the policy network contains initial *embedding layers* that make it possible for the network to learn a continuous vectorial representation encoding the meaning of and differences between *categorical* inputs. As mentioned above, in this setting each loot item must be identified by a unique ID in order to be understandable by agents. For this reason, if designers want to add new loot types, for example changing the object definition in order to have a different number of attribute bonuses, it is difficult or impossible to define a unique ID for each object *a priori* – particularly if the attribute bonuses are determined randomly during game play.

To mitigate these problems we implemented a new *procedural loot generation* system and incorporated it into the training protocol: instead of a fixed list of discrete items, in our new system an item is parametrized by a fixed set of attributes, potentially even an extensible set of attributes. These values are drawn from a uniform distribution when generating a new training episode and increase the intrinsic properties of actors when collected. We also propose two alternative policy network architectures that are able to handle the new procedural loot system. As we illustrate in figure 1, the new system combined with procedural loot generation during training renders trained NPCs more adaptive and scalable from a game developer's perspective. This new AI system helps in the design of NPC agents while being robust to iterative design changes across the loot distribution that can happen during video game development. As our experiments show, these new agents are perfectly capable of adapting to loot distributions they have never seen during training, without the need to retrain.

2 Related work

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

Procedurally Generated Environments. There is a growing interest in DRL algorithms applied in environments with Procedural Content Generation (PCG) systems: (Cobbe et al. 2019) demonstrated that diverse environment distributions are essential to adequately train and evaluate RL agents, as they observed that agents can overfit to exceptionally large training sets. On the same page are (Risi and Togelius 2019), who stated that often an algorithm will not learn a general policy, but instead a policy that only works for a particular version of a particular task with specific initial parameters. (Justesen et al. 2018) explored how procedurally generated levels during training can increase generalization, showing that for some games procedural level generation enables generalization to new levels within the same distribution. Subsequently, the growing need for a PCG environments was also demonstrated by (Küttler et al. 2020), (Chevalier-Boisvert, Willems, and Pal 2019), and (Juliani et al. 2019).

DRL *in* video games. Modern video games are environments with complex dynamics, and these environments are useful testbeds for testing complex DRL algorithms. Some notable examples are: (Vinyals et al. 2019) that uses a specific deep neural network architecture based on Transformers (Vaswani et al. 2017) able to create super-human agents for StarCraft, and (OpenAI et al. 2019) that use embedding layers similar to (Sestini, Kuhnle, and Bagdanov 2019) to manage the inner attributes of the agent and other heroes in DOTA 2 in order to train agents that outperform human players.

DRL *for* video games. At the same time, there is an increasing interest from the game development community on how to properly use DRL for video game development. (Zhao et al. 2020) argued that the industry does not need agents built to "beat the game", but rather to produce human-like behavior to help with game evaluation and balance. (Delalleau et al. 2019) dealt with the importance of having an easy-to-train neural network and how it is important to have a framework that enriches the expressiveness of the policy. (Pleines, Zimmer, and Berges 2019) studied different action-space representations in order to create agents that mimic human input, without being super-human. As already discussed, (Sestini, Kuhnle, and Bagdanov 2019) contributed to this aim, defining a DRL framework suited for the production of turn-based strategy games. Our aim is to improve on the latter framework in order to render it more robust to changes to gameplay mechanics during development – i.e., to render DRL agents more *mechanics-free*.

3 Proposed models

Our work builds upon the DeepCrawl framework. Our overarching goal is to make the system as independent as possible from dynamic changes during the development phase, and we argue that a crucial step in this direction is a *procedural loot generation system* which helps encourage generalizing agent behavior in a fully procedural environment. In particular, we want to fulfill the following desiderata:

- **Performance.** We desire agents able to properly handle a procedural loot system, so they must understand which object is most useful for defeating the game;
- **Adaptation.** Agents must adapt to changes in gameplay mechanics, in particular changes to the loot generation system during playtesting and rebalancing, without the need of retraining; and
- **Scalability.** We desire a framework that can scale in both the number of possible objects and in the number of attribute bonuses of each object type. Moreover, the framework must have limited complexity to facilitating targeting of systems like mobile devices.

With these three new desiderata in mind, we now describe two architectural solutions that satisfy them. Both are significant modifications of the early, frontend layers of the DeepCrawl network that allow it to better manage our new procedural loot system. We begin with a brief introduction of the original DeepCrawl environment and network, and then continue with the description of two different architectures that address the problems defined above.

The DeepCrawl environment and policy network

DeepCrawl (figure 2) is a *Roguelike* game that shares all the typical elements of the genre, such as the *procedurally created environment*, the *turn-based* system, and the *non-modal* characteristic that makes every action available to actors regardless the level of the game. In this environment the player faces one or more agents controlled by a DRL policy network. Player and agents act in procedurally-generated rooms, and both of them have exactly the same characteristics, can perform the same 17 actions, and have access to the same information about the world around them. The reward function is extremely sparse and only gives positive reward in case of victory. Player and agent are aware of a fixed number of personal characteristics such as HP, ATK, DEX, and DEF.

The visible environment at any instant in time is represented by a grid with maximum size of 10×10 tiles. Each tile can contain an agent or player, an impassible object, or collectible loot. Each of these entities are represent with a categorical integer ID. Loot can be of three types: melee weapons, ranged weapons, or potions. There is a fixed set of loot, each of which increases an actor characteristic by a predefined value according to the type of object. In this context, the agent must learn which loot ID is the best to have in order to win the game.

Success and failure in DeepCrawl is based on direct competition between the player and one or more agents guided by a deep policy network trained using DRL. Player and agents have exactly the same characteristics, can perform the same actions, and have access to the same information concerning the world around them.

The input state is divided into one global view, the whole grid map, and two local views, smaller maps centered around the agent's position at different scales, that are passed as input to a convolutional neural network. A fourth input branch takes as input an array of discrete values containing information about the agent and the player. Due to the categorical nature of the input state space described above, we call this architecture a *Categorical network* and the overall details are illustrated in figure 3. We refer to the first input branches as State Embedding module and the fourth input branches as Property module. In this paper we focus mainly on the State Embedding module.

As was discussed in the introduction, this input structure limits the adaptation nature of the trained agents. We overcome this limitation by first defining and implementing a different way to generate collectible items in the environment.

Procedural loot system

In our proposed parametric loot system, each object has a number of attributes whose values during training are drawn from a uniform distribution when generating an environment for a new training episode. When an actor collects an item, the actor characteristics will increase or decrease according to the attributes of the instance of the looted object. In our current implementation, each object has the same four attributes corresponding to the four characteristics of the actors. This is not a requirement, however, rather it reflects the original design and implementation of categorical loot system in the original DeepCrawl game.

This system brings a lot of benefits to DeepCrawl: it makes the game more complex and varied, with the corresponding possibility of creating more convincing NPCs and player/environment interactions. The environment is now fully procedural, which should increase the generalization of the agents. Moreover, during playtesting developers can choose either to use random objects or to define a set of fixed objects with fixed attributes in order to balance the game.

However, to enjoy these benefits the policy networks trained for agent behaviors must be able to accommodate this new procedural loot system. The network described above cannot easily do this due to the categorical nature of its input space. Thus we propose two new solutions.

Dense embedding policy network

Our first model is a straightforward extension of the one used in the original DeepCrawl paper. We were inspired by the ideas of (OpenAI et al. 2019) and DeepCrawl to treat the map of categorical inputs via embedding layers. In contrast to these approaches, however, we use multi-channel maps where each channel represents a different categorical value:

- The first channel represents the type of entity in that position:
  - 0 = impassable object;
  - 1 = empty tile;
  - 2 = agent;
  - 3 = player;
  - 4 = melee weapon;
  - 5 = range weapon; and
  - 6 = potion item.
- The other channels each contain an attribute of the object in that tile, represented by a categorical value. For instance, if a tile contains a melee weapon, its attribute bonuses

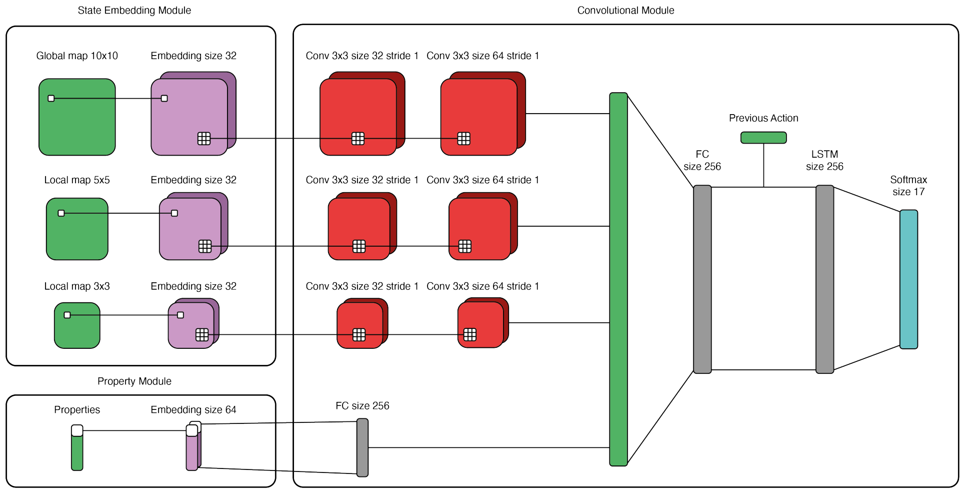
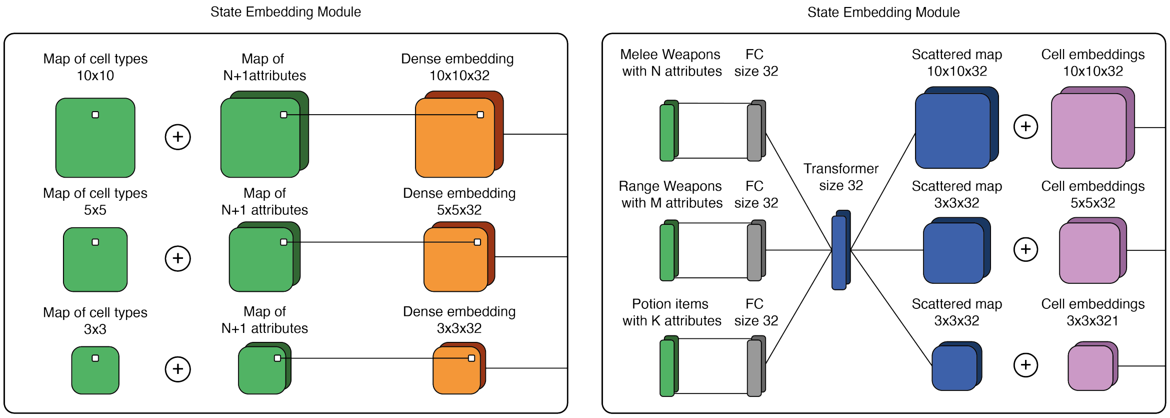


Figure 3: The Categorical network used for NPCs in DeepCrawl (see section 3 for a detailed description). This network architecture is not able to properly handle a procedural loot generation system due to the *State Embedding Module* that requires categorical IDs for each entity in the game. Our approach replaces this module with two possible alternatives shown in figure 4.



(a) Dense Embedding module

(b) Transformer Embedding module

Figure 4: The two new architectures proposed in this paper: (a) The Dense Embedding module, and (b) the Transformer Embedding module. These modules replace the *State Embedding Module* in the original architecture shown in figure 3, while the rest of the policy network is left unchanged. See section 3 for a detailed description.

like health points (HP), attack (ATK), defense (DEF), and dexterity (DEX) are represented by an array of all attributes (plus tile type): `[TYPE, HP, ATK, DEF, DEX]`. If the tile does not contain loot (like an impassable object), this array is filled with the special value `no-attribute`: `[TYPE, NONE, NONE, NONE, NONE]`.

This multi-channel map input, which like the original DeepCrawl network as shown in figure 3, is divided in global and local views and passed through what we refer to as *"dense embedding"* layers: multiple categorical values are combined together and mapped to their corresponding fixed-size continuous representation by a single dense embedding operation. To implement the dense embedding operation, we simply convert each channel into a one-hot representation and apply a 1×1 convolution with stride 1 and $tanh$ activation through all channels. In the special case of a single channel, the operation is equivalent to standard embedding layers. The full model architecture is shown in figure 4a.

This architecture satisfies the requirements we are looking for: the framework is independent from attribute changes to the loot system as long as the types of character attributes remain the same. Moreover, if developers want to change the set of attributes during production, it is no longer necessary to change the entire agent architecture, only the corresponding channels of the dense embedding layer need to be added or removed. As an additional benefit, the network size remains relatively small. A detailed analysis of experimental results for this architecture are given in section 4.

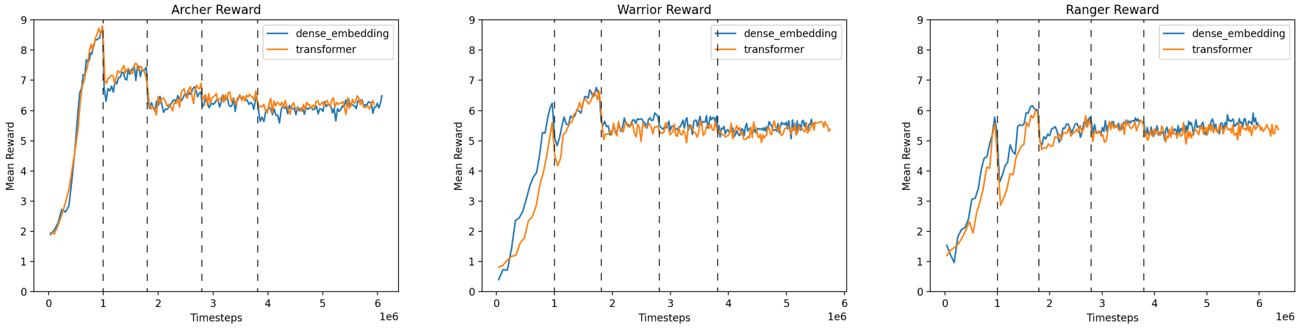


Figure 5: Mean reward during the training phase for all classes as a function of timestep. From left to right: archer, warrior, and ranger. The dashed vertical lines on the plots delineate the different curriculum phases, which are the same as in (Sestini, Kuhnle, and Bagdanov 2019).

Transformer-based policy network

We propose an alternative model based on the recently popular Transformer architecture (Vaswani et al. 2017), and particularly its self-attention layer which has also been successfully applied as state encoder in RL applications (Baker et al. 2019; Vinyals et al. 2019; Zambaldi et al. 2018). This model uses self-attention to iteratively reason about the relations between entities in a scene, and is expected to improve upon the efficiency and generalization capacity over convolutions by more explicitly focusing on entity-entity relations.

Concretely, the self-attention layer takes as input the set of entities e_i for which we want to compute interactions (not including auxiliary `no-attribute` objects), and then computes a multi-head dot-product attention (Vaswani et al. 2017): given N entities, each is projected to a query q_i, a key k_i and a value v_i embedding, and the self-attention values are computed as

$$A = \text{softmax}\left(\frac{QK^t}{\sqrt{d}}\right)V, \tag{1}$$

where A, Q, K, and V represent the cumulative interactions, queries, keys and values as matrices, and d is the dimensionality of the key vectors. As in the original paper, we use 4 independent such self-attention heads. Subsequently, the output vectors per head are concatenated and passed on to a fully-connected layer, and finally added to the entity vector e_i via residual connection to yield a fixed-size embedding vector per entity.

The Transformer operation thus produces embeddings which encode relations between loot in the environment. In this case we represent each object by an array of its attribute bonuses, normalized between 0 and 1, which are further processed by fully connected layers with shared weights across loot types. Based on these representations, a Transformer layer is applied to reason about loot-loot relations, resulting in a fixed-size embedding per entity. Following the concept of *spatial encoders* from AlphaStar, all entity representations are then scattered into a spatial map so that the embedding at a specific location corresponds to the unit/object placed there.

More specifically, we create an empty map and place the embeddings returned by the Transformer at the corresponding positions where the loot is located in the game. We produce such scattered maps for both global and local views which, as before, are concatenated with the embedding map of categorical tile type and then passed on to the remaining convolutional layers. The full network is shown in figure 4b. This model is, again, independent from changes to the loot generation system, and even if developers change the number of attributes during production, this architecture does not require any adaptation, but can simply be retrained on the new game. The biggest weakness is that this architecture is quite complex and requires more computational resources, which goes against the last desideratum defined in section 3.

4 Experimental results

In this section we report on experiments performed to evaluate differences, advantages, and disadvantages of the two new architectures with respect of the Categorical network. All of our policy networks were implemented using the Tensorforce library (Kuhnle, Schaarschmidt, and Fricke 2017).[1]

We follow the same training setup of our original DeepCrawl work. At the beginning of each episode, the shape and the orientation of the map, as well as the number of impassable and collectible objects and their positions are randomly generated; the initial position of the player and the agent is random, as well as their initial equipment. We also use curriculum learning (Bengio et al. 2009) with the same phases as the original paper and during training the agents fight against an opponent that always makes *random moves*. The only difference is the addition of the *loot generation system* described in section 3: each collectible item now has four attributes which correspond to and modify the actor properties (HP, DEX, ATK, DEF). At the beginning of each episode these values are drawn randomly from a uniform distribution for each loot object on the map.

We trained three NPC classes (Archer, Warrior, and Ranger (the same as those from the original DeepCrawl paper) using

[1] Code available at http://tiny.cc/ad_npc

the Transformer, Dense Embedding, and the original Categorical deep policy networks. The NPC classes are distinguished from one another by their character attributes – see (Sestini, Kuhnle, and Bagdanov 2019) for a complete description of the training procedure. In the following, we assess each of the main requirements discussed above in section 3 in light of our experimental results.

Performance. Figure 5 shows the training curves for our two proposed policy networks. The two architectures achieve the same reward, demonstrating that both are able to properly handle the new version of the environment. Table 1 shows that, if two agents of the same class fight each other in the testing configuration (where they start with the max amount of HP and their initial equipment are neutral weapons, while the loot in the map is still procedural), the Transformer based policy has a slight advantage against the Dense Embedding network.

We cannot compare these training curves with those in the original work because of the dynamic nature of the environment introduced by the procedural loot system. The only way to compare with the Categorical network is to train agents with it in the new environment after discretizing the loot attributes into potentially very many unique object IDs. We can, however, have agents of the same class but with different policy fight each other in this new environment. As table 1 shows, the proposed policies have higher average success rate with respect to the Categorical policy network. This demonstrates that these solutions better capture the differences between loot objects.

Adaptation. To demonstrate the improved generalization capacity of our proposed network architectures, we tested them by changing the loot distribution from the fully procedural one used during training to a fixed distribution. This new environment has only three different type of weapons: low, medium and high power (both ranged and melee) that have clear differences between each other – similar to the fixed weapons in the original DeepCrawl. For *high power* weapons we mean loot that gives high value bonuses for all attributes, and so forth for medium and low power ones. Based on the four attributes in DeepCrawl, in this variant a high power sword has attribute bonuses of `[+2, +2, +2, +2]`, a medium power sword has `[+0, +0, +0, +0]`, and a low power sword `[-2, -2, -2, -2]`. We refer to this distribution as the *uniform loot distribution*. We then compare the agents, which have been trained with the full procedural loot, in this testing environment. As table 1 shows, our proposed models have a small advantage compared to the original Categorical framework.

In a subsequent experiment, we change the distribution of fixed loot power: there are still three weapon types, low, medium and high power, but the high power weapons are *far* more powerful than the medium and low power ones, which are comparatively similar. More concretely, a high power sword here has attribute bonuses of `[+5, +5, +5, +5]`, a medium sword `[-2, -2, -2, -2]` and a low power sword `[-3, -3, -3, -3]`. We call this distribution the *skewed loot distribution*. As shown in table 1, with this configuration the success rate for our proposed architectures is much higher, outperforming agents trained with the previous framework and hence showing better adaptation than the Categorical architecture to such a change in the balance of the game.

Scalability. To handle the procedural loot system with the Categorical architecture developers must define a fixed set of class IDs prior to training. This is not a trivial task and can quickly become intractable when the number of attributes for each object increases. Moreover, since the Categorical framework does not generalize (see table 1), if developers want to change loot generation they must define a new set of classes, and that forces retraining of agents. Instead, with our proposed solutions developers simply train their agents with procedurally generated loot and can decide after training whether to use, in the final game, random objects or a fixed set of weapons for balancing the game: our agents will manage both situations without the need of retraining.

Both the proposed frameworks properly handle changes in the number of attributes. In this case retraining is mandatory, but developers need not to worry about changing the network architectures: with the Dense Embedding network they need only to add a new channel for each new attribute, while the Transformer based does not require any changes since it is completely independent of loot parameterization. The Transformer embedding can even handle loot with various number of attributes per type, providing a big advantage with respect to Dense Embedding which requires loot with the same number of attributes.

The biggest drawback of the Transformer network is its complexity. While the Dense and Categorical embedding networks require about 1.3 minutes to train 100 episodes, the Transformer network takes twice as long. The average training times for 100 episodes are 3.31 minutes, 1.36 minutes and 1.10 minutes for the Transformer, Dense Embedding and Categorical networks, respectively. Training was performed on an NVIDIA RTX 2080 SUPER GPU with 8GB of RAM.

In a video game design and development context, this is an important aspect to consider: the continuous changes in the gameplay mechanics require many retrainings, and having a small network is essential. In addition, these frameworks must be implemented to target devices with reduced performance, increasing the need for small and efficient models.

Optimization and hyperparameters

We use the Proximal Policy Optimization algorithm (Schulman et al. 2017) to optimize the agent model. The agent is trained over the course of multiple episodes, each of which lasts at most 100 steps. An episode may end with the agent achieving success (i.e. agent victory), failure (i.e. agent death) or reaching the maximum step limit. After every fifth episode, an update of the agent weights is performed based on the previous five episodes. PPO is an Actor-Critic algorithm with two networks to be learned: the actor policy and the critic state-value function. For both the dense embedding and transformer variant, the critic uses the same structure as the policy network.

Hyperparameter values are the same in all experiments and were chosen after a preliminary set of experiments with

Table 1: Success rates averaged over 100 episodes for pairs of policy networks playing against each other in different testing environments. *Procedural Loot* refers to the environment with fully procedural loot, where each item attribute is drawn randomly at the beginning of an episode. *Uniform loot* refers to the variant with a fixed set of only three types of weapons (low, medium and high power ones). *Skewed Loot* refers to another another such variant, one in which the strongest weapons are far more powerful than the other two. For more details about the experimental setup and loot distributions, see section 4. The proposed network architectures generalize better across distributional loot changes compared to the original categorical architecture.

	Transformer vs Dense Embedding			Dense Embedding vs Categorical			Transformer vs Categorical		
	Procedural loot	Uniform loot	Skewed loot	Procedural loot	Uniform loot	Skewed loot	Procedural loot	Uniform loot	Skewed loot
Archer	55%	56%	52%	62%	58%	**67%**	60%	57%	66%
Warrior	50%	52%	50%	60%	56%	**66%**	60%	58%	66%
Ranger	52%	50%	49%	58%	56%	63%	56%	58%	**64%**

different configurations: the policy learning rate $lr_p = 5 \cdot 10^{-6}$, the baseline learning rate $lr_b = 5 \cdot 10^{-4}$, the agent exploration rate $\epsilon = 0.2$, and the discount factor $\gamma = 0.99$. For the transformer architecture, we used a two-headed self-attention layer, with queries, keys and values of size 32 and a two-layers MLP with size 128 and 32.

5 Conclusions

In this paper, we described several extensions to our DeepCrawl framework (Sestini, Kuhnle, and Bagdanov 2019). First, we implemented a procedural loot generation system which augments the game with a degree of complexity that makes the game more compelling as benchmark for DRL algorithms, particularly in the context of game development. Moreover, we proposed two neural network architectures, one based on Dense Embeddings and one based on Transformers, which both show substantially improved performance due to their capabilities to reason about loot and attribute bonuses. Overall, our experimental analysis slightly favors the Dense Embedding approach due to its reduced complexity and computational requirements.

The advantages for game development are twofold. On the one hand, Roguelikes such as DeepCrawl may contain a large number of items, or indeed employ a procedural loot generation system, so the ability to effectively learn how to compare and prioritize loot is important for NPCs. On the other hand, this ability makes NPCs robust to modifications to the loot system during development, without the need to retrain the behavioral models from scratch every time. This is important, first, for the balancing process during playtesting which is crucial to final quality; and second, both our proposed architectures can easily be adapted in the face of major changes to the loot system which may occur during production.

References

Baker, B.; Kanitscheider, I.; Markov, T.; Wu, Y.; Powell, G.; McGrew, B.; and Mordatch, I. 2019. Emergent tool use from multi-agent autocurricula. *arXiv preprint arXiv:1909.07528* .

Bengio, Y.; Louradour, J.; Collobert, R.; and Weston, J. 2009. Curriculum Learning. In *Proceedings of ICML*.

Chevalier-Boisvert, M.; Willems, L.; and Pal, S. 2019. Minimalistic gridworld environment for openai gym. Github Repository. URL https://github.com/maximecb/gym-minigrid.

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

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

Jacob, M.; Devlin, S.; and Hofmann, K. 2020. “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*. Association for the Advancement of Artificial Intelligence (AAAI), Association for the Advancement of Artificial Intelligence (AAAI). URL https://www.microsoft.com/en-us/research/publication/its-unwieldy-and-it-takes-a-lot-of-time-challenges-and-opportunities-for-creating-agents-in-commercial-games/.

Juliani, A.; Khalifa, A.; Berges, V.-P.; Harper, J.; Teng, E.; Henry, H.; Crespi, A.; Togelius, J.; and Lange, D. 2019. Obstacle Tower: A Generalization Challenge in Vision, Control, and Planning. In *Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19*, 2684–2691. International Joint Conferences on Artificial Intelligence Organization. doi:10.24963/ijcai.2019/373. URL https://doi.org/10.24963/ijcai.2019/373.

Justesen, N.; Torrado, R.; Bontrager, P.; Khalifa, A.; Togelius, J.; and Risi, S. 2018. Illuminating Generalization in Deep Reinforcement Learning through Procedural Level Generation. *Proceedings of NeurIPS Workshop on Deep Reinforcement Learning* URL https://sites.google.com/view/deep-rl-workshop-nips-2018/home.

Kuhnle, A.; Schaarschmidt, M.; and Fricke, K. 2017. Tensorforce: a TensorFlow library for applied reinforcement learning. Web page. URL https://github.com/tensorforce/tensorforce. Accessed: January 6, 2019.

Küttler, H.; Nardelli, N.; Miller, A. H.; Raileanu, R.; Selvatici, M.; Grefenstette, E.; and Rocktäschel, T. 2020. The NetHack Learning Environment. *arXiv preprint arXiv:2006.13760* .

OpenAI; Berner, C.; Brockman, G.; Chan, B.; Cheung, V.; Debiak, P.; Dennison, C.; Farhi, D.; Fischer, Q.; Hashme, S.; Hesse, C.; Józefowicz, R.; Gray, S.; Olsson, C.; Pachocki, J.; Petrov, M.; de Oliveira Pinto, H. P.; Raiman, J.; Salimans, T.; Schlatter, J.; Schneider, J.; Sidor, S.; Sutskever, I.; Tang, J.; Wolski, F.; and Zhang, S. 2019. Dota 2 with Large Scale Deep Reinforcement Learning. *arXiv preprint 1912.06680* URL https://arxiv.org/abs/1912.06680.

Pleines, M.; Zimmer, F.; and Berges, V. 2019. Action Spaces in Deep Reinforcement Learning to Mimic Human Input Devices. In *2019 IEEE Conference on Games (CoG)*, 1–8.

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

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

Sestini, A.; Kuhnle, A.; and Bagdanov, A. D. 2019. DeepCrawl: Deep Reinforcement Learning for Turn Based Strategy Games. In *Proceedings of AIIDE Workshop on Experimental AI in Games*.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, Ł.; and Polosukhin, I. 2017. Attention is all you need. In *Advances in neural information processing systems*, 5998–6008.

Vinyals, O.; Babuschkin, I.; Czarnecki, W. M.; Mathieu, M.; Dudzik, A.; Chung, J.; Choi, D. H.; Powell, R.; Ewalds, T.; Georgiev, P.; et al. 2019. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature* 575(7782): 350–354.

Zambaldi, V.; Raposo, D.; Santoro, A.; Bapst, V.; Li, Y.; Babuschkin, I.; Tuyls, K.; Reichert, D.; Lillicrap, T.; Lockhart, E.; et al. 2018. Deep reinforcement learning with relational inductive biases. In *International Conference on Learning Representations*.

Zhao, Y.; Borovikov, I.; Silva, F. D. M.; Beirami, A.; Rupert, J.; Somers, C.; Harder, J.; Kolen, J.; Pinto, J.; Pourabolghasem, R.; et al. 2020. Winning Isn't Everything: Enhancing Game Development with Intelligent Agents. *IEEE Transactions on Games* .