AndroidEnv: A Reinforcement Learning Platform for Android

Daniel Toyama[*,1], **Philippe Hamel**[*,1], **Anita Gergely**[*,1], **Gheorghe Comanici**[*,1], **Amelia Glaese**[1], **Zafarali Ahmed**[1], **Tyler Jackson**[1], **Shibl Mourad**[1] **and Doina Precup**[1]

[*]Equal contributions, [1]DeepMind

We introduce AndroidEnv, an open-source platform for Reinforcement Learning (RL) research built on top of the Android ecosystem. AndroidEnv allows RL agents to interact with a wide variety of apps and services commonly used by humans through a universal touchscreen interface. Since agents train on a realistic simulation of an Android device, they have the potential to be deployed on real devices. In this report, we give an overview of the environment, highlighting the significant features it provides for research, and we present an empirical evaluation of some popular reinforcement learning agents on a set of tasks built on this platform.

1. Introduction

Reinforcement learning (RL) is a branch of artificial intelligence (AI) which studies computational models of learning from interaction with an environment and from numerical rewards (Sutton and Barto, 2018). RL methods have demonstrated success not only in game playing, for example checkers (Schaeffer et al., 1992), chess (Campbell et al., 2002; Silver et al., 2018), Go (Silver et al., 2016), poker (Moravčík et al., 2017), Atari (Mnih et al., 2015) and Starcraft II (Vinyals et al., 2019), but also in real-world applications, such as robotics (Kormushev et al., 2013), logistics (Refanidis et al., 2001), chemical synthesis (Segler et al., 2018) and personalised recommendations (Liu et al., 2019). In many of these applications, RL agents were able to achieve super-human performance, yet they can be prone to over-specialising to any single domain. In order to assess the performance of RL algorithms over a range of different tasks, it is desirable to have platforms which expose diverse challenges through a unified interface. This approach was pioneered in the original Atari suite (Bellemare et al., 2013) and has been followed up by a variety of platforms, such as DeepMind Lab (Beattie et al., 2016), OpenAI Universe (OpenAI, 2016) and World of Bits (Liu et al., 2018). To complement these existing platforms, we present *AndroidEnv*, a research platform built on top of the Android Operating System (OS). The open-source library, along with detailed technical documentation and a set of tasks are available on GitHub.[1]

AndroidEnv has a universal touchscreen interface that enables the empirical evaluation of general purpose RL algorithms designed to tackle a wide variety of tasks. The agent-environment interaction in AndroidEnv matches that of a user and a real device: the screen pixels constitute the observations, the action space is defined by touchscreen gestures, the interaction is real-time, and actions are executed asynchronously, while the environment runs at its own time scale. With these features, agent performance can be realistically compared to humans. Moreover, environments that behave as closely as possible to their real-world counterparts also facilitate production deployment, without added work to adapt to different interfaces or data distributions.

We chose Android as the underlying system because it is a popular, open-source operating system with over two billion monthly active users and a selection of over two million applications. The sheer number of applications, built for a multitude of important aspects of human life, ranging from education and business to communication and entertainment, provides virtually unlimited challenges for RL research.

[1] https://github.com/deepmind/android_env

Furthermore, externally written apps ground the research in real problems, avoiding common pitfalls of systems tailored for specific research agendas.

This technical report is structured as follows: Section 2 provides an overview of the notable features of AndroidEnv. Section 3 describes what defines a *Task*, and presents a set of tasks included in the release. Section 4 provides some initial empirical results of popular RL agents on a selection of AndroidEnv tasks. Section 5 provides some technical details worth considering when using the AndroidEnv platform. Lastly, Section 6 discusses some existing RL research platforms and highlights their relevance to AndroidEnv.

2. Environment Features

AndroidEnv enables RL agents to interact with, and learn to solve tasks on any Android application, including the operating system itself. In particular, AndroidEnv implements the dm_env API (Muldal et al., 2019) on top of an emulated Android device. Virtual, emulated Android devices allow the dynamics of the environment to be entirely generated by the OS itself. In the rest of this section, we expand on the most important distinguishing features of the environment.

2.1. Real-time execution

The Android OS, whether it is running as an emulator or on a real device, runs in real-time, independently of the agent interacting with it. All observations and actions are asynchronous and OS does not pause when providing observations or when accepting actions. Users can control the rates for fetching observations and for sending actions, but they cannot speed up or slow down the OS. As such, AndroidEnv is unable to run in lock-step, and agents may need to handle a non-negligible amount of delay between consecutive action executions. Furthermore, the screen refresh rate varies between 60Hz and 120Hz and capturing the screen beyond that limit does not provide the agent with more information. Android and its specific apps are in control of processing and interpreting agent actions, and the platform allows buffering up to a device and version-dependent limit. However, sending a high number of actions at a time does not give the agent more control over the simulation. These characteristics make AndroidEnv a more naturalistic platforms for developing and testing RL algorithms.

2.2. Action interface

Raw action space. The native action space of the environment consists of a tuple consisting of a position $(x, y) \in [0, 1] \times [0, 1]$, determining the location of the action on the screen, and a discrete value `ActionType` $\in$ {`TOUCH`, `LIFT`, `REPEAT`} indicating whether the agent opts for touching the screen at the indicated location, lifting the pointer from the screen, or repeating the last chosen action, respectively. This action space is the same across all tasks and apps.

It is worth noting that while two actions a_1 = {`ActionType = LIFT, position =` (x_1, y_1)} and a_2 = {`ActionType = LIFT, position =` (x_2, y_2)} are different from the agent's perspective, in practice they result in the same effect on the device, because the *lack* of a touch has no association to a particular location.

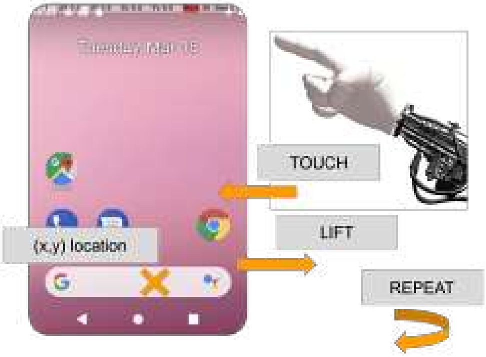
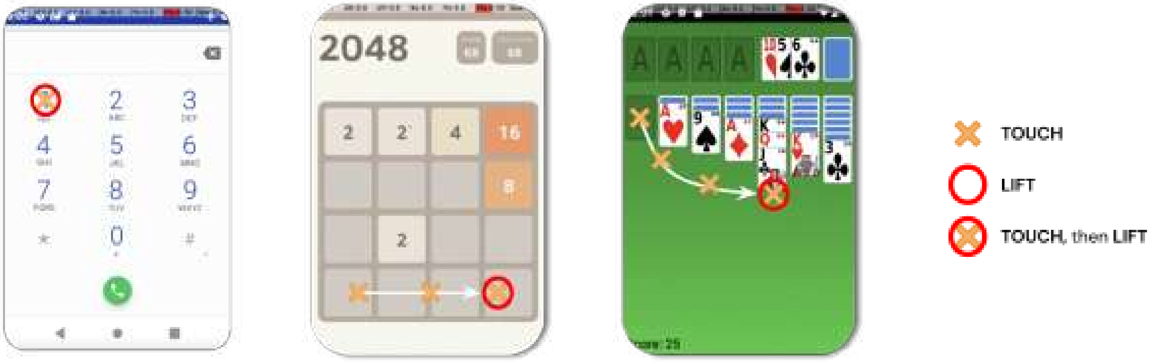


Figure 1 | The action space is composed of a discrete action type and a screen location.

Gestures. The complexity of the interface arises from the fact that individual raw actions on their own do not necessarily trigger a meaningful change in the environment. It is more useful for agents to control Android applications via *gestures*, such as pressing, long pressing, swiping, scrolling, or drag-and-drop. Each of these correspond to a particular sequence of raw actions: for example, a screen *touch* at a particular location, followed by a *lift* of the the imaginary finger is a sequence that Android can interpret as a *press of a button*. Similarly, Android will interpret a sequence of aligned *touches* as *scrolling*.



(a) Tapping (b) Swiping (c) Drag-and-drop

Figure 2 | Examples of gestures. Actions are performed one after the other, tracing out a particular path.

This distinction between the *raw* action space and a particular app *interface* makes AndroidEnv a challenging domain. A random sequence of actions will typically have a small probability of producing a meaningful gesture in most Android apps. This need to compose actions, paired with the difficulty of solving the underlying task itself, leads to a difficult exploration problem. For example, in order to learn to play chess in AndroidEnv, an agent must not only find a winning strategy, it also has to learn to move pieces through drag-and-drop gestures.

Relation to observations. Another notable feature of AndroidEnv is the spatial correlation between actions and observations. Often, an action can result in local changes in the pixels near the location of the action, or the position of certain items in the observation might hint at the next best location to take an action. In particular, the screen is often suggestive of the kind of *gestures* the application expects: smartphone users would often find it intuitive to *tap* where they see an item in the shape of a button, or to *scroll* where they see a drop-down menu.

Altering the action space. AndroidEnv allows users to define wrappers around the raw action space of the environment. For example, one might discretise the action space by splitting up the screen into a grid, restrict the `ActionType` to `TOUCH`, or group action sequences like [`LIFT`, `TOUCH`, `LIFT`] into a single *tap* action. We provide some useful and natural wrappers (see Section 5). Note that these wrappers but alter the set of actions available to the agent, but not the way in which AndroidEnv interprets raw actions.

2.3. Observations

Observation space. The observation space of AndroidEnv consists of three main components: {`pixels`, `timedelta`, `orientation`}. The most notable component is `pixels`, representing the current frame as an RGB image array. Its dimensions will depend on the device used (real or virtual), but given that it will correspond to real device screen sizes, this array will typically be large (of course, users can scale down their dimensionality, e.g. with wrappers). The `timedelta` component captures the amount of time passed since AndroidEnv fetched the last observation. The `orientation`, even though it does not affect the layout of the RGB image in the observation, might carry relevant information for the agent. For example, if there is text on the screen, its orientation is useful for automatic processing. As mentioned above, observations often carry spatial cues and are suggestive of meaningful gestures to perform in a given state. The fact that the observation space is the same across all tasks is what makes it useful for agents, and creates the opportunity to generalize across tasks.

Task extras. In addition to default observations, ({`pixels`, `timedelta`, `orientation`}), some tasks might expose structured information after each step (see Sec. 3). An *extra* in AndroidEnv is any information that the environment sends to aid the understanding of the task. The information sent through this channel is typically very useful for learning, yet difficult to extract from raw pixels. For example, extras may include signals indicating events such as a button press or opening of a menu, text displayed on the screen in string format, or a simple numerical representations of the displayed state. Note that extras are a standard mechanism for communicating information used in Android apps.

We note that, unlike the observation and raw action space, which are the same across all AndroidEnv, task extras are specific to individual tasks, are entirely optional, and may not be available at all. Furthermore, task extras, even if provided, are not part of the default observation; rather AndroidEnv returns them upon explicit request (see detailed documentation).

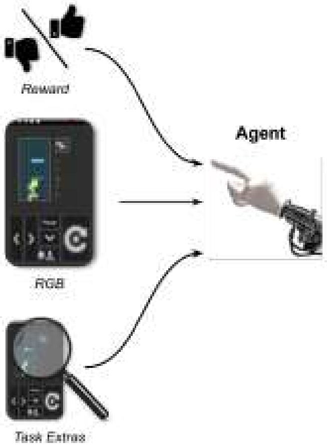
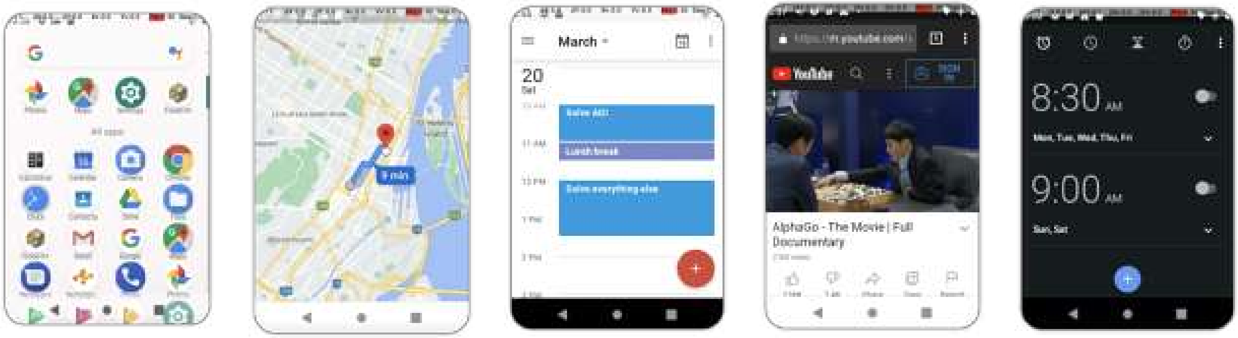


Figure 3 | Information available to the agent.

3. Tasks

While Android is an operating system with no inherent rewards or episodes, AndroidEnv provides a simple mechanism for defining *tasks* on it. Tasks capture information such as episode termination conditions, rewards, or the apps with which the agent can interact. Together, these define a specific RL problem for the agent.



(a) Android menu (b) Google Maps (c) Calendar (d) Chrome (e) Clock

Figure 4 | Examples of Android OS apps and use cases.

Task structure. We capture aspects that make up a task definition in a *Task* protocol buffer message. These include information on:

- How to initialise the environment: for example, installing particular applications on the device.
- When should an episode be reset: for example, upon receiving a particular message from the device or app, or upon reaching a certain time limit.
- Events triggered upon an episode reset: for example, launching a given app, clearing the cache, or pinning the screen to a single app (hence restricting the agent's interaction to that app).
- How to determine the reward: for example, this might depend on different signals coming from Android, such as Android accessibility service or log messages implemented in applications.

With these protocol buffer messages, users can define a wide variety of tasks on Android. For example, a task could be to set an alarm in the Android standard Clock app, by opening this app upon launch, and rewarding the agent and ending an episode once an alarm has been set. We detail the full specification of the protocol buffer message structure in the code repository.

Available tasks. Along with the AndroidEnv platform implementation, we provide an initial set of ready-to-use tasks. At the time of the release, this includes over 100 tasks across roughly 30 different apps, ranging from basic tasks with straightforward objectives, to more sophisticated tasks that require long-term reasoning. The selection contains time-sensitive tasks (e.g. `catch`), physics-based environments (e.g. `vector_pinball`), puzzles (e.g. `classic_2048`), card games (e.g. `simple_solitaire`), spatial reasoning (e.g. `perfection`), UI navigation (e.g. `clock_set_timer`), strategy games (e.g. `droidfish`) and more. Note that several of these tasks are defined around the same app by varying parameters such as the game level, the reward signal or the difficulty. We emphasize that this set serves as a starting point and not as a definitive benchmark. Users can define their own tasks. We refer the reader to the code repository for instructions on creating additional tasks, as well as for an up-to-date list of available tasks.

4. Experimental results

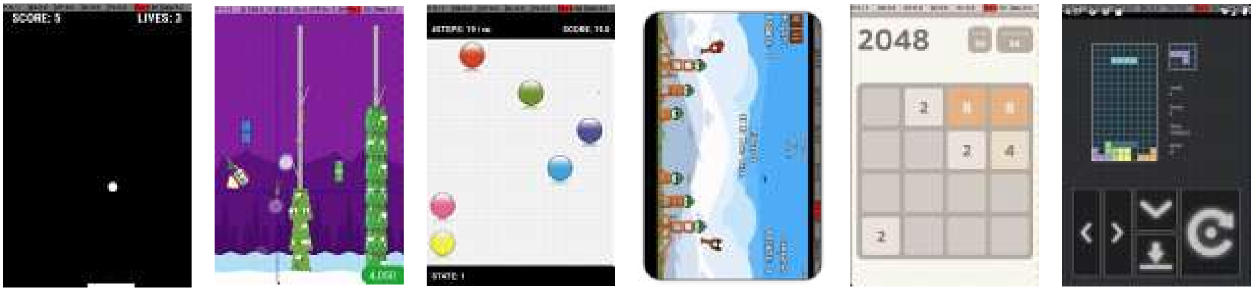
In this section, we present some empirical results for a selection of baseline RL agents on a small subset of tasks. For our experiments, we used the Acme framework (Hoffman et al., 2020) and its TensorFlow (Abadi et al., 2015) agents available at Acme's Github Repository.[2]

Since the action interface in AndroidEnv is a hybrid of discrete and continuous components, we defined some Wrappers (described below) for ease of experimentation. The continuous control agents we ran are Deep Deterministic Policy Gradient (DDPG) (Lillicrap et al., 2016), its distributional version (D4PG) (Barth-Maron et al., 2018), and Maximum a Posteriori Policy Optimisation (MPO) (Abdolmaleki et al., 2018). All these agents interact with a wrapped version of the environment for which they have to provide an `ActionType` as a continuous value in the interval $[0, 1]$. AndroidEnv rounds this number to the nearest integer and forwards the corresponding discrete `ActionType` to the simulator.

We also tested the following agents designed for finite action interfaces: DQN (Mnih et al., 2015), IMPALA (Espeholt et al., 2018), and R2D2 (Kapturowski et al., 2019). In this case, we discretised the screen as a 6×9 grid, resulting in 108 possible actions, corresponding to a choice of `ActionType` among (`LIFT`, `TOUCH`) combined with any of the 54 cells in the grid. To help memoryless agents, we augmented the current observation with a one-hot encoding of the location of the last taken action, which provides a more informative input for learning.

For our experiments, we chose the following tasks: `catch`, `rocket sleigh`, `press button`, `apple flinger`, `2048`, `blockinger`. They were selected to be representative of the variety of apps, difficulties, and action interfaces available across Android. This variety is reflected in the experimental results,

[2] https://github.com/deepmind/acme/tree/master/acme/agents/tf



(a) Catch (b) Rocket Sleigh (c) Press Button (d) Apple Flinger (e) 2048 (f) Blockinger

Figure 5 | Small selection of tasks used in the experiments.

showing that the same agents can have drastically different performance depending on each of these factors. For example, most agents perform well on tasks such as `catch` that have a simple action interface and dense rewards, whereas the combination of a highly structured interface, time sensitivity and sparse rewards render `blockinger` particularly difficult to solve.

Since none of these tasks require high-resolution inputs to achieve optimal behavior, we down-sampled the image observation to 80 × 120 pixels. Since this size is comparable to the resolution commonly used in the ATARI Learning Environment, we were able to run all agents using the network architectures reported by the authors of each corresponding agent. We generated training data using 128 distributed actors and we compiled results for each hyper-parameter configuration by averaging the performance of 4 independent runs using different seeds. See Figure 6 for an overview of the results of these experiments.

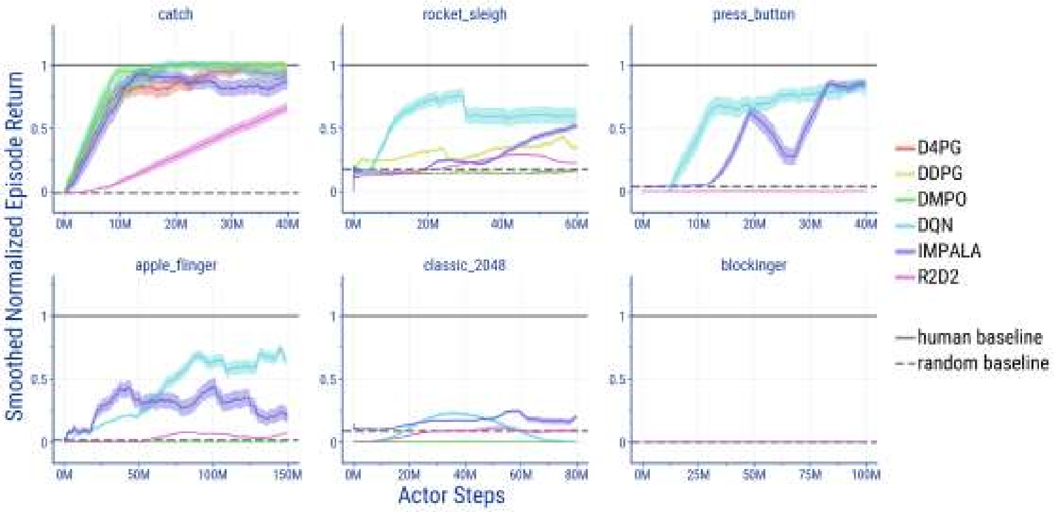


Figure 6 | **Agent performance**: The baseline continuous and discrete control agents ran on selection of AndroidEnv tasks, covering games where the action interface requires interactions including localised touches (`catch`), swiping (`classic_2048`), and drag-and-drop (`apple_flinger`). Continuous control agents perform well only in tasks where the interface does not expect complex gestures, but fail to achieve reasonable performance otherwise. Discrete control agents display better overall performance. We compiled the results above by averaging human-normalized scores (with `1.0` corresponding to average human performance) over four different seeds for each agent configuration. Note the clear difference in task difficulty, highlighted by the performance of baseline agents, with `catch` being solved by almost all agents, while no agents can generate useful behavior on `blockinger`.

5. Technical Details

ADB-based communication. *Android Debug Bridge* (ADB) provides a way of communicating with an Android device, be it physical or virtual. It exposes a shell that allows users to send commands to the device. AndroidEnv uses ADB for control operations, such as launching an app, querying the current activity, resetting episodes and listening for task extras coming from the app.

Simulator. AndroidEnv uses the Android Emulator[3], which is provided with Android Studio as its default simulator. In order to run the emulator, users need to specify an *Android Virtual Device* (AVD). In particular, one can use Android Studio to create AVDs with a specific screen resolution and OS version. Thus, users can choose the device type used for RL simulations. In principle, they can also extend AndroidEnv to work with other simulators. Simulations also provide a safe environment for RL agents to learn and make mistakes without any real world impact.

Real-time interaction. Because AndroidEnv is a real-time platform, some timings will be inherently unpredictable. Depending on the machine and the simulator, there is a rate limit at which AndroidEnv fetches observations from the OS, which depends on the resolution of the device, the performance of the machine, and whether the rendering is done through software or hardware.

Another important factor to consider in real-time environments is that agents require some deliberation time to generate the next action, given an observation. In traditional lockstep environments, the environment generates an observation and pauses to wait until the agent responds with an action, before stepping the simulation forward, as illustrated in Figure 7. Thus, in that setting, the actor deliberation time has no consequence on the agent-environment interaction. In a real-time setting, the environment does not pause to wait for the agent's action, as seen in Fig. 7, so large deliberation times can be harmful to performance. We view this as an interesting challenge that RL agents need to tackle, and which is not present in most other simulation platforms.

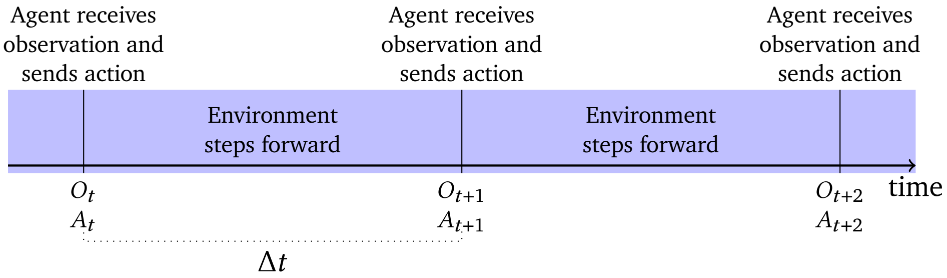

Agent receives observation and sends action
Agent receives observation and sends action
Agent receives observation and sends action
Environment steps forward
Environment steps forward
O_t
A_t
O_{t+1}
A_{t+1}
O_{t+2}
A_{t+2}
time
Δt

Figure 7 | Timeline of lockstep interaction between an environment and an agent. After sending an observation, the environment waits for the agent's action before stepping the simulation time forward.

We note that a step time with high variance could cause unpredictable interactions with the device. For instance, an unexpectedly long agent deliberation time could turn an intended *tap* gesture into a *long press*. To prevent these issues, AndroidEnv can optionally insert a wait time before requesting observations, in order to be closer to a fixed rate of interaction, while still providing the agent with the most recent observation possible. Figure 8 shows how the agent-environment cycle unfolds in time. Given a desired `max_steps_per_second`, AndroidEnv waits $\Delta t = 1/$`max_steps_per_second`, in order to come as close as possible to the desired interaction rate. The optional wait has a stabilizing effect on the time Δt between consecutive observations when the variance in the agent deliberation and/or

[3] https://developer.android.com/studio/run/emulator

rendering time is large. A well-chosen step rate can also extend the effect of a particular action, hence regularizing the corresponding training data.

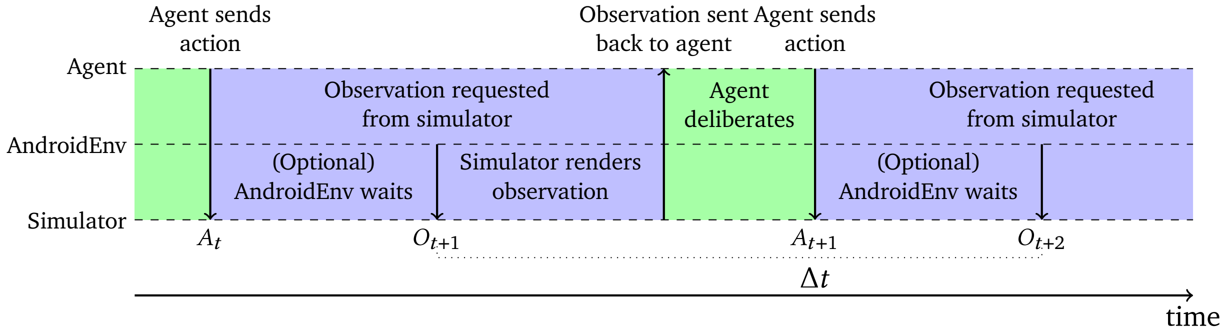

Agent sends action
Observation sent back to agent
Agent sends action
Agent
AndroidEnv
Simulator
Observation requested from simulator
Agent deliberates
Observation requested from simulator
(Optional) AndroidEnv waits
Simulator renders observation
(Optional) AndroidEnv waits
A_t
O_{t+1}
A_{t+1}
O_{t+2}
Δt
time

Figure 8 | Timeline of the real-time interaction between an agent and AndroidEnv.

Wrappers. We also provide environment wrappers to help users customise their experiments. They allow modifying the observation space (e.g. `ImageRescale` wrapper to resize pixel observations), the action space (e.g. `DiscreteAction` wrapper to discretise the hybrid action space), or the interface (e.g. `GymWrapper` for agents expecting an OpenAI Gym interface).

6. Relevant Work: Other RL Research Platforms

We designed AndroidEnv to complement existing platforms, by leveraging Android's rich ecosystem. In this section, we give an overview of some of these alternatives and highlight their features in relation to AndroidEnv. A summary of the features of these different platforms is given in Table 1.

Atari2600. (Bellemare et al., 2013) This test bed was the first platform allowing an RL agent to interact with various tasks through the same observation-action interface. It allowed building agents that use a *single* neural network architecture over a suite of 57 games. It has since been used as a core deep reinforcement learning research platform, leading to significant advances in algorithm design. Some of its characteristics include: a relatively small action space (18 discrete actions), operating in lock-step (i.e. the underlying simulation waits for the agent to act), and a diverse set of tasks that test core agent capabilities such as exploration, credit assignment, and generalisation. Still, the platform has limited flexibility: fetching rewards from games required environment developers to access privileged memory, the games are deterministic, the platform itself does not provide auxiliary signals to aid learning, and designing new tasks, although possible, is not easy. This is an important drawback, as the platform could be quite limiting when testing algorithms for scalability, large or continuous action spaces, stochastic or real-time environments, complex memory capabilities or language skills (Machado et al., 2017). AndroidEnv and OpenAI universe, which we discuss below, are alternatives that address of these limitations. In fact, OpenAI Universe includes Atari 2600 games, hence making the platform available for testing more complex action interfaces and real time interaction.

DeepMind Lab. (Beattie et al., 2016) Deepmind Lab is a 3D environment that provides a suite of challenging navigation and puzzle-solving tasks for learning agents. The observation consists of a first person pixel-based view of the 3D world, along with depth and velocity information. Users can customise the resolution of the observatios, which are rendered by a GPU or by a CPU. The action interface consists of multiple simultaneous actions to control movement (translation, rotation, jump/crouch). The suite includes several task types such as resource collection, navigation and laser tagging. Although researchers can easily extend the task suite with Deepmind Lab tools for level creation, the tasks are all within this consistent 3D world. AndroidEnv tasks are not restricted to a specific world simulation, as tasks can be defined on any app or service running within the OS.

Minecraft. Johnson et al. (2016) Minecraft is one of the most popular video games and an RL domain has been constructed on top of it, which raises important research challenges, due to the need for active perception and lifelong learning solutions. In this game, the agents' success depends on their ability to navigate, build structures, interact with objects, collect resources, and avoid obstacles and other attacking entities (e.g. zombies) (Mojang, 2014). The platform provides a set of tools that facilitate in-game design to study and develop algorithmic solutions for specific cognitive faculties (Tessler et al., 2017). For example, recent work demonstrated that Minecraft can be a useful platform for research related to robotics, with strong support for an experimental setup based on the Object Oriented Markov Decision Process (OO-MDP) paradigm (Aluru et al., 2015).

Despite the fact that Minecraft is an open-world game with complex game-play elements that require long-term credit assignment, RL research on Minecraft to date has been rather limited, with a strong focus on toy tasks with short horizons, restricted navigation, movement restricted to 2D, or interaction limited to a small set of objects (Bonanno et al., 2016; Oh et al., 2016; Tessler et al., 2017). Other methods leverage prior human demonstration or prior knowledge (Abel et al., 2015, 2016; Frazier and Riedl, 2019; Guss et al., 2019; Shu et al., 2017). Moreover, the tasks are commonly designed to allow agents to act by using images downsampled to 84 x 84 pixels as input, similar to the Atari Learning Environment. The agent is also limited to choosing from a small set of actions (e.g. 6 to 8 actions for navigation, pickup, breaking, placing, etc.) corresponding to low-level actuators that interact with the emulator.

Robotics/dm_control (Tassa et al., 2020). Practitioners commonly use physical robots or run computer-based simulations for RL research on robotics. Physical devices provide the highest possible fidelity to real world problems, but they are generally costly, slow, non-deterministic and inflexible. Computer-based simulations cannot match their physical counterparts in fidelity (i.e. there is always a simulation gap), but they can scale to thousands or even millions of instances at a fraction of the cost. This is important for RL research due, because RL algorithms can be data inefficient. Moreover, defining rewards that match the expectations of designers can be particularly challenging in robotics. The most common approach to overcome both of these challenges is to rely on human demonstrations.

MuJoCo (Todorov et al., 2012) is a widely used simulator in RL research, and the basis of dm_control, a suite of various robotics-like tasks. Its observations and actions are sets of continuous multidimensional vectors, and they vary according to different body types (e.g. humanoid, quadruped, half-cheetah etc). Users can conveniently pause and resume the simulation of the environment at will. Moreover, tasks are easily modifiable by customising XML task descriptions, and they can be easily inspected by using physical interactivity tools provided by the engine.

OpenAI Universe. (OpenAI, 2016) The Universe platform, released in 2016, has the same broad goals and motivation as AndroidEnv. Both platforms expose similar universal visual interfaces, i.e. pixels for observations. Universe provides keyboard and mouse gestures for actions. Moreover, both platform allow for the easy design and addition of a wide variety of tasks, and the incorporation of auxiliary structured information.However, Universe predominantly specifies the reward function through a convolutional neural network that extracts numbers from images, while AndroidEnv has access to app logs and system events to compute rewards.

Universe was in many ways ahead of its time. State of the art RL agents at the time of its release were not even close to addressing all the challenges that the environment offered. Universe included Atari games in its task suite, yet no agent could adequately play them using the Universe interface, i.e. mouse and keyboard gestures and large observations. To demonstrate learning, the authors discretised the action interface and specialised it to select only among a fixed number of keyboard keys that would fully control the Atari Suite. As shown in the empirical results, AndroidEnv presents a variety of tasks, some which are definitely within reach for current RL agents, and some which are quite challenging, therefore providing an interesting runway for novel RL agents.

World of Bits (WoB) (Shi et al., 2017). WoB is an RL environment based on OpenAI Universe, with tasks defined on real or cached web pages from the internet. The observation contains pixels and the Document Object Model (DOM) of the current page, along with useful annotations such as bounding boxes of DOM elements. Much like Universe, keyboard and mouse events determine the action space, inheriting its universal applicability. Users can handcraft WoB tasks or collect them via crowd-sourcing Question-Answer interactions. In particular, WoB and MiniWob++ (Liu et al., 2018) include a variety of tasks that expose user interface challenges for RL agents based on similar interactions with a single Android application.

Table 1 | Summary of environment properties

Environment	Universal Interface	Extensible Task Suite	Real-time	Continuous Action Space
Atari	✗			
DM Lab	✗	✗		✗
DM Control Suite		✗		✗
Minecraft		✗		✗
OpenAI Universe	✗	✗	✗	✗
World of Bits	✗	✗	✗	✗
AndroidEnv	✗	✗	✗	✗

7. Conclusion

We described AndroidEnv, an AI platform based on the Android Operating System, which provides tasks based on its large app ecosystem. The environment's universal observation and action space, along with real-time simulation, make it a particularly interesting challenge for current state-of-the-art agents. AndroidEnv is a suitable environment for studying a wide range of RL research problems such as exploration, hierarchical RL, transfer learning, or continual learning. We hope that it will provide a useful complement to the existing set of research platforms. Since Android has billions of users, and AndroidEnv provides tasks that run on the standard Android OS simulator, agents trained on the platform could potentially tackle a wide range of use cases leading to direct, real-world impact. For example, the ability to automatically learn sequences of actions might lead to advanced hands-free voice navigation tools; on-device AI models could help provide a better user experience; and trained agents could assist in device testing and quality assurance by benchmarking new apps, measuring latency, or detecting crashes or unintended behaviours in the Android OS.

References

M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL https://www.tensorflow.org/. Software available from tensorflow.org.

A. Abdolmaleki, J. T. Springenberg, Y. Tassa, R. Munos, N. Heess, and M. A. Riedmiller. Maximum a posteriori policy optimisation. *CoRR*, abs/1806.06920, 2018.

D. Abel, D. Hershkowitz, G. Barth-Maron, S. Brawner, K. O'Farrell, J. MacGlashan, and S. Tellex. Goal-based action priors. In *ICAPS*, 2015.

D. Abel, A. Agarwal, F. Diaz, A. Krishnamurthy, and R. E. Schapire. Exploratory gradient boosting for reinforcement learning in complex domains. *CoRR*, abs/1603.04119, 2016.

K. C. Aluru, S. Tellex, J. Oberlin, and J. MacGlashan. Minecraft as an experimental world for AI in robotics. In *AAAI Fall Symposia*, 2015.

G. Barth-Maron, M. W. Hoffman, D. Budden, W. Dabney, D. Horgan, D. TB, A. Muldal, N. Heess, and T. P. Lillicrap. Distributed distributional deterministic policy gradients. *CoRR*, abs/1804.08617, 2018.

C. Beattie, J. Z. Leibo, D. Teplyashin, T. Ward, M. Wainwright, H. Küttler, A. Lefrancq, S. Green, V. Valdés, A. Sadik, J. Schrittwieser, K. Anderson, S. York, M. Cant, A. Cain, A. Bolton, S. Gaffney, H. King, D. Hassabis, S. Legg, and S. Petersen. Deepmind lab. *CoRR*, abs/1612.03801, 2016.

M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The Arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, Jun 2013. ISSN 1076-9757. doi: 10.1613/jair.3912.

D. Bonanno, M. Roberts, L. Smith, and D. Aha. Selecting subgoals using deep learning in minecraft : A preliminary report. 2016.

M. Campbell, A. Hoane, and F. hsiung Hsu. Deep blue. *Artificial Intelligence*, 134(1):57–83, 2002. ISSN 0004-3702. doi: https://doi.org/10.1016/S0004-3702(01)00129-1.

L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning, S. Legg, and K. Kavukcuoglu. Impala: Scalable distributed deep-RL with importance weighted actor-learner architectures, 2018. cite arxiv:1802.01561.

S. Frazier and M. Riedl. Improving deep reinforcement learning in Minecraft with action advice. *CoRR*, abs/1908.01007, 2019.

W. H. Guss, B. Houghton, N. Topin, P. Wang, C. Codel, M. Veloso, and R. Salakhutdinov. MineRL: A large-scale dataset of Minecraft demonstrations, 2019.

M. Hoffman, B. Shahriari, J. Aslanides, G. Barth-Maron, F. Behbahani, T. Norman, A. Abdolmaleki, A. Cassirer, F. Yang, K. Baumli, S. Henderson, A. Novikov, S. G. Colmenarejo, S. Cabi, C. Gulcehre, T. L. Paine, A. Cowie, Z. Wang, B. Piot, and N. de Freitas. Acme: A research framework for distributed reinforcement learning. *arXiv preprint arXiv:2006.00979*, 2020. URL https://arxiv.org/abs/2006.00979.

M. Johnson, K. Hofmann, T. Hutton, and D. Bignell. The malmo platform for artificial intelligence experimentation. In *IJCAI*, pages 4246–4247, 2016.

S. Kapturowski, G. Ostrovski, J. Quan, R. Munos, and W. Dabney. Recurrent experience replay in distributed reinforcement learning. In *ICLR*, 2019.

P. Kormushev, S. Calinon, and D. Caldwell. Reinforcement learning in robotics: Applications and real-world challenges. *Robotics*, 2:122–148, 2013.

T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. In Y. Bengio and Y. LeCun, editors, *ICLR*, 2016.

E. Z. Liu, K. Guu, P. Pasupat, T. Shi, and P. Liang. Reinforcement learning on web interfaces using workflow-guided exploration. *International Conference on Learning Representations (ICLR)*, 2018.

F. Liu, R. Tang, X. Li, W. Zhang, Y. Ye, H. Chen, H. Guo, and Y. Zhang. Deep reinforcement learning based recommendation with explicit user-item interactions modeling, 2019.

M. C. Machado, M. G. Bellemare, E. Talvitie, J. Veness, M. Hausknecht, and M. Bowling. Revisiting the Arcade learning environment: Evaluation protocols and open problems for general agents, 2017.

V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540): 529–533, Feb. 2015. ISSN 00280836.

Mojang. Minecraft. https://minecraft.net, 2014.

M. Moravčík, M. Schmid, N. Burch, V. Lisý, D. Morrill, N. Bard, T. Davis, K. Waugh, M. Johanson, and M. Bowling. Deepstack: Expert-level artificial intelligence in no-limit poker. *Science*, 356, 01 2017. doi: 10.1126/science.aam6960.

A. Muldal, Y. Doron, J. Aslanides, T. Harley, T. Ward, and S. Liu. dm_env: A python interface for reinforcement learning environments, 2019. URL http://github.com/deepmind/dm_env.

J. Oh, V. Chockalingam, Satinder, and H. Lee. Control of memory, active perception, and action in minecraft. In M. F. Balcan and K. Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 2790–2799, New York, New York, USA, 20–22 Jun 2016. PMLR.

OpenAI. OpenAI Universe. https://openai.com/blog/universe/, 2016.

I. Refanidis, N. Bassiliades, I. Vlahavas, and T. Greece. AI planning for transportation logistics. 12 2001.

J. Schaeffer, J. Culberson, N. Treloar, B. Knight, P. Lu, and D. Szafron. A world championship caliber checkers program. *Artificial Intelligence*, 53:53–2, 1992.

M. H. S. Segler, M. Preuss, and M. P. Waller. Planning chemical syntheses with deep neural networks and symbolic AI. *Nat.*, 555(7698):604–610, 2018.

T. Shi, A. Karpathy, L. Fan, J. Hernandez, and P. Liang. World of bits: An open-domain platform for web-based agents. In D. Precup and Y. W. Teh, editors, *Proceedings of the 34th International Conference on Machine Learning*, volume 70 of *Proceedings of Machine Learning Research*, pages 3135–3144. PMLR, 06–11 Aug 2017.

T. Shu, C. Xiong, and R. Socher. Hierarchical and interpretable skill acquisition in multi-task reinforcement learning. *CoRR*, abs/1712.07294, 2017.

D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484, 2016.

D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

Y. Tassa, S. Tunyasuvunakool, A. Muldal, Y. Doron, S. Liu, S. Bohez, J. Merel, T. Erez, T. Lillicrap, and N. Heess. dm_control: Software and tasks for continuous control, 2020.

C. Tessler, S. Givony, T. Zahavy, D. J. Mankowitz, and S. Mannor. A deep hierarchical approach to lifelong learning in Minecraft. In S. P. Singh and S. Markovitch, editors, *AAAI*, pages 1553–1561. AAAI Press, 2017.

E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012. doi: 10.1109/IROS.2012.6386109.

O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, J. Oh, D. Horgan, M. Kroiss, I. Danihelka, A. Huang, L. Sifre, T. Cai, J. P. Agapiou, M. Jaderberg, A. S. Vezhnevets, R. Leblond, T. Pohlen, V. Dalibard, D. Budden, Y. Sulsky, J. Molloy, T. L. Paine, C. Gulcehre, Z. Wang, T. Pfaff, Y. Wu, R. Ring, D. Yogatama, D. Wünsch, K. McKinney, O. Smith, T. Schaul, T. P. Lillicrap, K. Kavukcuoglu, D. Hassabis, C. Apps, and D. Silver. Grandmaster level in Starcraft II using multi-agent reinforcement learning. *Nat.*, 575(7782):350–354, 2019.

Acknowledgements

It would have been impossible to create AndroidEnv without the help of many people:

- Mathieu Méa for integrating dozens of open source apps.
- Natalie Lambert for coordinating between research teams and interfacing with external parties.
- Xutong Zhao for setting up benchmarks and sandboxing.
- Daniel Rodriguez, Gabriel Taubman, Chris Rawles, Wei Li, Robert Berry and Alice Li for extensive research collaborations with Google.
- Eser Aygün, Rui Zhu, Tom Ward, Alexandre Moufarek, Jay Lemmon, Davide Vercelli, Alban Rrustemi, Jeff Stanway, Damion Yates, David Barker, Duncan Williams, Tim Harley and Erwin Jansen for helping to set up AndroidEnv on Google's infrastructure.
- Linfeng (Frank) Yang and Yahan Zhou for Android Emulator guidance.
- Justin Novosad, Antonio Maiorano, Sean Risser, André Kaba, Alban Chagnoleau, Loïc Gelle, Jing Liu and Félix Larose-Gervais for prototyping interesting ideas in the early stages of the project.
- Tom Ward, Ankit Anand and Nando de Freitas for very useful feedback on a draft of this report.
- Phoebe Kirk, Gabrielle Ohlsen, Michelle Dunlop, Richard Ives for their legal advice.
- Aliya Ahmad, Emma Yousif, Louise Deason, Malcolm Reynolds for assistance on the open sourcing process and on communications.
- The Google and DeepMind Montréal team for enthusiastic discussions throughout the inception and refinement of AndroidEnv.