Vision-Based Screen Navigation Environment for RL Agents

Sai Konkimalla [*1] **Matthew Noto** [*1] **Charles Chen** [*1]

Abstract

Autonomous screen navigation is a very useful task, as automating human interactions with a computer would enable more efficient operation of devices. While there exist some RL and LLM-based screen navigation agents, these agents lack precision in actions such as clicking. In addition these agents are also trained with DOM/XML trees as inputs about the structure of websites, which inhibits true model generalizability from computer web navigation to general computer control. Current RL screen navigation agents also suffer from the latter two issues. For a true end-to-end screen navigation agent, the agent must operate using only visual input. As a first step towards this, we design a new environment which distills the screen navigation problem into its most fundamental form: a graph traversal problem where the agent is only given access to visual data associated with the screen itself. With two variations of this environment with a discrete and continuous action space respectively, we were able to run various off-the-shelf RL algorithms, including DQN, PPO, and DDPG. In our experiments, we tested the limits of the RL algorithms and how they could effective policies to navigate the environment when increasing the image size and the complexity of the underlying tree structure of the environment. In designing the environment and running experiments, we demonstrate that this is an important and challenging problem which RL agents can solve.

1. Introduction

Large language models have demonstrated incredible performance across a wide variety of NLP tasks and have consequently revolutionized research across many different AI fields. This has led to the emergence of LLM agents which possess perception, memory, cognition, and action capabilities, allowing them to perform a plethora of tasks (Wang et al., 2023b). As electronic devices continue to play an instrumental role in our daily lives, an agent which can navigate these devices through the control of screens would be a significant advancement towards general and autonomous intelligence.

While a completely autonomous screen navigation agent is still in its infancy, the focus of much current research on this area is mainly related to using LLM agents to perform these tasks. However, these LLM agents lack precision in clicking, the most fundamental action in the web navigation task (Niu et al., 2024). Despite the fact that they've been under-utilized over the past few years, reinforcement learning agents have shown promise in this screen navigation task (Liu et al., 2018). However, these agents take in a DOM/XML tree of a web page as input in order to learn various navigation tasks, such as booking a flight or forwarding an email.

The use of such a DOM tree prevents true generalizability of such an agent, as in order to navigate other screens which aren't on the web (such as mobile screens), the agent needs to be able to learn this graph structure implicitly. An agent which is capable of navigating a computer screen should also be able to navigate a mobile screen as the two tasks are fundamentally identical and hence shouldn't be treated as distinct.

Fundamentally, the screen navigation task is a graph traversal problem without giving features associated with the graph structure to the agent as an observation. The use of such a framework is more representative of the desired task we want our agent to learn. In particular, with screen navigation, the observations given to the agent is simply the screen itself.

Inspired by environments such as DeepMind's Vokram (Toyama et al., 2021), OpenAI's point maze environment (Fu et al., 2020), we wanted to create a highly abstracted screen navigation environment which "segments" the different clickable portions of a screen into different colored regions. We created both a environment with both discrete states and actions and continuous states and actions. In the

[*]Equal contribution [1]Department of Computer Science, Stanford University, Stanford, California, USA. Correspondence to: Sai Konkimalla <saik@stanford.edu>, Matthew Noto <mattnoto@stanford.edu>, Charles Chen <yifuchen@stanford.edu>.

Proceedings of the 35^{th} International Conference on Machine Learning, Stockholm, Sweden, PMLR 80, 2018.

discrete case, these clickable portions can also be thought of as representing "interesting parts" of a website/app such as various important UI elements on a website/app such as buttons which lead to the a new page/screen, albeit highly simplified and abstracted. Based on the techniques of segmentation from computer vision, we create a highly distilled environment which simply consists of a rectangular grid of randomly colored buttons on a randomly colored background.

The idea of creating this environment is to have a basic environment which we can train RL agents to easily learn a policy on. In particular, with the discrete action space and continuous action space variations of this screen navigation environment, we were able to run further experiments by training DQN and PPO agents on the discrete environment and by training PPO and DDPG agents on the continuous environment. In these tests, we scaled up the size of the screens (which also make up the respective state space) as well as the complexity of the tree structure of the environment to see how robust the RL agents were in learning policies for more and more complex environments.

Beyond the work we did here, there are two main paths in which we can extend on these current results. The first is by making the environment and the formulation of the navigation problem more complex. This primarily consists of expanding the goals of the agent in navigating the environment and by making the synthetic state space more complex (by adding noise and generally more variance in the buttons and screens). The second is by training further RL algorithms on these more complex versions of the environment. Some further tests we considered were performing hyperparameter tuning on the current RL algorithms used, potentially training with a vectorized environment, and using different policies like MCTS, which may be more effective in larger-scale versions of the current environment formulation.

2. Related Work

2.1. LLM-based Navigation Agents

LLMs that possess general world knowledge have been gaining popularity in AI agent research. Some researchers represent the state space and feed LLMs the device's accessibility tree and browser's HTML file (Xie et al., 2023; Gur et al., 2022), while others rely on Multi-Modal Vision Language Model (VLM) to identify pixels of UI elements, reason about the best plan, and generate the precise screen coordinates needed for actions such as mouse clicks (Xie et al., 2024a; Liu et al., 2023; Gao et al., 2024). In addition to using LLMs to process the observed environment, some agents use LLMs as a core component for orchestrating the various sub-modules of an agentic system. Prompting strategies such as ReAct (Yao et al., 2023b), Chain-of-Thought (Wei et al., 2022), and Tree-of-Thought (Yao et al., 2023a) were used to further enhance the reasoning capability of LLMs, since screen navigation often involves long-term planning and iterative refinement. DoraemonGPT uses Monte Carlo Tree Search to explore multiple action plans, and combining retrieval-augmented generation (RAG) to retrieve both external information (web search) as well as internal memory vector stores (memory lookup) (Yang et al., 2024). Despite these efforts, sometimes an agent fail to recover from mistakes in the plan or action (Lee et al., 2024; Hong et al., 2023a), perhaps due to the next-token-prediction training paradigm causing the LLM to become repetitive in its output when it is uncertain.

RAG is used in device-operating agents to retrieve semantic memory (mapping between special tokens in the environment to their special meaning to the task), procedural memory (the trajectory for a successful task completion, or instructions for tool-use), and episodic memory (events that the agent needs to remember to act in future time-steps). Some projects spawn sub-agents to perform memory management and retrieval. RAG has been proven to be a critical part of complex LLM systems, and is an active area of research.

LLMs with function-calling capabilities are leveraged enable skills such as making API calls (Gao et al., 2023) and even invoking additional domain-specific models to process environment information with modalities that the main model does not support (Qin et al., 2024; Yang et al., 2024). For example, generating text captions of video data to enable easier retrieval and search API calls, or using `pyautogui` to control the computer to perform click, type, and hotkey actions (Niu et al., 2024; Sumers et al., 2024).

Since LLMs with coding ability were found to have stronger reasoning capability than other LLMs not trained on code (Madaan et al., 2022), large models fine-tuned for function-calling, such as `gpt-4-turbo`. `gemini`, and `LLaMA` has gained mainstream usage compared to models trained mainly on web-crawl and instruction-tuning datasets, such as the Text-to-Text Transfer Transformer (`T5`).

Despite various efforts, agents based on even the state-of-the-art models (e.g., `gpt-4-vision` and `gemini-1.5-pro`) have shown poor localization accuracy and often give incorrect coordinates for mouse clicks, leading to null or incorrect actions, model context confusion, and unrecoverable trajectories (Hong et al., 2023b; Niu et al., 2024). This may be due to the fact that VLMs pre-process images into resized patches before tokenizing and encoding the information into continuous vector representation spaces. Therefore, without additional tuning, the output coordinates have low accuracy for a given screenshot resolution. Although agents based on open-source LLMs

can be fine-tuned, they display bad generalization capabilities when faced with an operating system with different layouts or icons (Niu et al., 2024; Hong et al., 2023b; Furuta et al., 2024).

A significant portion of techniques used by device-operating agents, such as self-critical planning, skill retrieval, agentic orchestration stem from research focused on embodiment in virtual environments such as Minecraft, which was used as a simulator for testing agentic systems (Fan et al., 2022; Wang et al., 2023b;a; Qin et al., 2024). Agents were evaluated based on whether it learned to mine resource, craft tools, and obtain diamonds. In many projects, researchers used LLMs or VLMs to power the perception, reasoning, skill acquisition, memory, reflection, and instruction-following for the game-playing agents.

Frameworks for agentic systems is an active area of research as well. For example, Autogen from Microsoft lets users easily creating systems of multi-agent that work together towards a common goal (Wu et al., 2023), with an option for having the human in the loop. Another framework, OS-Copilot includes a wide range of tools for performing action-planning and RAG (Wu et al., 2024b). AgentKit is a framework to create complex tasks flowcharts through a graph representation (Wu et al., 2024a). In MetaGPT and ChatDev, autonomous software-building agents are orchestrated where each agent plays a specific role, such as product manager and tester, to develop software together (Hong et al., 2023a; Qian et al., 2023a;b).

2.2. RL-based Navigation Agents

Previous work involving training RL agents to navigate the Web usually involves training agents to follow an certain set of instructions, such as *order a book on Amazon* or *book a flight to Hawaii* which require learning in an environment with very large state and action spaces with very sparse rewards, making this problem difficult. For highly specific instructions such as booking a flight from one specific location to another on a given date, or even just navigating to a specific web page, reaching the target page or the desired objective (in the case of the first example) through exploration can be very difficult. This is because many trajectories will likely contain no reward signal. Previous work has shown that web navigation agent performance can be enhanced through human demonstrations (Liu et al., 2018). However, in incredibly large environments which are representative of highly complex websites, separate human demonstrations are used for each particular environment and as the environment becomes increasingly complex, human demonstrations fail to provide useful learning signal for RL agents. Researchers have tried to mitigate this problem by combining curriculum learning and Q learning along with a bi-LSTM encoder for encoding a DOM tree (Gur et al., 2018).

Other RL agents trained on this task include combining DQNs and attention for navigating environments with visual inputs (Shah et al., 2018).

While all of these agents show some promise in the general web navigation task, the web navigation and mobile screen navigation task are treated as separate which impedes producing generalizable agents.

2.3. Navigation Environments

DeepMind's AndroidEnv (Toyama et al., 2021) provides an incredibly useful open-source platform for RL embedded in an Android ecosystem. There are a variety of toy tasks such as an Angry Birds clone, 2048 clone, and an Asteroids clone. This set of environments also consists of a timer and various other games. The objective behind these environments is to teach an agent how to do certain things on a mobile screen through the use of common apps.

Several benchmarks for web-browsing have been developed to train and evaluate device-operating agents. Mind2Web is a dataset with 2,350 tasks that involve a wide variety of real world scenarios such as ordering a plane ticket (Deng et al., 2023). MiniWob++ extends OpenAI's MiniWob benchmark, that allows a neural network to learn to generate actions from the HTML and screenshot inputs (Liu et al., 2018). WebArena uses HTML DOM trees, as well as accessibility trees provided from Microsoft Windows back-end to represent the world information (Koh et al., 2024). OSWorld is a simulator for 369 long-horizon tasks spanning major operating systems such as Ubuntu, Windows, and MacOS (Xie et al., 2024b). At this time, few benchmarks exist for purely vision-based navigation agents on mobile UI layouts.

3. Environment

3.1. Motivation

Our custom environment was created with the Gymnasium API and we took inspiration from the DeepMind Vokram environment (Toyama et al., 2021) and the OpenAI Gym (Brockman et al., 2016) Point Maze environment. The inspiration from the Vokram environment was its emphasis on a simplified clickable UI with buttons. This heavily inspired our idea of creating an environment with a randomized background color and randomized button colors with transitions to the next cscreen. Furthermore, the point maze environment from OpenAI inspired us to think about the problem as a graph exploration task.

The primary algorithm we decided to test first is DQN (Mnih et al., 2013) because of its success on Atari environments whereby an agent takes actions according to an observation which only consists of visual input.

The crux of this work was implementing a version of the initial screen navigation which an RL formulation would be effective in finding a policy for. As a result, we created an environment with the following properties:

1. The states are images (but more simplified than normal apps/webpages)
2. The underlying structure of the environment resembles a graphical structure similar to most websites

In particular, we first created artificial images of "app-like" screens as seen below to the left, and we designed square buttons on a distinctly colored screen representing different accessible parts of the app.

We satisfied the second property by maintaining certain properties of the underlying graphical structure of the environment, including the presence of a "home node" which all other nodes have directed edges toward, as well as a specialized "skeleton and chaining" algorithm to ensure the graph was connected from each node and resembled a realistic graphical structure from an app/website.

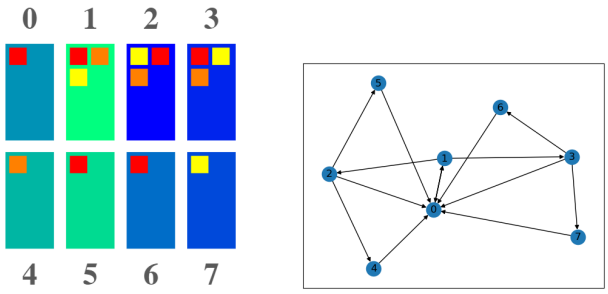

Figure 1. The left is an example of the synthetic screen images generated by the environment, while the right is a visualization of the underlying graph structure of the environment.

3.2. MDP

We use a deterministic dynamics model for our MDP, whereby clicking a given button leads to a specific page. In our implementation of the environment, whenever the environment is set-up, the button positions, colors and background colors are randomly chosen but are then fixed for the remainder of the episode.

In addition, we define our reward function to be the following:

$$R(s) = \begin{cases} 0 & s = s_t \\ -1 & \text{otherwise} \end{cases} \tag{1}$$

where s_t is the desired target node. Our reward function is defined as such in order to encourage the agent to find the shortest path to the target node. Note that we do not use discounting because this problem is formulated as a finite horizon MDP.

The biasing of the agent towards finding shortest paths is useful as this incentivizes the agent to potentially find paths to a certain destination on a screen which are better than human performance. For example, for some potential applications such as ordering a particular item on Amazon, there might be some "back-door" method to finding a path which is shorter than the path which a human might take.

3.3. Graphical Structure

For the graphical structure of the environment, we used two implementations: interleaving chains connected by a common root node which we refer to as "skeleton and chain" and a tree. Note that each node in the graph represents a state and each edge represents a particular action that can be taken.

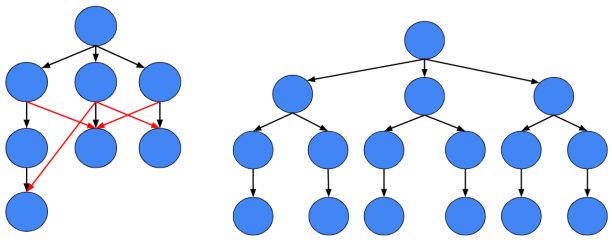

Figure 2. The left is an example of the skeleton-and-chaining method, while the right is an example of the tree method. Note that in the two images above, the directed edges from each node to the home node are not shown, but these edges are added in the implementation of the graph structure in the actual environment.

3.3.1. Skeleton and Chaining Method

In this formulation of the environment, we have a "home" node which acts as the root and is connected to all of the other chains. This part of the graph with just the root which is connected to all of the linked lists is referred to as the skeleton. One can think of this as a set of linked lists which are each connected by a common root node. However, since there are multiple paths to get to the same part of an app/website, we add "chaining" whereby we connect different linked lists together with an edge which can go to nodes at a different depth in a different linked list.

3.3.2. Tree Method

For the tree based environment, the "home" node is represented by the root of the tree. For each level in the tree, which we will refer to as a "tier", we define a per-tier branching factor hyperparameter which gives us flexibility in scal-

ing up the action space for each node.

3.4. Continuous and Discrete

As mentioned previously in the introduction, in implementing the custom environment, we came up with two separate formulations using a discrete action space in one case and a continuous action space in the other case.

With the discrete action space environment, we still used the actual synthetic app screen images as the states given to the agent, but instead, the action prompted from the agent was only to choose one of the finite number of buttons on the screen. This formulation is more consistent with ideas in having the RL agent being somewhere in the middle of the pipeline by interacting with a simiplified environment and then proceeding by making a prediction on the best action from there. This sort of pipeline could involve a segmentation model, for instance, to first simplify a given webpage or app screen, and this simplified image is then passed to an RL agent which can make further decisions on the best actions (buttons, in this case) to choose.

On the other hand, with the continuous action space environment, the state space is still represented with the synthetic app screen images, but instead, the action prompted from the agent is a pair of numbers (x, y) representing a coordinate on the app screen. In particular, this is representative of the agent choosing a certain position on the app to click. The main use for this formulation of the environment is for the agent to have more control on the actions it takes. As mentioned previously, LLM agents struggled with selecting coordinates on the webpage screen, and this is a place where we thought RL agents could be more effective. As such, we also created a second environment using these pixel coordinates on the screen as actions taken by the agent.

4. Approach

We tested a variety of off-the-shelf RL algorithms on our environment. In particular, we tested DQN and PPO on our discrete environment and PPO and DDPG on our continuous environment.

For DQN and DDPG we had to play around with the buffer size to get it to work, since the default parameters were set for a much more complicated environment. Rather than using a default buffer size of 1 million, we ended up using a buffer size of 1 thousand which resulted in faster and smoother training.

For PPO, we decided to normalize advantages for smoother training and we also needed to determine the appropriate target KL for optimal policy performance. From our brief experimentation, we found that a target KL of 0.01 resulted in significantly less noisy model training.

All other parameters were kept at their default values according to the Stable-Baselines3 library (Raffin et al., 2021).

5. Experimental Results

With the experiments we ran, we wanted to push the RL agents to their limits in the respective discrete and continuous environments, in order to see how the agents learned effective policies as the environment they were trying to navigate became more complex. In particular, we split up the experiments for the discrete and continuous environments into three separate tests. The first test involved scaling up the size of the images given to the agent as the respective state observation. The next two tests involved scaling up the complexity of the graph structure of the environment. We did this in two ways: by increasing the number of resulting branches at each node and by increasing the overall depth of the tree used as the base for the environment. In the later analysis of the experimental results, we will outline the importance of these two different increases in the complexity of the graphical structure.

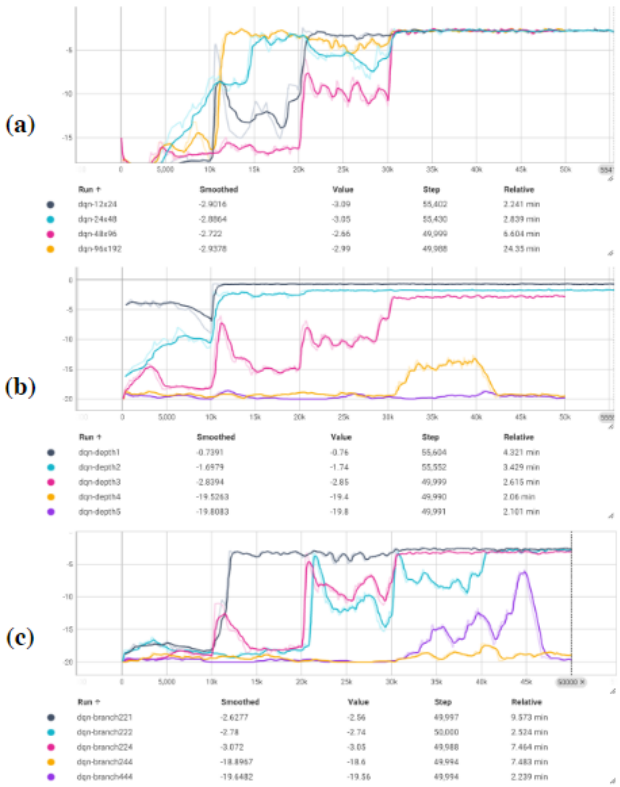

Figure 3. These graphs represent the results of training the DQN algorithm on the environment with the discrete action space.

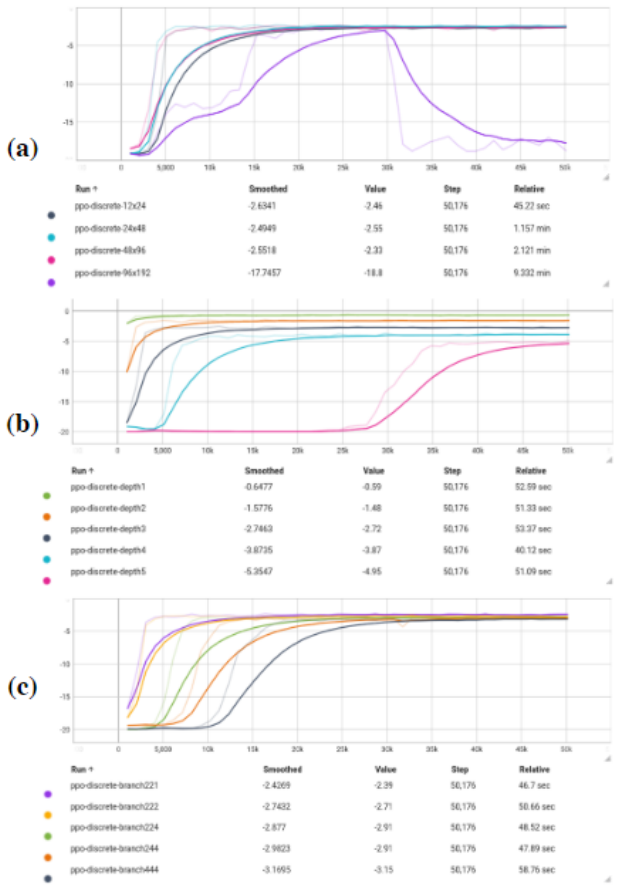

Figure 4. These graphs represent the results of training the PPO algorithm on the environment with the discrete action space.

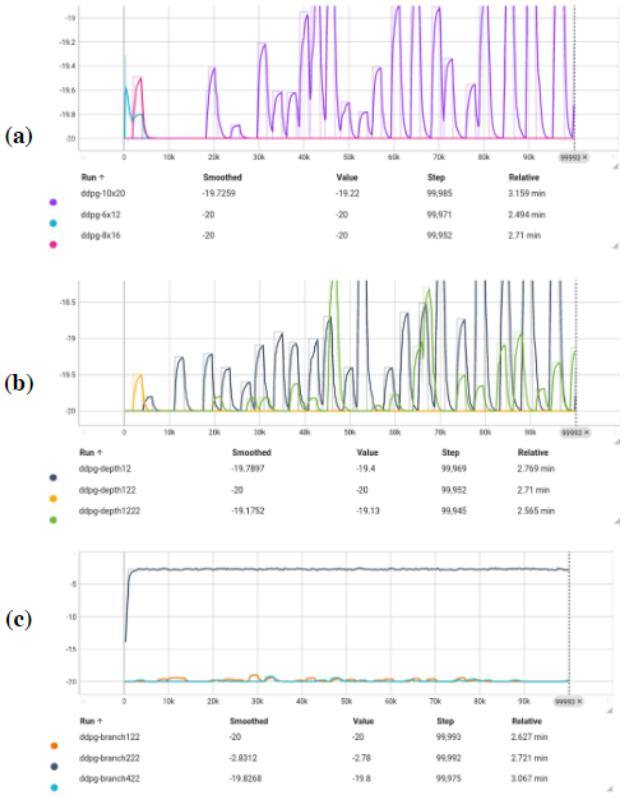

Figure 5. These graphs represent the results of training the DDPG algorithm on the environment with the continuous action space.

5.1. Image Tests

The first test we made was to fix the tree structure of the environment, but instead increase the size of the images given to the model with every separate run. The results from this first test is shown in the first row of Figures 3, 4, 5, and 6 respectively for the discrete environment with DQN and PPO and the continuous environments with DDPG and PPO respectively.

With the discrete environment, we first notice that scaling the size of the screen images had an indeterminate effect on the overall training and performance of the algorithms, especially for the DQN algorithm and arguably less so for the PPO algorithm. These results differed from our initial expectations, especially since we thought that increasing the size of the images directly increases the dimension of the observations which the agent receives, which then makes it more difficult for the agent to learn how to navigate the environment. However, one potential explanation we have for this is that the only aspect of the image which the agent has to learn is what specific "node" in the graph is associated with each image. As such, since the screen sizes (and the respective area taken up by the buttons) were scaled up equally, the overall effect on the agent's ability to learn how to navigate the environment wasn't as pronounced.

On the other hand, with the continuous environment, the results matched our expectations as increasing the size of the images also increases the space of actions which the agent has to take (specifically, the pixel coordinates on the screen). As a result, we suspect that since both the action and state space increase in complexity with each subsequent size increase of the screen images, the resulting agent has much more difficulty learning to navigate environments with larger images, which shows that the RL agents are much more affected by increases in the image size in the continuous environment.

5.2. Graph Depth Tests

As mentioned before, the second test we made was to fix the size of the screen images but to instead increase how deep the tree structure extended. We did this by fixing the branching constant to 2 at each tier in the tree, and instead

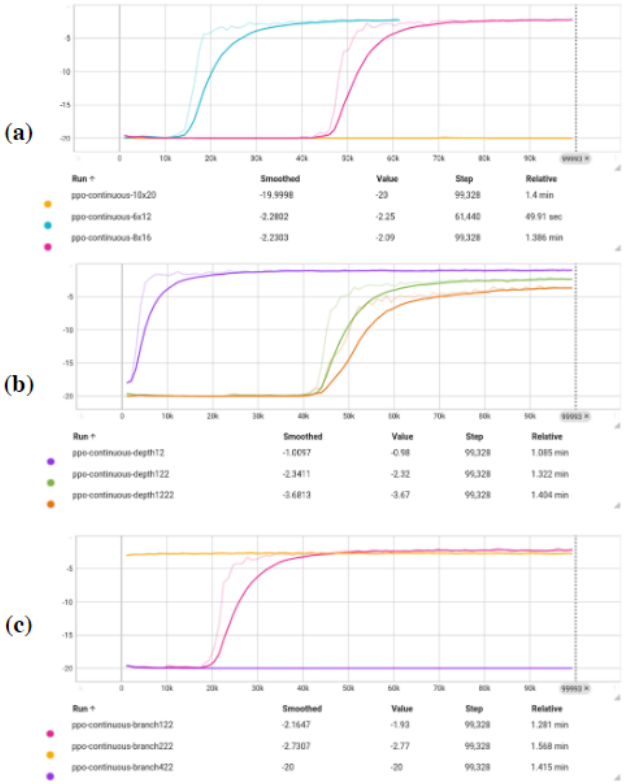

Figure 6. These graphs represent the results of training the PPO algorithm on the environment with the continuous action space.

to increase the depth of the tree by adding more tiers. In particular, this test measures the agent's ability to learn longer trajectories to get to the final target state.

With the discrete environment, we first notice that there was a clear drop in performance as the graph depth of the environment increased. In particular, we also saw that the final reward which the agent stayed at after learning an effective policy also decreased as the depth of the environment increased. This particular result is reasonable with the given formulation of the environment since the agent necessarily has to take a longer trajectory to reach the target state as the depth increases. We also saw both DQN and PPO required a higher number of training steps in order to reach the optimal policy, and this is also reasonable since the agent has to explore much more within the environment in order to reach the final target state.

In addition, with the continuous environment, we notice that the same trends in the results, but the magnitude of the changes by inreasing the depth were much more signficant in the continuous environment.

5.3. Graph Branching Tests

We also explored how training differs based on the branching factor while holding resolution and depth fixed. The main importance of this experiment was to test the agent's ability to learn to navigate an environment when there are more buttons on a screen to choose from. This is more applicable in cases where there a significant number of buttons on a screen, and the agent has to decide on the optimal one to choose, from this increased number of options.

With the discrete environment, we varied the branching factor at each tier. For high branching factors, the DQN agent sometimes achieves a good reward, but regress to a suboptimal policy due to instability in training. The PPO algorithm performed better overall. Compared to itself, it learned quicker in low branching factor settings, and vice versa. This is expected because the complexity of the environment is directly affected by how many next states are possible in each time step. We compared the speed of convergence between doubling the branching factor at the 2nd tier versus at the 3rd tier. Interestingly, when only a small number of episodes have passed, the agent that learned quicker (higher first-order derivative with respect to reward at the same episode count) is the one where the 2nd tier branching factor was doubled. As more episodes has passed, the agent that were in the enviroment where the 3rd tier branching factor was doubled quickly caught up. At approximately the 50,000 episode mark, all agents achieved similar levels of performance consistently. The DQN agent performed as expected as well, learning the quickest with the fewest-branching environment, and vice versa.

For the continuous environment, we similarly maintain the same depth while varying the branching at tiers 1, 2, and 3 respectively. For the PPO algorithm specifically, we saw that the resulting decrease in performance from increasing the overall branching in the environment was of a much higher magnitude than with the discrete environment. This went all the way to the extent that the PPO algorithm did not learn an effective policy when the branching on the first tier was double from 2 to 4.

5.4. General Comments About Training

One thing to note in our training is the results we received when using the DDPG algorithm on the continuous environment. As seen in the results shown previously, the DDPG algorithm did not learn an effective policy consistently as PPO demonstrated in many of the same cases, and this is likely due to that fact that we did not properly tune the parameters of DDPG well. This was also an issue, albeit to a lesser extent, which we had when using the DQN algorithm in the discrete environment. One common aspect of both of these algorithms is their reliance on a replay buffer which we may not have fine-tuned properly.

6. Conclusion

We show that DQN effectively learns this abstracted screen navigation environment, illustrating as proof of concept that an RL agent can explore a graph using only a vision-based observation. We also contribute a new environment which is more representative of the screen navigation task. Environments like Vokram which simulate aspects of screen navigation are only geared towards teaching an agent how to perform the very basic action of clicking on a button, whereas our environment allows an agent to learn the concept of screen navigation at the most fundamental level.

Another experiment we could perform is seeing how changing the reward model affects the agent's behavior. For example, we could try using a reward function of -2 if a non-button is clicked, -1 if the wrong button is clicked, and +1 if the correct button is clicked. As an additional layer of complexity, we can also add an action where the agent decides when to terminate the episode, which gives the agent more autonomy which is more representative of a real-world use case of screen navigation. We also hope to test other RL algorithms on this environment.

6.1. Future Work

With all of our work in creating a custom environment and training, there are various avenues we can take to extend on our current results. Additionally, we can make the environment and our formulation of the problem more complex. This will probably involve expanding the goals of the agent in navigating the environment and by making the increasing state space complexity through the introduction of additional noise in the buttons and screens.

The alternative option is to train RL algorithms on these more complex environments. Some low hanging fruit for additional tests include hyperparameter tuning on the current RL algorithms used, training with a vectorized environment for improved efficiency, and using different algorithms like Monte Carlo Tree Search, which are likely to be more performant in much larger environments.

We can also extend our environment using a curriculum learning approach whereby we decrease the level of abstraction of our environment, ultimately building up to an actual website/app. The idea behind this is that we can get an agent to eventually learn a good policy for the ultimate end task of screen navigation.

Acknowledgements

All team members contributed equally to the final project report and poster. Matthew worked on implementation of the environment and hyperparameter tuning. Sai worked on implementation and testing of the environment. Charles worked on testing the environment and hyperparameter tuning. We'd like to thank our project mentor, Joao Araujo, for his guidance throughout the course of our project and helping us narrow down the scope of our ideas. The code for this project can be found in this repository

References

Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. Openai gym. *CoRR*, abs/1606.01540, 2016. URL http://arxiv.org/abs/1606.01540.

Deng, X., Gu, Y., Zheng, B., Chen, S., Stevens, S., Wang, B., Sun, H., and Su, Y. Mind2web: Towards a generalist agent for the web, 2023.

Fan, L., Wang, G., Jiang, Y., Mandlekar, A., Yang, Y., Zhu, H., Tang, A., Huang, D.-A., Zhu, Y., and Anandkumar, A. MineDojo: Building Open-Ended Embodied Agents with Internet-Scale Knowledge, November 2022. URL http://arxiv.org/abs/2206.08853. arXiv:2206.08853 [cs].

Fu, J., Kumar, A., Nachum, O., Tucker, G., and Levine, S. D4rl: Datasets for deep data-driven reinforcement learning, 2020.

Furuta, H., Lee, K.-H., Nachum, O., Matsuo, Y., Faust, A., Gu, S. S., and Gur, I. Multimodal Web Navigation with Instruction-Finetuned Foundation Models, February 2024. URL http://arxiv.org/abs/2305.11854. arXiv:2305.11854 [cs, stat].

Gao, D., Ji, L., Zhou, L., Lin, K. Q., Chen, J., Fan, Z., and Shou, M. Z. AssistGPT: A General Multi-modal Assistant that can Plan, Execute, Inspect, and Learn, June 2023. URL http://arxiv.org/abs/2306.08640. arXiv:2306.08640 [cs].

Gao, D., Ji, L., Bai, Z., Ouyang, M., Li, P., Mao, D., Wu, Q., Zhang, W., Wang, P., Guo, X., Wang, H., Zhou, L., and Shou, M. Z. ASSISTGUI: Task-Oriented Desktop Graphical User Interface Automation, January 2024. URL http://arxiv.org/abs/2312.13108. arXiv:2312.13108 [cs].

Gur, I., Rueckert, U., Faust, A., and Hakkani-Tur, D. Learning to Navigate the Web, December 2018. URL http://arxiv.org/abs/1812.09195. arXiv:1812.09195 [cs, stat].

Gur, I., Jaques, N., Miao, Y., Choi, J., Tiwari, M., Lee, H., and Faust, A. Environment Generation for Zero-Shot Compositional Reinforcement Learning, January 2022. URL http://arxiv.org/abs/2201.08896. arXiv:2201.08896 [cs].

Hong, S., Zhuge, M., Chen, J., Zheng, X., Cheng, Y., Zhang, C., Wang, J., Wang, Z., Yau, S. K. S., Lin, Z., Zhou, L., Ran, C., Xiao, L., Wu, C., and Schmidhuber, J. Metagpt: Meta programming for a multi-agent collaborative framework, 2023a.

Hong, W., Wang, W., Lv, Q., Xu, J., Yu, W., Ji, J., Wang, Y., Wang, Z., Zhang, Y., Li, J., Xu, B., Dong, Y., Ding, M., and Tang, J. Cogagent: A visual language model for gui agents, 2023b.

Koh, J. Y., Lo, R., Jang, L., Duvvur, V., Lim, M. C., Huang, P.-Y., Neubig, G., Zhou, S., Salakhutdinov, R., and Fried, D. VisualWebArena: Evaluating Multimodal Agents on Realistic Visual Web Tasks, January 2024. URL http://arxiv.org/abs/2401.13649. arXiv:2401.13649 [cs].

Lee, S., Choi, J., Lee, J., Wasi, M. H., Choi, H., Ko, S. Y., Oh, S., and Shin, I. Explore, Select, Derive, and Recall: Augmenting LLM with Human-like Memory for Mobile Task Automation, March 2024. URL http://arxiv.org/abs/2312.03003. arXiv:2312.03003 [cs].

Liu, E. Z., Guu, K., Pasupat, P., Shi, T., and Liang, P. Reinforcement learning on web interfaces using workflow-guided exploration, 2018.

Liu, H., Li, C., Wu, Q., and Lee, Y. J. Visual Instruction Tuning, December 2023. URL http://arxiv.org/abs/2304.08485. arXiv:2304.08485 [cs].

Madaan, A., Zhou, S., Alon, U., Yang, Y., and Neubig, G. Language models of code are few-shot commonsense learners, 2022.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. A. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013. URL http://arxiv.org/abs/1312.5602.

Niu, R., Li, J., Wang, S., Fu, Y., Hu, X., Leng, X., Kong, H., Chang, Y., and Wang, Q. Screenagent: A vision language model-driven computer control agent, 2024.

Qian, C., Cong, X., Liu, W., Yang, C., Chen, W., Su, Y., Dang, Y., Li, J., Xu, J., Li, D., Liu, Z., and Sun, M. Communicative agents for software development, 2023a.

Qian, C., Dang, Y., Li, J., Liu, W., Chen, W., Yang, C., Liu, Z., and Sun, M. Experiential co-learning of software-developing agents, 2023b.

Qin, Y., Zhou, E., Liu, Q., Yin, Z., Sheng, L., Zhang, R., Qiao, Y., and Shao, J. Mp5: A multi-modal open-ended embodied system in minecraft via active perception, 2024.

Raffin, A., Hill, A., Gleave, A., Kanervisto, A., Ernestus, M., and Dormann, N. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8, 2021. URL http://jmlr.org/papers/v22/20-1364.html.

Shah, P., Fiser, M., Faust, A., Kew, J. C., and Hakkani-Tür, D. Follownet: Robot navigation by following natural language directions with deep reinforcement learning. *CoRR*, abs/1805.06150, 2018. URL http://arxiv.org/abs/1805.06150.

Sumers, T. R., Yao, S., Narasimhan, K., and Griffiths, T. L. Cognitive Architectures for Language Agents, March 2024. URL http://arxiv.org/abs/2309.02427. arXiv:2309.02427 [cs].

Toyama, D., Hamel, P., Gergely, A., Comanici, G., Glaese, A., Ahmed, Z., Jackson, T., Mourad, S., and Precup, D. AndroidEnv: A reinforcement learning platform for android. abs/2105.13231, 2021. URL http://arxiv.org/abs/2105.13231.

Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., and Anandkumar, A. Voyager: An Open-Ended Embodied Agent with Large Language Models, October 2023a. URL http://arxiv.org/abs/2305.16291. arXiv:2305.16291 [cs].

Wang, Z., Cai, S., Liu, A., Jin, Y., Hou, J., Zhang, B., Lin, H., He, Z., Zheng, Z., Yang, Y., Ma, X., and Liang, Y. JARVIS-1: Open-World Multi-task Agents with Memory-Augmented Multimodal Language Models, November 2023b. URL http://arxiv.org/abs/2311.05997. arXiv:2311.05997 [cs].

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Chi, E. H., Le, Q., and Zhou, D. Chain of thought prompting elicits reasoning in large language models. *CoRR*, abs/2201.11903, 2022. URL https://arxiv.org/abs/2201.11903.

Wu, Q., Bansal, G., Zhang, J., Wu, Y., Li, B., Zhu, E., Jiang, L., Zhang, X., Zhang, S., Liu, J., Awadallah, A. H., White, R. W., Burger, D., and Wang, C. Autogen: Enabling next-gen llm applications via multi-agent conversation, 2023.

Wu, Y., Fan, Y., Min, S. Y., Prabhumoye, S., McAleer, S., Bisk, Y., Salakhutdinov, R., Li, Y., and Mitchell, T. Agentkit: Flow engineering with graphs, not coding, 2024a.

Wu, Z., Han, C., Ding, Z., Weng, Z., Liu, Z., Yao, S., Yu, T., and Kong, L. Os-copilot: Towards generalist computer agents with self-improvement, 2024b.

Xie, J., Chen, Z., Zhang, R., Wan, X., and Li, G. Large multimodal agents: A survey, 2024a.

Xie, T., Zhou, F., Cheng, Z., Shi, P., Weng, L., Liu, Y., Hua, T. J., Zhao, J., Liu, Q., Liu, C., Liu, L. Z., Xu, Y., Su, H., Shin, D., Xiong, C., and Yu, T. Openagents: An open platform for language agents in the wild, 2023.

Xie, T., Zhang, D., Chen, J., Li, X., Zhao, S., Cao, R., Hua, T. J., Cheng, Z., Shin, D., Lei, F., Liu, Y., Xu, Y., Zhou, S., Savarese, S., Xiong, C., Zhong, V., and Yu, T. Osworld: Benchmarking multimodal agents for open-ended tasks in real computer environments, 2024b.

Yang, Z., Chen, G., Li, X., Wang, W., and Yang, Y. DoraemonGPT: Toward Understanding Dynamic Scenes with Large Language Models, February 2024. URL http://arxiv.org/abs/2401.08392. arXiv:2401.08392 [cs].

Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T. L., Cao, Y., and Narasimhan, K. Tree of thoughts: Deliberate problem solving with large language models, 2023a.

Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. React: Synergizing reasoning and acting in language models, 2023b.