Object Detectors as Input for Reinforcement Learning Agents

Benjamin van Oostendorp
DigiPen Institute of Technology
Redmond, Washington
ben.vanoostendorp@digipen.edu

***Abstract*—In this paper we will introduce an alternative approach to handling inputs for reinforcement learning models in video games. Traditional reinforcement learning agents get the whole image, or several full images for training data. The approach we propose instead is pre-processing the input using an object detector with bounding boxes, converting an entire image into bounding boxes and classes. The bounding boxes and classes will then be used for training the reinforcement learning agent. The object detector is fixed after training, and the Q-learning network is the only section that updates with reinforcement learning. The reasoning for this approach is it addresses the ability to extend reinforcement learning to environments where we do not have direct access to the state, and that it could be used to generalize to similar environments. We also experiment using this approach on an application to determine its viability.**

***Index Terms*—object, detection, reinforcement, learning**

I. Introduction

Generalization of agents and models is a problem which prevents tuned agents from being able to complete similar tasks or even similar tasks in slightly different environments [1]. Often, AI agents have to be made specifically for the environment and tuned accordingly, even for environments that are very similar. Other times, the new environments that we delve an agent into does not have a direct link to the environment, meaning we cannot get information directly into an agent as easily as we would for a person. For example, in games where we are not able to access memory or debug features, we cannot get information such as position, score, progress, etc. This can make it hard for an agent to learn a task, especially when it comes to calculating rewards.

Object detectors [2] can handle these problems somewhat easily, since we can track specific objects on the screen between different observations. We can also see whether or not certain objects collide with simple collision checking, allowing us to write easy rules for reward, if desired. This approach also treats observations more like humans would, instead of seeing every individual pixel in an image, it focuses on the specific objects and items inside the image. This method also allows us to change environments easier as there is an amount of information filtering. For example, changing levels in games where the background and floor are now differently colored could radically change how a traditional agent would learn, but we train the object detector to treat them as the same object since they are both in the same category.

Another potential advantage to this approach is explainability. Traditionally, it is very difficult to interpret why a Convolutional Neural Network (CNN) makes the decisions it does. While our approach does not currently attempt to explain the actions of the agent, using techniques such as mutual-information-guided linguistic annotations (MILAN) [3] could lead us to understand the reasoning behind the agent's actions.

In this paper, we first provide an overview of the framework of the system, then our implementation on an environment, and finally an analysis of its performance.

II. Framework

A. Game Selection

The game of choice for the proposed agent to learn is Shovel Knight: Shovel of Hope [4] (referred to from here on as just Shovel Knight). This game is selected as it appeared as an interesting task since there are no methods to gain direct access to its internal variables. The only information from the game that can be gathered is images, unlike many other traditional games that similar agents train on, such as the ATARI 2600 library that most agents train on. It posed an interesting challenge of needing to create a custom reward function as well as not being directly accessible through the environment. Traditionally, the environment is either created to be directly controlled, like in OpenAI's Gym [5], where we can directly send the actions to be performed and the environment processes the actions and gives us all elements of the observation. Shovel Knight is a more traditional game in the sense of a player would interact with the game via a keyboard or controller, process the action within the game, and display the next state for the player to interpret. Training data for the object detector is collected by capturing screenshots every few seconds, and then bounding boxes and classes were then created manually using MakeSense.ai [6].

B. Object Detection

There are many object detectors and classifiers in literature, such as Region-based CNN (RCNN) [2], Spatial Pyramid Pooling (SPP-Net) [7], Single Shot Detector (SSD) [8], and You Only Look Once (YOLO) [9]. Each of these detectors complete two tasks: finding an arbitrary number of objects in a given image, and then classify each object and estimate its position and size with a bounding box. For our framework, we adopted YOLOv5 [10], the newest version of the YOLO

algorithm. The main reason for choosing YOLOv5 is, YOLOR [11] was not available, and after analyzing several models such as RCNN, Fast RCNN [12], Faster RCNN [13], and YOLOv4 [14], YOLOv5 outperformed the other models in terms of accuracy and inference time.

The specific version of YOLOv5 we adopted is the YOLOv5, the second largest version of the full YOLOv5 algorithm. This version handles images that are 640x640 in size, and has an average inference time of 460 milliseconds. This version is chosen as the accuracy and speed were both adequate. We could choose a different network configuration if a faster inference time was necessary, or if the size of the image

The training of the object detector is done separately from the reinforcement agent learning within the game, and is done first. After the training data is created and processed with MakeSense.ai, it was passed to YOLOv5 to train until adequate accuracy was achieved.

C. Agent

The reinforcement learning agent is built using PyTorch [15] and DeepRL-Tutorials [16]. The reinforcement learning agent is a an implementation of the Rainbow agent [17], consisting of a double Q-learning architecture [18], a prioritized replay [19], dueling networks [20], multi-step learning [21], distributional reinforcement learning [22], and noisy nets [23]. Each network consists of 4 noisy layers consisting of 512 neurons each with 2 noisy layers consisting of 512 neurons for both the value and action advantage portion of the dueling network, displayed in Fig. 1. Each layer uses the ReLU activation function and the final output uses the Softmax activation function. The optimizer is PyTorch's implementation of ADAM [24], and the loss calculation is Mean Squared Error.

D. Action

Since Shovel Knight does not allow us to make direct actions into the game, we must "hack" a way to make actions so the agent can interact with the environment and learn to play. This is done by creating a virtual controller with vgamepad [25]. Vgamepad works by using the ViGEM C++ framework [26] with Python bindings, allowing us to create controller inputs and interact with Shovel Knight.

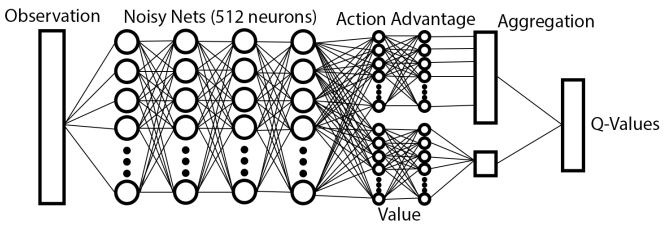

Fig. 1. Layout of the network.

III. Implementation

A. Object Detection Training

The YOLOv5 portion of the framework is trained using screenshots taken from in game, manually labeled with MakeSense.ai, as shown in Fig. 2, and exported in COCO format [27]. The whole data set consists of 121 images processed this way with images being captured throughout the first level in Shovel Knight. Next, the YOLOv5 network is then trained using these images until the results were deemed adequate.

B. Observation Handling

The first part of handling the observation is getting the images of Shovel Knight. The approach we adopted was obtaining screenshots of the game using mss [28], a Python package that captures screenshots and saves them as PNG images, however this could be done many different ways. Next, the image must be reduced in size in order to be processed by the object detector, from 1280x720 to 640x320. We do this by using openCV [29] to scale the image with nearest-neighbor interpolation, which is the fastest. With the new image resized, we then must letterbox the image to make it a square 640x640 image that YOLOv5 can detect from. We do this using one of YOLO's utility functions. Next, the image is passed to the Graphics Processing Unit (GPU) so that it is on the same device as the agent. This is necessary since the inference time of both the agent and YOLOv5 are decreased using the GPU, and transferring data between the GPU and Central Processing Unit (CPU) is slow. The image is then converted to floating point numbers, normalized between 0 and 1, and an extra dimension is added on to match the input requirements of YOLOv5. Finally, the adjusted image is passed through YOLOv5 and Non-Maximum Suppression (NMS) [30] to receive our inferences. The layout of each object in the inference is the left, top, right, and bottom of the bounding box, followed by the confidence and class of the object, as shown in Fig. 3.

Once our inferences are obtained, we must transform them into a format that the agent will understand. In our implementation, the agent is allowed to know the locations of 150 objects. This number can be tuned to fit different environments. The first step in this process is to sort all of the objects from left to right, then bottom to top. This is done to keep objects in similar positions as they were previously

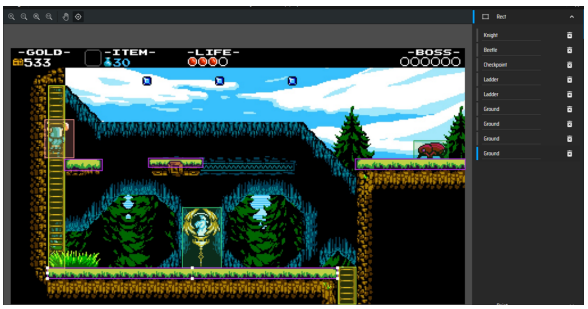

Fig. 2. Example of training data in MakeSense.ai.

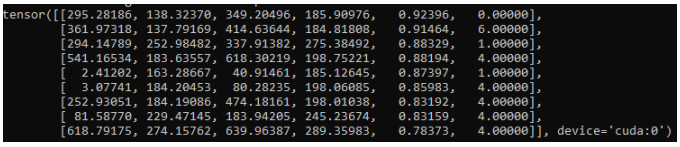

Fig. 3. Example output from YOLOv5 after detection and before normalization. Order is left, top, right, bottom, confidence, and then class.

Fig. 4. Predetermined locations for objects to be placed into, with the health augmented at the very end. The size of the rectangle for each class is roughly how many objects per class in the observation. For example, there are more Ground and Sand object spaces than Spikes.

shown in. Next, all of the objects are extracted from the inferences. We do this by extracting objects based on class, and then discard the class and confidence value for the object. For each class, we flatten the bounding box for each object into a one-dimensional array and then store it in a pre-determined location of the observation array, as shown in Fig. 4. We do this such that objects of the same class always appear in the same location in the observation for every state. Once all of the classes have been moved into the observation array, it is again normalized between 0 and 1, as the object detector's inferences are pixel locations rather than image locations.

The final step in creating the observation is calculating the health of the agent. We detect the agent's health directly by interpreting pixel values at the location where health is displayed in the game. The health is is then augmented onto the bottom of the observation array, and normalized between 0 and 1.

C. Agent Decision Making

Now that the observation is constructed, we pass it to the agent to determine what action to take. The agent computes the Q-values of the observation for each action, and returns the action with the maximum Q-value. This action is an integer value between 0 and 63 (the count of all possible actions), and must be interpreted into controller inputs. We do this by using a bitwise AND for each button the agent can press. Each button acts as a mask of the action passed in, and if the bit is set we press that button, and if the bit is not set, we release that button. After every button is parsed from the action, we update the state of the controller, and it is sent to the virtual controller.

D. Reward Calculation

One of the most difficult portions of handling environments that do not already have reward functions already is writing a custom reward function. One major benefit to this approach is that we can write a reward function based on the health and objects in the observation and derive progression. For the case of Shovel Knight, we derive progression by movement and health. The first check we do is to see if the agent is still alive, and if it is not, we return a large negative reward. If the agent is still alive, the process continues with the following steps. First, the health portion of the reward is calculated by finding if the agent lost health between states. Losing health is a negative reward, and gaining health is a positive reward, scaled by how much health the agent lost/gained. Next, we must try and derive the progression of the agent in the game. Since the camera in Shovel Knight is not stationary at all times, meaning the Knight can sometimes be on the far left or right of the screen, or directly centered with the camera following, we must handle both cases differently. If the Knight (agent controlled character) is not centered, progression calculation is straight forward: we compute the differences in the left and right portions of the bounding box for the Knight between the previous and next state, and average the difference. Movement to the right is a positive reward, and moving to the left is a negative reward, and is directly influenced by how much distance the Knight made between observations; more rightward movement means a higher positive reward.

However, if the Knight is centered, using the approach from above will not net in reward since the bounding box locations of the Knight will be similar in each observation. So instead of tracking the movement of the Knight, we look at objects in the scene and attempt to track a specific object to determine movement. First, we compute the size of each of the objects in the current and next observations. Then, for each object in both the previous and next state, we find a pair of objects with roughly the same bounding box size. We then compare if the objects are in roughly the same position, accounting for the fact that the Knight may have moved between observations and thus the positions will be slightly different as the Camera follows the Knight. If we find two objects of similar size and in similar locations, they are most likely the same object from the previous and next state. Now we do a similar calculation of progression as in the case with the non-centered Knight, but with the objects we tracked instead.

There are a few edge cases which must also be handled. The first important one is that if we do not find the Knight in either of the states. If this is the case, we are unable to determine movement and only use the health for reward. The next edge case is when we find the Knight in both states, but we cannot find any objects in the states or we cannot find one object in the previous and next state to track. If this is the case, we check the action of the agent to determine movement. In general, it is not good practice to parse the action of the agent to determine progression and calculate reward since we do not want to tell the agent what actions to press directly, and this is the only case in which we must do so since there is no information we can otherwise give the agent. The reward in this case is very small, since we want the agent to get back to a state where we can better derive progression. The final edge case is based on how the camera sometimes moves during screen transitions. In some sections of the game, the Knight will enter a new location and the camera will pan very quickly to the new location. If this is the case, we must detect

the camera movement since otherwise we will be giving our agent a positive reward when we are supposed to give it a negative reward, and vice versa. If we check and the camera is panning, which is done by checking the speed at which the Knight is moving between observations since during the pan it will be faster than during normal movement, we take the negative of the progression to correct for the panning.

The final portion of the reward calculation is to compute the weighted sum of the progression and health with tunable parameters. Our current implementation uses 30% of the health reward and 70% of the progression reward.

E. Replay

After the agent selects an action and performs it in the environment, we store the previous state, action, reward, and next state into the replay buffer. After a certain number of frames (which is a tunable parameter) the network is updated by sampling a batch of experiences from the experience replay, computing the loss, and optimizing the online model. After the agent has updated the online model a specific number of times (also a tunable parameter) the target model is updated with the weights of the online network.

IV. Main Results and Analysis

A. Results

The agent is trained for 1000 episodes consisting of a maximum of 1000 frames each, with the reward for each episode being recorded. Upon reaching the end of an episode or dying (a terminal state), the agent resets the environment to the beginning of the first stage, and the next episode begins. Fig. 5 shows the reward at each episode as well as the 100 episode moving average.

B. Analysis

To show the effectiveness of the proposed method we will determine how far the agent has progressed in the game. However, the reward is not enough to show the effectiveness as it is set up to reinforce behavior and not to measure progress in the game. This is most likely due to lack of training time, as a maximum of 1 million frames is not nearly enough time for the agent to learn the environment. The Rainbow agent that learned the ATARI environments started to achieve exceptional results around 30 million frames, and needed 7 million frames to match the performance of Deep Q-Learning (DQN) alone [17].

We believe the reason for this behavior is that the agent learns the parts of the level before progressing further into sections it has not yet learned before progressing further. This is hinted at in Fig. 6 where the agent begins to receive a higher median reward from the previous episodes before it drops significantly, followed by a climb in reward again. However, with better metrics such as episode length, level progression as opposed to just reward, and health lost per episode, the learning and progression of the agent could be better explained.

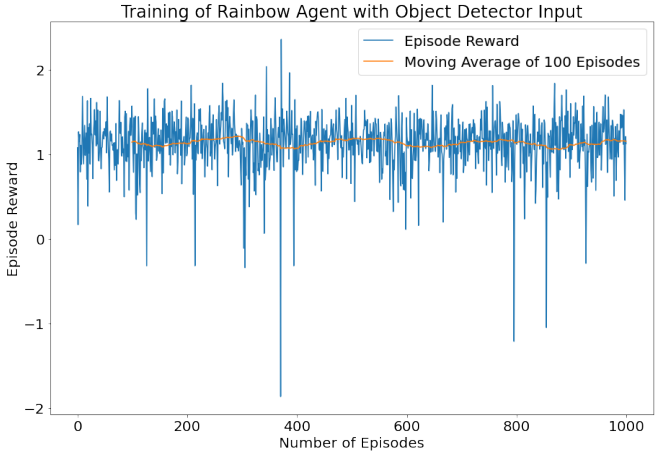

Fig. 5. The agent's reward at the end of each episode as well as 100 episode moving average.

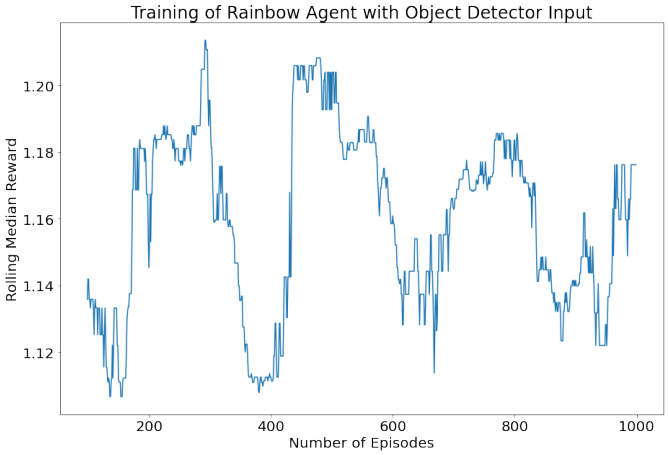

Fig. 6. Rolling median reward of 100 episodes

V. Conclusion

We have proposed a new approach for inputs for reinforcement learning agents in video games and demonstrated its application on Shovel Knight. It has been determined that the agent begins to learn the environment, however more training will be needed for the agent to complete the first level. This form of observation handling could allow agents to learn environments they might not have been able to learn before, as well as provide an alternative approach with the potential benefits of explainable decision making. Some limitations of this approach are that it increases the action decision time of the agent with having to process the input. Future work to be done with this approach would be a comparison to the traditional approaches to reinforcement learning with games and directly compare them.

Acknowledgment

I would like to thank Dr. Barnabas Bede, Dr. Yilin Wu, and Shivam Kumar for all the help they have given me for this project. I would also like to thank DigiPen Institute of

Technology as well as Prof. Peter Toth, Dr. Ola Amayri, Dr. Brigitta Vermesi, and Dr. Pushpak Karnick who all helped me realize my love for machine learning and math.

REFERENCES

[1] D. Malik, Y. Li, en P. Ravikumar, “When Is Generalizable Reinforcement Learning Tractable?”, 01 2021.

[2] R. Girshick, J. Donahue, T. Darrell and J. Malik, "Rich Feature Hierarchies for Accurate Object Detection and Semantic Segmentation," 2014 IEEE Conference on Computer Vision and Pattern Recognition, 2014, pp. 580-587, doi: 10.1109/CVPR.2014.81.

[3] E. Hernandez, S. Schwettmann, D. Bau, T. Bagashvili, A. Torralba, en J. Andreas, “Natural Language Descriptions of Deep Visual Features”, 01 2022.

[4] Shovel Knight: Shovel of Hope. United States of America: Yacht Club Games, 2014.

[5] G. Brockman et al., “OpenAI Gym”, ArXiv, vol abs/1606.01540, 2016.

[6] P. Skalski, “Make Sense”, 2019. [Online]. Available at: https://github.com/SkalskiP/make-sense/.

[7] K. He, X. Zhang, S. Ren and J. Sun, "Spatial Pyramid Pooling in Deep Convolutional Networks for Visual Recognition," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 37, no. 9, pp. 1904-1916, 1 Sept. 2015, doi: 10.1109/TPAMI.2015.2389824.

[8] W. Liu et al., “SSD: Single Shot MultiBox Detector”, in Computer Vision -- ECCV 2016, 2016, bll 21–37.

[9] J. Redmon, S. Divvala, R. Girshick and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR), 2016, pp. 779-788, doi: 10.1109/CVPR.2016.91.

[10] G. Jocher et al., ultralytics/yolov5: v6.1 - TensorRT, TensorFlow Edge TPU and OpenVINO Export and Inference. Zenodo, 2022.

[11] C.-Y. Wang, I.-H. Yeh, en H.-Y. Liao, “You Only Learn One Representation: Unified Network for Multiple Tasks”, 05 2021.

[12] R. Girshick, "Fast R-CNN," 2015 IEEE International Conference on Computer Vision (ICCV), 2015, pp. 1440-1448, doi: 10.1109/ICCV.2015.169.

[13] S. Ren, K. He, R. Girshick and J. Sun, "Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 39, no. 6, pp. 1137-1149, 1 June 2017, doi: 10.1109/TPAMI.2016.2577031.

[14] A. Bochkovskiy, C.-Y. Wang, en H.-Y. Liao, “YOLOv4: Optimal Speed and Accuracy of Object Detection”, 04 2020.

[15] A. Paszke et al., “PyTorch: An Imperative Style, High-Performance Deep Learning Library”, in Advances in Neural Information Processing Systems 32, H. Wallach, H. Larochelle, A. Beygelzimer, F. Alché-Buc, E. Fox, en R. Garnett, Reds Curran Associates, Inc., 2019, bll 8024–8035.

[16] Q. Fettes, “DeepRL-Tutorials”, GitHub repository. GitHub, 2018.

[17] M. Hessel et al., “Rainbow: Combining Improvements in Deep Reinforcement Learning”, 10 2017.

[18] H. van Hasselt, A. Guez, and D. Silver, “Deep Reinforcement Learning with Double Q-learning,” 2015.

[19] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, “Prioritized Experience Replay.” 2015.

[20] Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, and N. de Freitas, “Dueling Network Architectures for Deep Reinforcement Learning.” 2015.

[21] K. De Asis, J. Hernandez-Garcia, G. Holland, en R. Sutton, “Multi-step Reinforcement Learning: A Unifying Algorithm”, 03 2017.

[22] M. Bellemare, W. Dabney, en R. Munos, “A Distributional Perspective on Reinforcement Learning”, 07 2017.

[23] M. Fortunato et al., “Noisy Networks for Exploration”, 06 2017.

[24] D. Kingma en J. Ba, “Adam: A Method for Stochastic Optimization”, International Conference on Learning Representations, 12 2014.

[25] Y. Bouteiller, J. Surette, en J. Zhu, “vgamepad”, GitHub repository. GitHub, 2021.

[26] B. Höglinger-Stelzer en J. Hart, “ViGEm”, GitHub repository. GitHub, 2017.

[27] T.-Y. Lin et al., “Microsoft COCO: Common Objects in Context”, in Computer Vision -- ECCV 2014, 2014, bll 740–755.

[28] M. Schoentgen, “python-mss”, GitHub repository. GitHub, 2016.

[29] G. Bradski, “The OpenCV Library”, Dr. Dobb’s Journal of Software Tools, 2000.

[30] J. Hosang, R. Benenson, en B. Schiele, “Learning non-maximum suppression”, 05 2017.