Video Game Description Language Environment for Unity Machine Learning Agents

Mads Johansen
IT University of Copenhagen
Copenhagen, Denmark
madj@itu.dk

Martin Pichlmair
IT University of Copenhagen
Copenhagen, Denmark
mpic@itu.dk

Sebastian Risi
IT University of Copenhagen
Copenhagen, Denmark
sebr@itu.dk

***Abstract*—This paper introduces *UnityVGDL*, a port of the Video Game Description Language (VGDL) to the widely used Unity game engine. Our framework is based on the General Video Game AI (GVGAI) competition framework and implements its core ontology, including a forward model. It integrates the Unity Machine Learning Agents (ML-Agents) toolkit with VGDL to train and run agents in VGDL-described games. We compare baseline learning results between GVGAI and UnityVGDL across four different games and conclude that the Unity port is comparable to the GVGAI framework. UnityVGDL is available at: https://github.com/pyjamads/UnityVGDL**

I. Introduction

In 2018 Juliani et al. [1] introduced a new toolkit for Machine Learning Agents (ML-Agents) within the Unity game engine. Unity is a widely used game engine with extensive functionality and flexibility to create games and simulation environments. The ML-Agents toolkit provides an easy way to integrate reinforcement learning agents, imitation learning agents and scripted agents into the environments created with Unity. The ML-Agents toolkit implements a custom python pipeline for training agents, as well as the OpenAI Gym interface [2].

Juliani et al. attribute the recent significant advances in deep reinforcement learning to the existence of rapid development environments such as the Arcade Learning Environment (ALE) [3], VizDoom [4] and Mujoco [5]. With a large set of games, ALE provided the base for a breakthrough in control-from-pixels called the Deep Q-Network by Mnih et al. [6] and games have been used extensively to test recent machine learning advances [7], [8]. However, while new environments such as the Obstacle Tower continue to be added to the ML-Agents toolkit [9], it only provides a small set of testing environments when compared to ALE.

One neglected field with significant potential to produce advances in AI research is that of General Video Game Playing (GVGP) proposed by Levine et al. [10]. GVGP aims to extend the challenge of playing many different games with the same algorithm, to playing many different games with the same agents. Levine et al. also envisaged "*the development of a Video Game Description Language (VGDL) as a way of concisely specifying video games*" [10]. The vision was a VGDL capable of describing 2D arcade style games like the Atari games found in ALE. The authors argue for hosting competitions in GVGP by challenging AI agents to play previously unseen games.

In 2014 Tom Schaul created an extensible version of VGDL [11] along with a framework for computational intelligence research [12]. Later Perez-Liebana et al. [13] built on VGDL to create the General Video Game AI (GVGAI) framework and ran the GVGAI Competition based on GVGP.

The GVGAI framework contains a large number of different games and game sets. These VGDL games are a mixture of arcade games like those in ALE and interesting computational intelligence challenges. The games vary in goals and types of interactions. Some are imitations of games that exist elsewhere, like Aliens[1], Frogs[2] or Lemmings[3]. Other games were designed as machine learning challenges, like Wait for breakfast (a simple game where you wait to be served breakfast) or the classic T-maze problem for testing reinforcement learning memorization [14]. The GVGAI competition has been running since 2014 and the list of games has been growing steadily ever since.

UnityVGDL expands the VGDL family from Python and Java to Unity and C#. UnityVGDL brings the GVGAI VGDL ontology to Unity and adds support for training agents with the ML-Agents toolkit. By combining those, ML-Agents can benefit from the corpus of games provided by UnityVGDL. Because UnityVGDL uses the GVGAI VGDL ontology, it allows GVGAI VGDL games to be interpreted and viewed in the Unity Editor and compiled as executables.

By integrating Unity and ML-Agents in UnityVGDL, we expose VGDL to a wider audience of potential machine learning researchers. At the same time we introduce VGDL to a wider audience of game creators. To validate the UnityVGDL framework we train reinforcement learning agents and demonstrate that their performance is comparable to GVGAI agents.

The UnityVGDL framework is available on GitHub[4] under the Apache open source license. The repository has instructions on how to set up ML-Agents, and include the same set of assets as GVGAI, available for AI research and competitions.

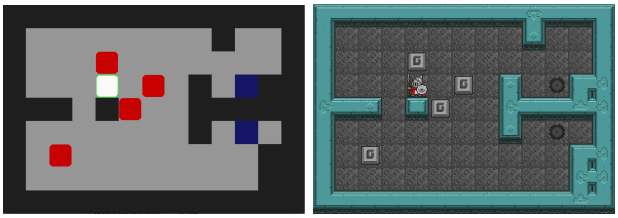


Fig. 1. Visualizations of VGDL Sokoban: left PyVGDL, right GVGAI

II. Related Work

A. Video Game Description Language (VGDL)

PyVGDL is the formalized version of VGDL created by Tom Schaul, focused on describing a wide variety of 2D arcade games. Descriptions are split into two parts. The first is the game description (objects, rules, goals, level mapping). The second is the level description. PyVGDL needs one of each to interpret and run a working game. The game description consists of four sets:

- The `SpriteSet` contains definitions of all objects (including look and behavior) in the game.
- The `LevelMapping` describes the mapping between characters in the level description and objects in the `SpriteSet`.
- The `InteractionSet` defines all interaction effects that happen when two objects collide.
- The `TerminiationSet` describes when the level ends.

PyVGDL generates playable versions from VGDL game and level descriptions using the Pygame library. An example of a running game can be seen in the left visualization in Fig. 1. PyVGDL features 21 example games based on grid physics and six games based on continuous physics. They are meant to show the diversity of games describable in PyVGDL. Imitations of Mario, Zelda, Sokoban, Aliens, Frogger, and Pong are featured, along with the T-maze and traveling salesman problem. PyVGDL also has the option of rendering a first-person perspective of games. An extended Backus-Naur Form description of the full PyVGDL grammar is in [12]. The framework is available on Github[5]. The game shown in Fig. 1 is a simple Sokoban game defined in VGDL; level description and game description can be seen in Fig. 2 and Fig. 3, respectively.

B. General Video Game AI (GVGAI)

Perez-Liebana et al. [13] launched the General Video Game AI Competition in 2014, introducing a Java implementation of the VGDL ontology of PyVGDL called the GVGAI framework. The GVGAI framework also added Sprite assets, improving the visual appearance of the VGDL games as seen on the right in Fig. 1. The framework contains sprite assets licensed by Oryx Design Lab, that are free to use for research and competition[6]. The sprite assets help to communicate the essence of the game to users and make the games more visually appealing. Alongside the GVGAI competition several research projects[7] have been conducted, extending the VGDL capabilities [16], [17] as well as the list of implemented games. The number of implementations of agent types has been growing too, see [13], [18], [19]. The GVGAI framework comes with several different simple, heuristic and planning agents [18], a few based on macro actions [17] and learning agents [19]. It also features functionality to replay recorded agent actions.

The GVGAI framework contains VGDL descriptions for 121 single player games, 49 two-player games, and 11 continuous physics games. Unlike PyVGDL a first person rendering option is not available in the GVGAI framework.

```
wwwwwwwwwwww
w          w  w
w   1         w
w   A 1 w 0ww
www w1  wwwww
w          w 0 w
w 1           ww
w             ww
wwwwwwwwwwww
```

Fig. 2. Simple Sokoban level description. 'A' denotes the Avatar (controlled by player), 'w' denotes walls, '1' denotes boxes and '0' denotes holes.

```
BasicGame
  SpriteSet
    hole   > Immovable color=DARKBLUE
    avatar > MovingAvatar
    box    > Passive
  LevelMapping
    0 > hole
    1 > box
  InteractionSet
    avatar wall > stepBack  #stop at wall
    box avatar  > bounceForward #push box
    box wall    > undoAll #wall stops box
    box box     > undoAll  #box stops box
    box hole    > killSprite #destroy box
  TerminationSet
    SpriteCounter stype=box limit=0 win=True
```

Fig. 3. Game description for a simple Sokoban game. Objects defined in the `SpriteSet` can be used to define interactions, terminations and level mapping. The VGDL Effects such as `stepBack` have comments '#stop at wall' to explain what they do when the two object types collide. The game is won by pushing boxes into holes, until no boxes are left.

[1]Space Invaders (Taito, 1978)
[2]Frogger (Konami, 1981)
[3]Lemmings (DMA Design, 1991)
[4]https://github.com/pyjamads/UnityVGDL
[5]https://github.com/schaul/py-vgdl

[6]Early versions of GVGAI used Open License assets by http://kenney.nl as can be seen in [15]
[7]http://gvgai.net/papers.php

The competition has changed a lot over the years [20]. The 2018 competition featured four tracks: Single Player, Two-Player, Level Generation and Rule Generation. The Single Player and Two-Player tracks are for playing the games. Level Generation aims to generate levels based on a VGDL game description. Rule Generation is a competition to generate game descriptions based on a VGDL level description. The overall structure of the game descriptions is similar in PyVGDL and GVGAI. Yet, there are minor differences between their respective ontology implementations — mostly regarding names and availability of specific Sprite and Effect types (i.e. WalkAvater vs. WalkerAvatar). A description of the VGDL Language and GVGAI ontology can be found online[8] in the GVGAI documentation.

In 2018 Torrado et al. [19] added the OpenAI Gym [2] interface to GVGAI. The OpenAI Gym interface was created to streamline the many different ways reinforcement learning systems interact with environments. The interface makes it much easier to compare and recreate the results of different algorithms across different environments. OpenAI also provides a set of baseline algorithms that can be used for comparison. Torrado et al. planned to compare their results with the results of Mihn et al. [6] using DQN on ALE [3]. However, the scores in the VGDL descriptions have not been modeled after the Atari games; as such the scores between VGDL and Atari games are not directly comparable. Instead, they decided to compare three of OpenAI's baseline reinforcement learning implementations with the state of the art planning agents on a selection of eight VGDL games from the GVGAI framework. The results presented by Torrado et al. will be discussed and compared in later sections.

C. Unity Machine Learning Agents (ML-Agents)

The ML-Agents toolkit allows developers to easily integrate machine learning agents in their games and provides AI researchers with an easily customizable platform to experiment with. The framework defines three different brain types: player brain, heuristic brain (i.e. scripted behavior), and learning brains. These brains control agents in the environment. In the ML-Agents toolkit Juliani et al. [1] chose to implement a baseline reinforcement learning (RL) [21] algorithm based on Proximal-Policy Optimization (PPO) [22].

Additionally for heuristic and RL agents, the ML-Agents toolkit provides an imitation learning agent. This agent gives developers the ability to teach their agents by example. It allows them to draft the kind of behaviors they would like to see in their game with relatively short training periods. Imitation learning can often work as a better reference than a random baseline agent when developing new algorithms.

III. VGDL in Unity

This paper introduces a Unity framework called UnityVGDL. The framework combines the GVGAI VGDL ontology with the ML-Agents toolkit inside Unity. The following section explains how the framework is structured.

A. Architecture

The Unity scenes are structured around the C# VGDL implementation with the outer most layer being the ML-Agents framework as depicted in Fig. 4. The `VGDLAcademy` manages which VGDL game environment to load. The `VGDLAgent` controls the player avatar. The agent uses the `VGDLRunner` to parse and run the VGDL game passed from the academy. The runner parses the game and level descriptions from the Resources folder, and instantiates the `VGDLGame` which in turn instantiates all VGDL sprites, effects, and terminations from the description. The VGDL game updates can be driven by the academy or by regular Unity updates. For learning purposes, letting the academy control updates guarantees the games are updated correctly even when running many instances in parallel. The `VGDLRunner` can also drive the visualization by calling the `VGDLRenderer`. The renderer renders the VGDL game world either to a Render Texture[9] or directly to the back buffer. When dealing with learning agents, the current VGDL ML-Agents implementation only supports visual observations. This means the game is rendered to a Render Texture and then passed as input to the `VGDLAgent`. The C# VGDL implementation in UnityVGDL is based on the GVGAI ontology and its JavaVGDL engine, with a few modifications.

The GVGAI ontology allows UnityVGDL to interpret and play games from the GVGAI backlog of games, however, a few things have been changed under the hood, mainly in regards to lookup tables.

`VGDLSprite`, `VGDLEffect` and `VGDLTermination` class lookup tables were replaced with C# Reflection, for increased extensibility, more closely resembling that in PyVGDL [11]. Further, JavaVGDL implements a type registration and lookup system for the user-defined objects in VGDL. This type registration system converts object names (the `stype`) to a unique integer. UnityVGDL stores the object names directly as keys in C# dictionaries. During development, this direct lookup avoided a layer of abstraction in the code. Since the abstraction also serves as an optimization in the JavaVGDL engine it could become useful again if the forward model needs optimization.

B. UnityVGDL Scenes

The UnityVGDL Testing scene hierarchy seen in Fig. 5 contains a `VGDLAcademy`, a single `VGDLRunner` and `VGDLAgent`. This scene is set up to run a single game with either human, heuristic, or learning brains controlling the VGDL avatar. The scene can be used directly in the editor to load and play a VGDL description with a player brain, to evaluate a trained learning brain play or to build and run a heuristic brain. Additionally, UnityVGDL has several different scenes with learning setups for one, two, four, or eight game instances in parallel. These scenes contain a single `VGDLAcademy`, the adequate number of VGDL instances (`VGDLRunner` and `VGDLAgent`) and a corresponding number of UI elements for

[8]https://github.com/GAIGResearch/GVGAI/wiki/VGDL-Language

[9]https://docs.unity3d.com/Manual/class-RenderTexture.html

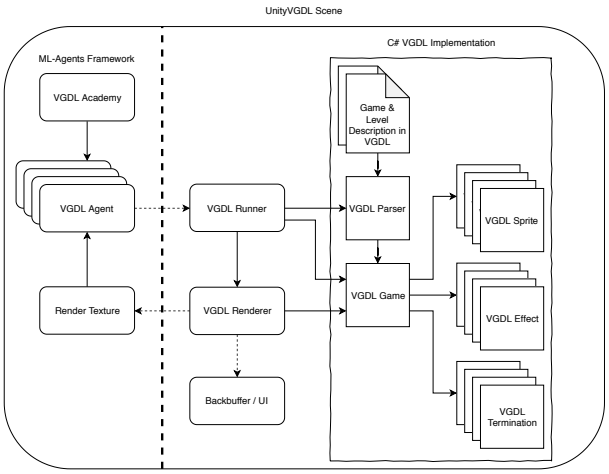


Fig. 4. UnityVGDL Scene Structure

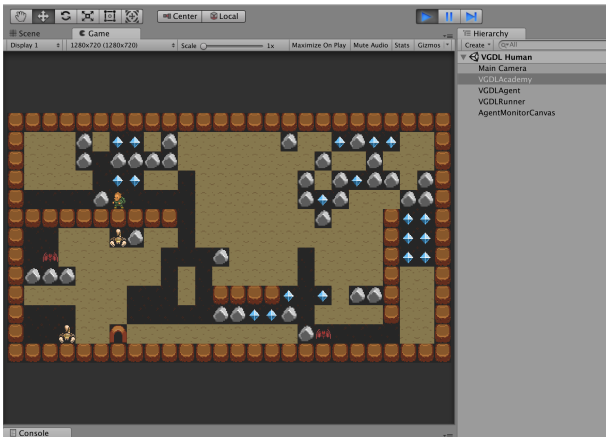


Fig. 5. UnityVGDL Testing scene running one game in the Unity editor

displaying multiple games side by side. The learning scenes can be compiled to executables for faster execution. The four instance version can be seen training on Wait for Breakfast in Fig. 6. Displaying multiple instances at once allows an observer to follow the ongoing training process. The training instances are each rendered in the resolution the agents need (84×84 pixels by default).

ML-Agents handles learning brains in two ways, either training a brain with the python backend or using a previously trained brain with the TensorFlow sharp plugin[10]. To train a learning brain to play a VGDL game, one of the learning scenes has to be edited to load the correct game. Then either a build is compiled or the editor can be used directly as the learning environment. The python ml-agents process is launched to start training. This process will automatically either connect to the editor or launch the executable, based on command line parameters. The default maximum learning steps are set to 50k. After reaching 50k steps the learning

[10]More information about ML-Agents training is available under "documentation" on https://github.com/Unity-Technologies/ml-agents/

Fig. 6. The generated executable from UnityVGDL of the game Wait for Breakfast, rendering four simultaneous game instances. Each game instance renders an 84×84 pixel view of the game world, that the agents get as their visual observation.

stops, and the model is saved. ML-Agents reports progress every 2000 steps to a TensorBoard[11] summery while training.

C. Controlling VGDL Avatars

UnityVGDL does not include the GVGAI competition framework. The ML-Agents framework can replace some of the wrapping functionality that GVGAI provides. The three different brain types in ML-Agents can be used in UnityVGDL, to define how avatars (i.e. player controlled sprites) in VGDL games are controlled. The player brain enables human players to control the avatar, the heuristic brain allows scripted behavior (i.e. random agents or planning agents) and lastly the learning brain for reinforcement or imitation learning. UnityVGDL implements a forward model to allow planning algorithms like Monte Carlo Tree Search as a `VGDLDecision` for the heuristic brain. The `VGDLDecision` class is an extension to the ML-Agents `Decision` class with the VGDL forward model. The current UnityVGDL ML-Agents integration only handles visual observations as input to the learning brains. With advances in deep reinforcement learning [7], learning games from pixels is very common. UnityVGDL does not implement general vector observation due to the many variations in VGDL games. VGDL games vary in size, number of objects and object types. Because the ML-Agents vector input space has to be fixed, capturing every possible VGDL variation was left to future work. Learning brains and `VGDLAgents` would have to be customized to individual games, to accommodate vector observations without a general approach.

D. New Environments for ML-Agents

Juliani et al. [1] suggested using their ML-Agents toolkit to make Unity a simulation platform for learning environments. UnityVGDL is one such environment, extending the available ML-Agents environments with the backlog of VGDL games created for the GVGAI framework. UnityVGDL also allows

[11]https://www.tensorflow.org/guide/summaries_and_tensorboard

researchers to define new games by using and extending the available VGDL ontology. New VGDL games simple have to be added to the resources folder in the project, after which they can be run or used for training with one of the scenes available in UnityVGDL. Extending the ontology can be achieved by inheriting from `VGDLSprite`, `VGDLEffect` or `VGDLTermination` or any of their sub-classes. Extensions will be available immediately because UnityVGDL uses C# reflection to look up the ontology.

IV. Experiments

To validate the UnityVGDL implementation, a selection of games should be compared between the UnityVGDL and JavaVGDL framework. The recent baseline learning results on GVGAI [19] presented a unique opportunity for comparing results on different games. This section outlines the games selected for comparison between UnityVGDL and GVGAI.

A. VGDL Games

Torrado et al. [19] selected eight games with the following consideration in mind:

> *"We tried to get an even distribution across the range going from games that are easy for planning agents, like Aliens, to very difficult, like Superman. The game difficulties are based on the analysis by Bontrager et al. [18]."*

Out of the eight games tested by Torrado et al. four games showed good learning behaviors: Aliens, Boulder Dash, Wait for Breakfast and Zelda. We describe these four games in more detail, which are also the games we choose for our learning test of UnityVGDL:

- **Aliens:** In Aliens the player uses a spaceship at the bottom of the screen to shoot at attacking aliens moving across the screen and dropping bombs. Based on the Atari game of the same name.
- **Boulder Dash**[12]: In Boulder Dash, the player digs tunnels to collect diamonds while avoiding enemies and falling rocks. Based on the Atari game of the same name.
- **Wait for Breakfast**: Game about waiting to be served breakfast, with the simple goal of waiting next to the correct table until the waiter comes by and serves the player. Mostly an interesting ML challenge.
- **Zelda**: Loose interpretation of Zelda or the Atari game Adventure with kill-able enemies and keys, that need to be picked up, before the level can be completed.

Torrado et al. compare their RL results to those of planning agents and of a random agent. By using the same games our results are directly comparable to theirs.

We added a max step limit per episode of 1,500 to ensure training episodes end. Aliens has a natural ending before that limit and Wait for Breakfast has a built-in time limit of 1,500 steps. However Boulder Dash and Zelda both need to be reset to avoid getting stuck. The experiments were all run using compiled executables on a high-end MacBook Pro 2018 laptop for one million steps. The game scene in each executable is set up with one or eight game instances running simultaneously[13], each with a separate agent and the same Learning-Brain attached. The brains use visual observations of 84×84×3 (width×height×RGB). Their action space is discrete with six options [NIL, UP, DOWN, LEFT, RIGHT, USE]. We provided no action masking, which means the agent had to learn to ignore [UP, DOWN] in Aliens and [USE] in Wait for Breakfast. Fig. 6 shows a learning scene with four parallel game instances and their low-resolution visual observation, built as an executable by Unity.

B. ML-Agents Training Parameters

We used the default ML-Agents hyperparameter settings as a comparison between Proximal-Policy Optimization (PPO) [22] and the GVGAI Gym results by Torrado et al. [19]. The parameters with default values and the recommended ranges[14] noted in [brackets] listed below:

- Gamma: 0.99 [0.8 - 0.995]
- Lambda: 0.95 [0.9 - 0.95]
- Batch Size: 1024 [512 - 5120]
- Buffer Size: 10240 [2048 - 409600]
- Number of Epochs: 3 [3 - 10]
- Learning Rate: 3.0e-4 [1e-5 - 1e-3]
- Time Horizon: 64 [32 - 2048]
- Max Steps: 1.0e6* [5e5 - 1e7]
- Beta: 5.0e-3 [1e-4 - 1e-2]
- Epsilon: 0.2 [0.1 - 0.3]
- Normalize: false [true/false]
- Number of Layers: 2 [1 - 3]
- Hidden Units: 128 [32 - 512]

One exception noted with an asterisk (*) in the list above is the Max Steps setting, which was increased to one million, making it comparable with the GVGAI Gym learning results. Each game was evaluated using both a one instance and an eight instance scene, noted as PPO (1) and PPO (8) in the results section.

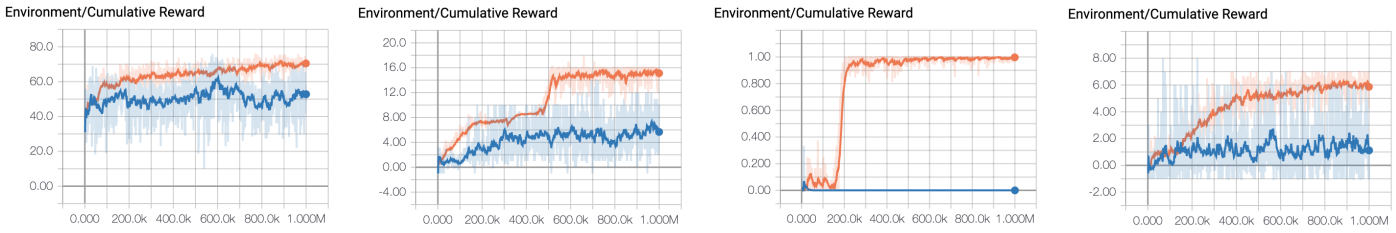
V. Results

ML-Agents allows agents to train either directly in the Unity editor, or using the faster option of a compiled executable. The speed of the training is dependent on the number of rules and size of the VGDL game. As an example training with eight game instances of Zelda, took around an hour per 100,000 steps on a high end 2018 MacBook Pro. No multi-threading was implemented for the game instances inside Unity. Fig. 7 shows the learning progress on each of the four games with one and with eight game instances running in parallel for one million steps. Running eight instances in parallel consistently performs better than a single instance for two reasons. First, more instances lead to more exploration

[12]The game description in UnityVGDL has been modified from the GVGAI implementation by adding a reward for exiting the level.

[13]Only six of them visualized in the build, the last two are rendered off-screen.

[14]https://github.com/Unity-Technologies/ml-agents/blob/master/docs/Training-PPO.md



(a) Aliens (b) Boulder Dash (c) Wait for Breakfast (d) Zelda

Fig. 7. Shows the training progression on the four games using ML-Agents in the UnityVGDL environment. Mean cumulative training reward across eight game instances PPO (8) orange, and one instance PPO (1) blue. The x-axis denotes steps. Note that rewards and the y-axis are different for each game. The lines are smoothed by 90% of the previous value, and the transparent outlines behind them are the actual means. The results clearly show learning progress across all four games, in line with the baselines from [19].

TABLE I
SCORE COMPARISON

Games	Random Agent	Best Planning Agent	GVGAI Gym Learning			ML-Agents Learning	
			DQN	PDDQN	A2C	PPO (1)	PPO (8)
Aliens	52	80.4	75	74	77	49.25 (40%)	71.35 (70%)
Boulder Dash	1.4	16.4	2.5	5	15.5	5.05 (0%)	15.95 (0%)
Wait for Breakfast	0	1	1	1	1	0 (0%)	1 (100%)
Zelda	-0.3*	7.6	4.2	4.2	6	1.4 (0%)	6.3 (0%)

Table I shows the results from [19] compared to final ML-Agents PPO scores averaged score over 20 runs with max step limit set to 1,500. Win-rates shown in parenthesise, i.e. percentage of runs in which the termination objective resulting in a win. Number of instances for PPO denoted in parenthesis.

during training. Secondly, in ML-agents steps are counted by the academy (the focal point for training agents in a game scene), which means the experience count is multiplied by the game instance count.

It is clear from the results that running a single instance with the default parameters is inferior to most other approaches. Preliminary experiments with a variation of tuned parameters have shown potential for better learning. We recommend tuning the hyperparameters or increasing the number of parallel training instances to achieve good results. Running learning on four games is not enough to determine how good the default PPO is compared to other algorithms, but these are baseline results, that give an indication of what is possible using ML-Agents with UnityVGDL.

A. Comparing with GVGAI Gym Results

The learning results of DQN, PDDQN, and A2C using GVGAI Gym should be directly comparable to the results of ML-Agents PPO if the framework is implemented correctly. Although the interpretation of VGDL is not guaranteed to be the same, the UnityVGDL interpretation should match JavaVGDL. The results are not meant to show that one algorithm or framework is superior to the other, but that the baseline results of UnityVGDL are comparable to those found by Torrado et al. The four games were selected exactly because agents can learn to play them well with the baseline algorithms. Finding vast discrepancies between the learning results would indicate an implementation issue or limitation in either framework.

Table I shows the results from Torrado et al. [19] for a random agent[15], the best planning agent score, GVGAI Gym Learning agents and the UnityVGDL ML-Agents score. The final ML-Agents scores were captured for each trained brain by averaging the score of 20 runs of each game with a max step limit of 1,500. Win rates, i.e. the percentage of the 20 runs in which the termination condition resulting in a win, are also shown. It is interesting to note here that no agent managed to achieve a win in neither Zelda nor Boulder Dash. The win condition in these games depend on specific actions, which makes them harder to reach. Like having the key in Zelda, or having gathered exactly ten diamonds in Boulder Dash, prior to reaching the door in either. The baseline GVGAI Gym and ML-agents results are very similar across all games (Table I). Just like the single instance DQN and PDDQN perform worse than A2C, single instance PPO performs worse than eight instance PPO.

VI. DISCUSSION

This paper introduced a new framework that combines the GVGAI VGDL ontology with ML-Agents in Unity. The comparison between learning results in UnityVGDL and GVGAI indicates that there are no major discrepancies between the two. As expected the learning results on the selected games were good, but many harder challenges exist in VGDL games. The learning agents have yet to train on more than a single level of a single game for the baseline results.

The hyperparameter values are meant to be adjusted when using ML-Agents, but the default values were kept except the number of max steps, which was adjusted for comparison with

[15]The score of -5.2 reported by Torrado et al. [19] for a random agent on Zelda must be an error, as Zelda never yields a score below -1.0. A new evaluation was made by averaging the score over 25 runs in GVGAI. The updated score has been marked in the table with an asterisk (*).

GVGAI Gym [19]. The default values provided in the ML-Agents framework have been created to provide the best results when learning from vector observations with continuous action spaces. The UnityVGDL implementation differs in both respects, as it only uses visual observations and has a discrete action space.

Hyperparameter tuning could provide even better learning conditions. Similarly, the imposed limit of 1,500 steps per agent episode was chosen because Wait for Breakfast has a built-in limit of 1,500 steps. During training, the agents would often perform some initial actions and then go wait in a corner for the episode to end. A lower step limit would most likely result in faster real-time training with similar results. Avatars in the GVGAI ontology can limit available actions, e.g. in Aliens the FlakAvatar actions are LEFT, RIGHT and USE. Action masking can be beneficial to limit the state-action space, when learning or planning. However, action masking was not implemented and therefore the neural networks had to learn that some actions did not have any effect on the game. It is unclear whether Torrado et al. removed unavailable actions from the action space of their learning agents. We plan to add action masking to UnityVGDL, which should require minimal changes but could improve learning speed in games with fewer available actions.

A. Adaptations

To maintain compatibility to JavaVGDL, UnityVGDL has inherited parts of the GVGAI framework. Large parts of the GVGAI framework are related to the actual competitions and not part of the VGDL core, so they have not been ported. Some support functionality has been added and made slightly more flexible, like parsing colors and names of fields and classes. UnityVGDL does not use a type lookup system. Instead, classes and fields are looked up using .NET Reflection. Reflection makes it easy to add new effects, sprite types and termination conditions by simply inheriting from another class in the ontology. Another significant change is the usage of `stypes` directly instead of using a registry index, which initially eased implementation. This change could become a performance bottleneck that has to be addressed in the future. UnityVGDL implements a forward model, but it is currently unused, except for relaying game state information to the ML-Agents (for step reward calculation). The VGDLPlayer player interface from JavaVGDL has been kept, albeit it creates a less than elegant implementation of the ML-Agents Agent class; it will most likely be changed in the future.

B. Limitations

Not all sprite types and effects have yet been ported from JavaVGDL to UnityVGDL. The porting process is relatively painless as Java and C# are very similar and most functionality in `VGDLGame` has kept their names in the porting process. In grid-based games path planning has yet to be implemented. Two-player games are only partially implemented, and will not currently work. Continuous physics games have been consciously left as future work. The VGDL implementation should also allow vector observations for VGDL agents as an alternative to visual observations. GVGAI supports using an observation grid containing sprite ids for each square on the grid. A similar approach could be used for vector observations in ML-Agents.

VII. Future Work

A. Missing Features

The relatively small task of porting the remaining class functionality in the ontology remains. All ontology classes are instantiated with the correct data and registered by the parser in `VGDLGame`. Functionality for some rarely used types still needs to be ported from JavaVGDL. For now, the main focus has been on getting the core functionality working and testing it on a small but diverse set of games. For sprites, the general functionality for sprite animations and auto-tiling (automatically choosing sprite from a list based on level map) has not been implemented, and collisions are less optimized than in JavaVGDL. As mentioned earlier, vector observations have also been left for future work.

B. Utilizing the Unity Engine

While a port of the grid-based VGDL engine of JavaVGDL is sufficient for grid physics games, continuous physics games would profit from being translated into native Unity scene hierarchies of game objects that interact with each other based on the VGDL description. This will require the interpretation and custom implementation of generalized VGDL `MonoBehaviours`[16] that can be attached to the game objects based on the VGDL description. Creating a custom implementation of the physics types from VGDL would create a new interesting challenge for transfer learning research.

Generating a custom Unity scene from a VGDL description would also enable Unity game developers to use VGDL to prototype game ideas. Extending the use of VGDL beyond the scope of computational intelligence research would benefit both game developers and researchers. Developers would benefit from a highly abstract description language for creating game prototypes and researchers could gain access to more complex games. A community website like Puzzlescript.net and an extensible ontology could become a rich resource for data and knowledge exchange.

C. New Opportunities through ML-Agents

There are several different avenues to explore using UnityVGDL's ML-Agents setup. The ML-Agents toolkit contains an easy-to-use implementation of a curiosity module, a memory module, and imitation learning. Curiosity has shown good results on games with sparse rewards [23]. VGDL games such as Zelda and Boulder Dash, for which the learning agents did not achieve the win-condition might benefit from curiosity. The memory module is a recurrent neural network (LSTM [14]), which allows agents to use knowledge from previous states when evaluating new states. Agents might

[16] https://docs.unity3d.com/ScriptReference/MonoBehaviour.html

use this capability to figure out when they collected enough diamonds in Boulder Dash to head to the exit. Lastly, imitation learning can be used to mimic behavior, and could in the future be used as a starting point for reinforcement learning agents. The combination of these capabilities makes UnityVGDL and ML-Agents an appealing playground for research.

D. Procedural Content Generation with UnityVGDL

In GVGAI VGDL has been used extensively for procedural content generation (PCG) competitions [20]. Both level generation from known game rules and rule generation from known levels have been a part of the competition for several years. Outside of competitions, agents trained in procedurally generated environments [24], instead of only a particular one, have shown better generalization abilities in various different domains and also have a lower chance of overfitting [25]–[27]. Using UnityVGDL for procedural content generation is thus an excited future possibility.

VIII. Conclusion

The UnityVGDL framework opens new possibilities for competitions in General Video Game Playing [10], research, and general game development. UnityVGDL turns VGDL into a prototyping tool for game developers and a research tool. We would like to encourage the community to help complete and expand the UnityVGDL framework.

Acknowledgements

We would like to thank Niels Justesen, Miguel Gonzlez and Djordje Grbic for fruitful discussions and insightful comments on the framework presented in this paper.

References

[1] Arthur Juliani, Vincent-Pierre Berges, Esh Vckay, Yuan Gao, Hunter Henry, Marwan Mattar, and Danny Lange. Unity: A General Platform for Intelligent Agents. *arXiv:1809.02627 [cs, stat]*, September 2018.

[2] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *arXiv:1606.01540 [cs]*, June 2016.

[3] Marc G. Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. The Arcade Learning Environment: An Evaluation Platform for General Agents. *Journal of Artificial Intelligence Research*, 47:253–279, June 2013.

[4] Michał Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaśkowski. ViZDoom: A Doom-based AI Research Platform for Visual Reinforcement Learning. *arXiv:1605.02097 [cs]*, May 2016.

[5] Emanuel Todorov, Tom Erez, and Yuval Tassa. MuJoCo: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, Vilamoura-Algarve, Portugal, October 2012. IEEE.

[6] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, February 2015.

[7] Niels Justesen, Philip Bontrager, Julian Togelius, and Sebastian Risi. Deep learning for video game playing. *IEEE Transactions on Games*, 2019.

[8] Sebastian Risi and Julian Togelius. Neuroevolution in games: State of the art and open challenges. *IEEE Transactions on Computational Intelligence and AI in Games*, 9(1):25–41, 2015.

[9] Arthur Juliani, Ahmed Khalifa, Vincent-Pierre Berges, Jonathan Harper, Hunter Henry, Adam Crespi, Julian Togelius, and Danny Lange. Obstacle Tower: A Generalization Challenge in Vision, Control, and Planning. *arXiv:1902.01378 [cs]*, February 2019.

[10] John Levine, Clare Congdon, Marc Ebner, Graham Kendall, Simon Lucas, Risto Miikkulainen, Tom Schaul, and Tommy Thompson. General Video Game Playing. *Dagstuhl-Followups*, 6:77, November 2013.

[11] Tom Schaul. An Extensible Description Language for Video Games. *IEEE Transactions on Computational Intelligence and AI in Games*, 6(4):325–331, December 2014.

[12] Tom Schaul. A video game description language for model-based or interactive learning. In *2013 IEEE Conference on Computational Inteligence in Games (CIG)*, pages 1–8, Niagara Falls, ON, Canada, August 2013. IEEE.

[13] Diego Perez-Liebana, Spyridon Samothrakis, Julian Togelius, Tom Schaul, Simon M. Lucas, Adrien Couetoux, Jerry Lee, Chong-U Lim, and Tommy Thompson. The 2014 General Video Game Playing Competition. *IEEE Transactions on Computational Intelligence and AI in Games*, 8(3):229–243, September 2016.

[14] Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Comput.*, 9(8):1735–1780, November 1997.

[15] Diego Perez, Spyridon Samothrakis, and Simon Lucas. Knowledge-based fast evolutionary MCTS for general video game playing. In *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–8, Dortmund, Germany, August 2014. IEEE.

[16] Raluca D. Gaina, Adrien Couetoux, Dennis J. N. J. Soemers, Mark H. M. Winands, Tom Vodopivec, Florian Kirchgesner, Jialin Liu, Simon M. Lucas, and Diego Perez-Liebana. The 2016 Two-Player GVGAI Competition. *IEEE Transactions on Games*, 10(2):209–220, June 2018.

[17] D. Perez-Liebana, M. Stephenson, R. D. Gaina, J. Renz, and S. M. Lucas. Introducing Real World Physics and Macro-Actions to General Video Game AI. In *2017 IEEE Conference on Computational Intelligence and Games (CIG)*, pages 248–255, August 2017.

[18] Philip Bontrager, Ahmed Khalifa, Andre Mendes, and Julian Togelius. Matching Games and Algorithms for General Video Game Playing. In *Twelfth Artificial Intelligence and Interactive Digital Entertainment Conference*, September 2016.

[19] Ruben Rodriguez Torrado, Philip Bontrager, Julian Togelius, Jialin Liu, and Diego Perez-Liebana. Deep Reinforcement Learning for General Video Game AI. *arXiv:1806.02448 [cs, stat]*, June 2018.

[20] Diego Perez-Liebana, Jialin Liu, Ahmed Khalifa, Raluca D. Gaina, Julian Togelius, and Simon M. Lucas. General Video Game AI: A Multi-Track Framework for Evaluating Agents, Games and Content Generation Algorithms. *arXiv:1802.10363 [cs]*, February 2018.

[21] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Adaptive Computation and Machine Learning Series. The MIT Press, Cambridge, MA, second edition edition, 2018.

[22] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal Policy Optimization Algorithms. *arXiv:1707.06347 [cs]*, July 2017.

[23] Deepak Pathak, Pulkit Agrawal, Alexei A. Efros, and Trevor Darrell. Curiosity-driven Exploration by Self-supervised Prediction. *arXiv:1705.05363 [cs, stat]*, May 2017.

[24] Julian Togelius, Georgios N Yannakakis, Kenneth O Stanley, and Cameron Browne. Search-based procedural content generation: A taxonomy and survey. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(3):172–186, 2011.

[25] Niels Justesen, Ruben Rodriguez Torrado, Philip Bontrager, Ahmed Khalifa, Julian Togelius, and Sebastian Risi. Illuminating generalization in deep reinforcement learning through procedural level generation. *NeurIPS 2018 Workshop on Deep Reinforcement Learning*, 2018.

[26] Karl Cobbe, Oleg Klimov, Chris Hesse, Taehoon Kim, and John Schulman. Quantifying generalization in reinforcement learning. *arXiv preprint arXiv:1812.02341*, 2018.

[27] Chiyuan Zhang, Oriol Vinyals, Remi Munos, and Samy Bengio. A study on overfitting in deep reinforcement learning. *arXiv preprint arXiv:1804.06893*, 2018.