A. Environment Descriptions

In all environments, procedural generation controls the selection of game assets and backgrounds, though some environments include a more diverse pool of assets and backgrounds than others. When procedural generation must place entities, it generally samples from the uniform distribution over valid locations, occasionally subject to game-specific constraints. Several environments use cellular automata (Johnson et al., 2010) to generate diverse level layouts.

A.1. CoinRun

A simple platformer. The goal is to collect the coin at the far right of the level, and the player spawns on the far left. The player must dodge stationary saw obstacles, enemies that pace back and forth, and chasms that lead to death. Note that while the previously released version of CoinRun painted velocity information directly onto observations, the current version does not. This makes the environment significantly more difficult.

Procedural generation controls the number of platform sections, their corresponding types, the location of crates, and the location and types of obstacles.

A.2. StarPilot

A simple side scrolling shooter game. All enemies fire projectiles that directly target the player, so an inability to dodge quickly leads to the player's demise. There are fast and slow enemies, stationary turrets with high health, clouds which obscure player vision, and impassable meteors.

Procedural generation controls the spawn timing of all enemies and obstacles, along with their corresponding types.

A.3. CaveFlyer

The player controlling a starship must navigate a network of caves to reach the goal (a friendly starship). Player movement mimics the Atari game "Asteroids": the ship can rotate and travel forward or backward along the current axis. The majority of the reward comes from successfully reaching the goal, though additional reward can be collected by destroying target objects along the way with the ship's lasers. There are stationary and moving lethal obstacles throughout the level.

Procedural generation controls the level layout via cellular automata, as well as the configuration of all enemies, targets, obstacles, and the goal.

A.4. Dodgeball

Loosely inspired by the Atari game "Berzerk". The player spawns in a room with walls and enemies. Touching a wall loses the game and ends the episode. The player moves relatively slowly and can navigate throughout the room. There are enemies which also move slowly and which will occasionally throw balls at the player. The player can also throw balls, but only in the direction they are facing. If all enemies are hit, the player can move to the unlocked platform and earn a significant level completion bonus.

Procedural generation controls the level layout by recursively generating room-like structures. It also controls the quantity and configuration of enemies.

A.5. FruitBot

A scrolling game where the player controls a robot that must navigate between gaps in walls and collect fruit along the way. The player receives a positive reward for collecting a piece of fruit, and a larger negative reward for mistakenly collecting a non-fruit object. On expectation, half of the spawned objects are fruit (positive reward) and half are non-fruit (negative reward). The player receives a large reward if they reach the end of the level. Occasionally the player must use a key to unlock gates which block the way.

Procedural generation controls the level layout by sequentially generating barriers with randomly-sized gaps. It also controls the quantity and configuration of fruit and non-fruit objects, as well as the placement of gates.

A.6. Chaser

Inspired by the Atari game "MsPacman". The player must collect all the green orbs in the level. 3 large stars spawn that will make enemies vulnerable for a short time when collected. A collision with an enemy that isn't vulnerable results in the player's death. When a vulnerable enemy is eaten, an egg spawns somewhere on the map that will hatch into a new enemy after a short time, keeping the total number of enemies constant. The player receives a small reward for collecting each orb and a large reward for completing the level.

Procedural generation controls the level layout by generating mazes using Kruskal's algorithm (Kruskal, 1956), and then removing walls until no dead-ends remain. The large stars are constrained to spawn in different quadrants. Initial enemy spawn locations are randomly selected.

A.7. Miner

Inspired by the classic game "BoulderDash". The player, a robot, can dig through dirt to move throughout the world. The world has gravity, and dirt supports boulders and dia-

monds. Boulders and diamonds will fall through free space and roll off each other. If a boulder or a diamond falls on the player, the game is over. The goal is to collect all the diamonds in the level and then proceed through the exit. The player receives a small reward for collecting a diamond and a larger reward for completing the level.

Procedural generation controls the position of all boulders, diamonds, and the exit. No objects may spawn adjacent to the player. An approximately fixed quantity of boulders and diamonds spawn in each level.

A.8. Jumper

A platformer with an open world layout. The player, a bunny, must navigate through the world to find the carrot. It might be necessary to ascend or descend the level to do so. The player is capable of "double jumping", allowing it to navigate tricky layouts and reach high platforms. There are spike obstacles which will destroy the player on contact. The screen includes a compass which displays direction and distance to the carrot. The only reward in the game comes from collect the carrot, at which point the episode ends.

Procedural generation controls the level layout via cellular automata, which is seeded with a maze-like structure. Long flat vertical edges are intentionally perturbed to avoid unsolvable levels, as the player can take advantage of irregular ledges on vertical walls. Obstacles cannot spawn adjacent to each other, as this could create impassable configurations.

A.9. Leaper

Inspired by the classic game "Frogger". The player must cross several lanes to reach the finish line and earn a reward. The first group of lanes contains cars which must be avoided. The second group of lanes contains logs on a river. The player must hop from log to log to cross the river. If the player falls in the river, the episode ends.

Procedural generation controls the number of lanes of both roads and water, with these choices being positively correlated. It also controls the spawn timing of all logs and cars.

A.10. Maze

The player, a mouse, must navigate a maze to find the sole piece of cheese and earn a reward. The player may move up, down, left or right to navigate the maze.

Procedural generation controls the level layout by generating mazes using Kruskal's algorithm (Kruskal, 1956), uniformly ranging in size from 3x3 to 25x25.

A.11. BigFish

The player starts as a small fish and becomes bigger by eating other fish. The player may only eat fish smaller than itself, as determined solely by width. If the player comes in contact with a larger fish, the player is eaten and the episode ends. The player receives a small reward for eating a smaller fish and a large reward for becoming bigger than all other fish, at which point the episode ends.

Procedural generation controls the spawn timing and position of all fish.

A.12. Heist

The player must steal the gem hidden behind a network of locks. Each lock comes in one of three colors, and the necessary keys to open these locks are scattered throughout the level. The level layout takes the form of a maze. Once the player collects a key of a certain color, the player may open the lock of that color. All keys in the player's possession are shown in the top right corner of the screen.

Procedural generation controls the level layout by generating mazes using Kruskal's algorithm (Kruskal, 1956). Locks and keys are randomly placed, subject to solvability constraints.

A.13. Climber

A simple platformer. The player must climb a sequence of platforms, collecting stars along the way. A small reward is given for collecting a star, and a larger reward is given for collecting all stars in a level. If all stars are collected, the episode ends. There are lethal flying monsters scattered throughout the level.

Procedural generation controls the level layout by sequentially generating reachable platforms. Enemies and stars spawn near each platform with fixed probabilities, except when spawning an enemy would lead to an unsolvable configuration. The final platform always contains a star.

A.14. Plunder

The player must destroy enemy pirate ships by firing cannonballs from its own ship at the bottom of the screen. An on-screen timer slowly counts down. If this timer runs out, the episode ends. Whenever the player fires, the timer skips forward a few steps, encouraging the player to conserve ammunition. The player should also avoid hitting friendly ships. The player receives a positive reward for hitting an enemy ship and a large timer penalty for hitting a friendly ship. A target in the bottom left corner identifies the color of the enemy ships to target. Wooden obstacles capable of blocking the player's line of sight may exist.

Procedural generation controls the selection of friendly and enemy ship types, as well as the spawn times and positions of all non-player ships. It also controls the placement of wooden obstacles.

A.15. Ninja

A simple platformer. The player, a ninja, must jump across narrow ledges while avoiding bomb obstacles. The player can toss throwing stars at several angles in order to clear bombs, if necessary. The player's jump can be charged over several timesteps to increase its effect. The player receives a reward for collecting the mushroom at the end of the level, at which point the episode terminates.

Procedural generation controls the level layout by sequentially generating reachable platforms, with the possibility of superfluous platform generation. Bombs are occasionally randomly placed near platforms.

A.16. Bossfight

The player controls a small starship and must destroy a much bigger boss starship. The boss randomly selects from a set of possible attacks when engaging the player. The player must dodge the incoming projectiles or be destroyed. The player can also use randomly scattered meteors for cover. After a set timeout, the boss becomes vulnerable and its shields go down. At this point, the players projectile attacks will damage the boss. Once the boss receives a certain amount of damage, the player receives a reward, and the boss re-raises its shields. If the player damages the boss several times in this way, the boss is destroyed, the player receives a large reward, and the episode ends.

Procedural generation controls certain game constants, including the boss health and the number of rounds in a level. It also selects the configuration of meteors in the level, and the attack pattern sequence the boss will follow.

B. Core Capabilities in RL

To better understand the strengths and limitations of current RL algorithms, it is valuable to have environments which isolate critical axes of performance. (Osband et al., 2019) recently proposed seven core RL capabilities to profile with environments in bsuite. We focus our attention on three of these core capabilities: generalization, exploration, and memory. Among these, Procgen Benchmark contributes most directly to the evaluation of generalization, as we have already discussed at length. In this section, we describe how Procgen environments can also shed light on the core capabilities of exploration and memory.

B.1. Evaluating Exploration

The trade off between exploration and exploitation has long been recognized as one of the principal challenges in reinforcement learning. Although exploration plays some role in every environment, the difficulty of the exploration problem can vary drastically. In many environments, the ability to adequately explore becomes an overwhelming bottleneck in agents' training. With Procgen environments, we strive to be deliberate in our consideration of exploration.

The generalization curves in Figure 2 show that training performance often increases with the size of the training set. This reveals an interesting phenomenon: exploration can become less of a bottleneck in the presence of greater diversity. On the other hand, when the training set is restricted and diversity is removed, an otherwise tractable environment can become intractable due to exploration. By taking this to the extreme and restricting training to a single high difficulty level, 8 of the Procgen environments can be made into highly challenging exploration tasks. In doing so, these environments come to resemble traditional hard exploration environments, like the infamous Atari game Montezuma's Revenge. We note that generalization is not measured in this setting; the focus is solely on the agent's ability to explore.

The 8 environments that specifically support the evaluation of exploration are CoinRun, CaveFlyer, Leaper, Jumper, Maze, Heist, Climber, and Ninja. For each environment, we handpick a level seed that presents a significant exploration challenge. Instruction for training on these specific seeds can be found at https://github.com/openai/train-procgen. On these levels, a random agent is extraordinarily unlikely to encounter any reward. For this reason, our baseline PPO implementation completely fails to train, achieving a mean return of 0 in all environments after 200M timesteps of training.

B.2. Evaluating Memory

The extent to which agents must attend to the past varies greatly by environment. In environments in the ALE, memory beyond a small frame stack is not generally required to achieve optimal performance. In more general settings and in more complex environments, we expect memory to become increasingly relevant.

By default, Procgen environments require little to no use of memory, and non-recurrent policies achieve approximately the same level of performance as recurrent policies. We designed environments in this way to better isolate the challenges in RL. However, 6 of the 16 Procgen environments support variants that do require memory. These variants remove linearity constraints from level generation and increase the impact of partial observability. By introducing a dependence on memory, these environments become dramatically more difficult.

The 6 environments that specifically support the evaluation of memory are CoinRun, CaveFlyer, Dodgeball, Miner, Jumper, Maze, and Heist. In this setting, we modify the environments as follows. In all environments we increase the world size. In Caveflyer and Jumper, we remove logic in level generation that prunes away paths which do not lead to the goal. In Dodgeball, Miner, Maze, and Heist, we make the environments partially observable by restricting observations to a small patch of space surrounding the agent. We note that Caveflyer and Jumper were already partially observable. With these changes, agents can reliably solve levels only by utilizing memory. Instructions for training environments in memory mode can be found at https://github.com/openai/train-procgen.

C. Normalization Constants

R_{min} is computed by training a policy with masked out observations. This demonstrates what score is trivially achievable in each environment. R_{max} is computed in several different ways.

For CoinRun, Dodgeball, Miner, Jumper, Leaper, Maze, BigFish, Heist, Plunder, Ninja, and Bossfight, the maximal theoretical and practical reward is trivial to compute.

For CaveFlyer, Chaser, and Climber, we empirically determine R_{max} by generating many levels and computing the average max achievable reward.

For StarPilot and FruitBot, the max practical reward is not obvious, even though it is easy to establish a theoretical bound. We choose to define R_{max} in these environments as the score PPO achieves after 8 billion timesteps when trained at an 8x larger batch size than our default hyperparameters. On observing these policies, we find them very close to optimal.

	Hard		Easy	
Environment	R_{min}	R_{max}	R_{min}	R_{max}
CoinRun	5	10	5	10
StarPilot	1.5	35	2.5	64
CaveFlyer	2	13.4	3.5	12
Dodgeball	1.5	19	1.5	19
FruitBot	-.5	27.2	-1.5	32.4
Chaser	.5	14.2	.5	13
Miner	1.5	20	1.5	13
Jumper	1	10	3	10
Leaper	1.5	10	3	10
Maze	4	10	5	10
BigFish	0	40	1	40
Heist	2	10	3.5	10
Climber	1	12.6	2	12.6
Plunder	3	30	4.5	30
Ninja	2	10	3.5	10
BossFight	.5	13	.5	13

D. Hyperparameters

We use the Adam optimizer (Kingma and Ba, 2014) in all experiments.

Table 1. PPO Hyperparameters

ENV. DISTRIBUTION MODE	HARD	EASY
γ	.999	.999
λ	.95	.95
# TIMESTEPS PER ROLLOUT	256	256
EPOCHS PER ROLLOUT	3	3
# MINIBATCHES PER EPOCH	8	8
ENTROPY BONUS (k_H)	.01	.01
PPO CLIP RANGE	.2	.2
REWARD NORMALIZATION?	YES	YES
LEARNING RATE	5×10^{-4}	5×10^{-4}
# WORKERS	4	1
# ENVIRONMENTS PER WORKER	64	64
TOTAL TIMESTEPS	200M	25M
LSTM?	NO	NO
FRAME STACK?	NO	NO

Table 2. Rainbow Hyperparameters

ENV. DISTRIBUTION MODE:	HARD
γ	.99
LEARNING RATE	2.5×10^{-4}
# WORKERS	8
# ENVIRONMENTS PER WORKER	64
# ENV. STEPS PER UPDATE PER WORKER	64
BATCH SIZE PER WORKER	512
REWARD CLIPPING?	NO
DISTRIBUTIONAL MIN/MAX VALUES	[0, R_{max}][2]
TOTAL TIMESTEPS	200M
LSTM?	NO
FRAME STACK?	NO

[2]In FruitBot, we change the distributional min value to -5.

E. Test Performance for All Training Sets

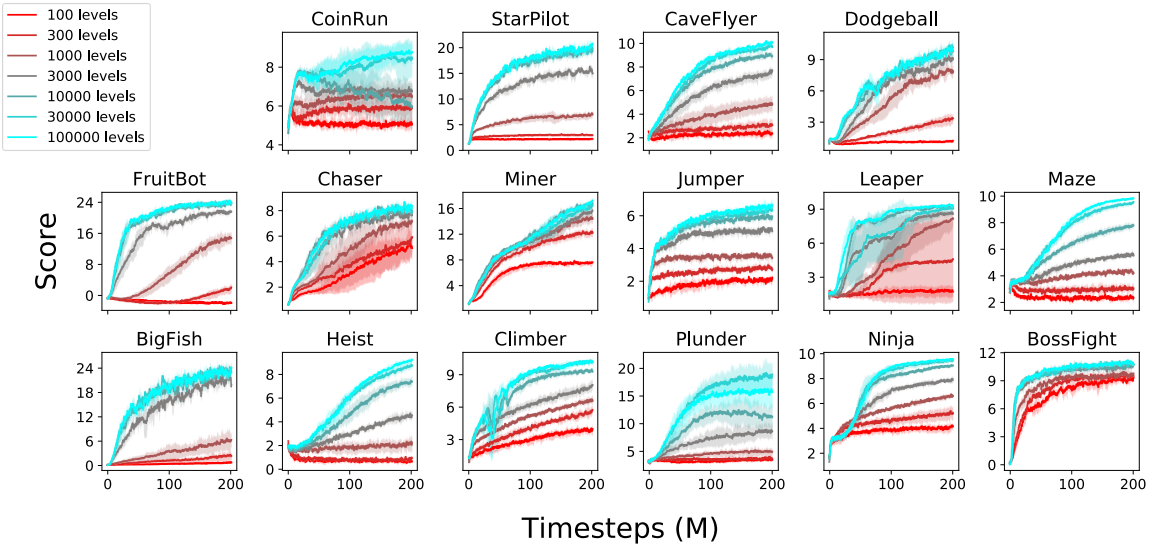

Figure 7. Test performance of agents trained on different sets of levels. All agents are evaluated on the full distribution of levels from each environment.

F. Arcade Learning Environment Performance

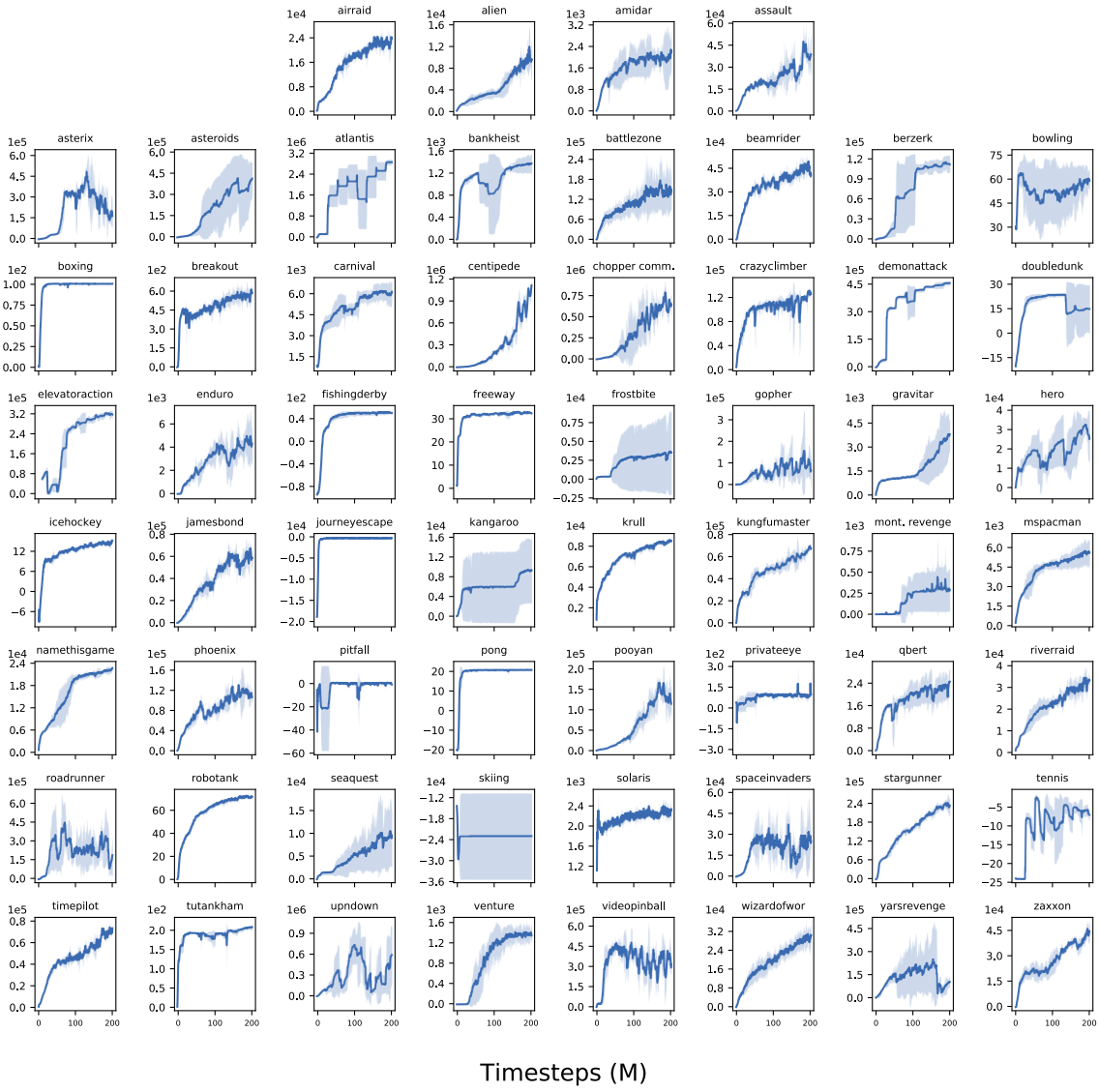

Figure 8. Performance of our implementation of PPO on the ALE.

In these ALE experiments, we use a frame stack of 4 and we do not use sticky actions (Machado et al., 2018). Note that although we render Procgen environments at 64x64 pixels, they can easily be rendered at 84x84 pixels to match the ALE standard, if desired.

G. Training Curves by Architecture

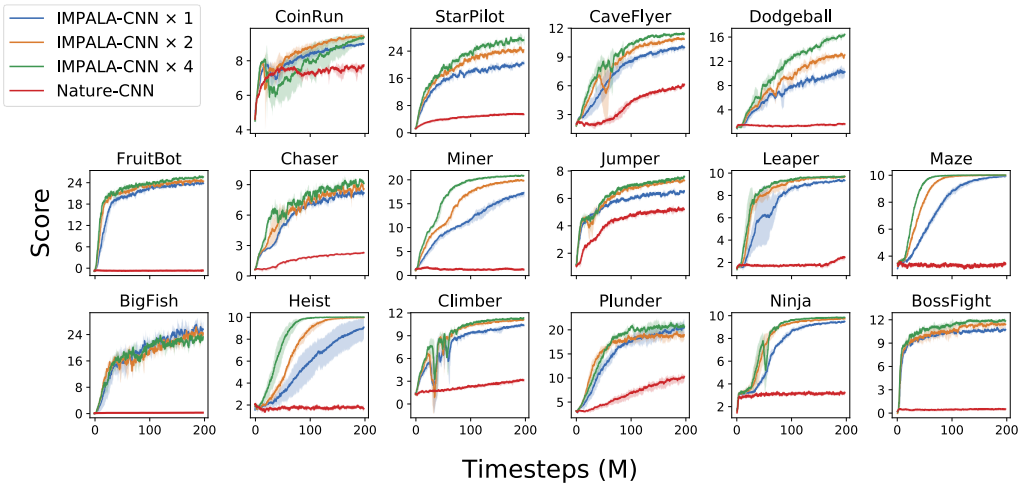

Figure 9. Performance of agents using each different architecture in each environment, trained and evaluated on the full distribution of levels.

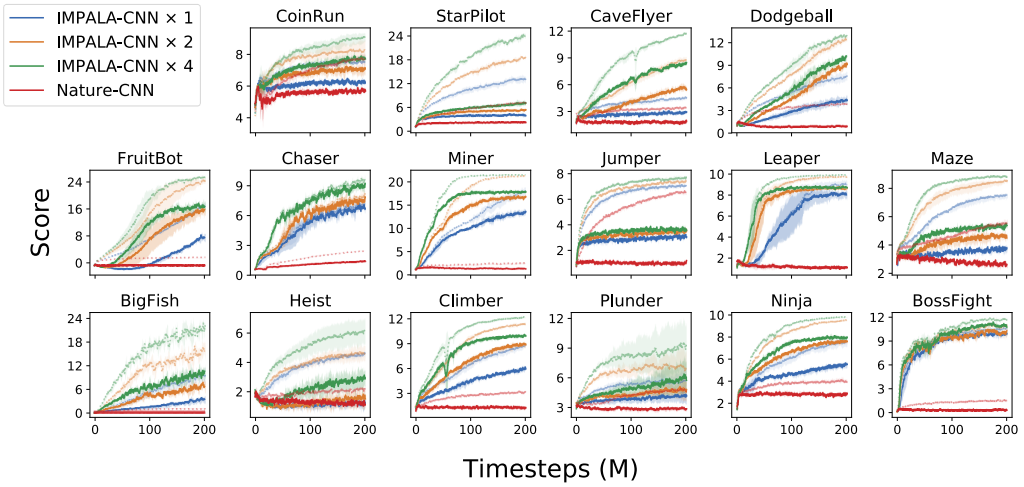

Figure 10. Performance of agents using each different architecture in each environment, trained on 500 levels and evaluated on held out levels. Light dashed lines denote training curves and dark solid lines denote test curves.

H. Frame Stack vs. LSTM

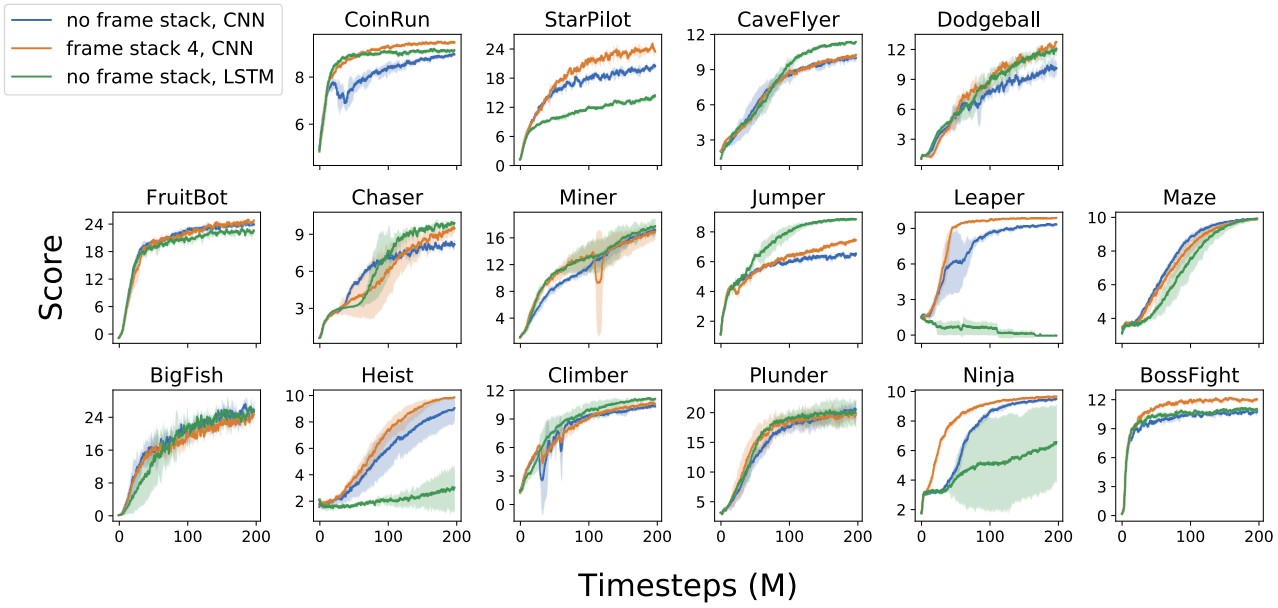

Figure 11. Comparison of our baseline to agents that use either frame stack or a recurrent architecture.

For simplicity, our baseline experiments forgo the use of frame stack by default. This limits agents to processing information from only the single current frame. We compare this baseline to agents that use a frame stack of 4. We also compare both methods to agents using an LSTM (Hochreiter and Schmidhuber, 1997) on top of the convolutional network.

In general, we find these methods to be fairly comparable. In the Jumper environment, the LSTM agents outperform others, perhaps as the ability to perform non-trivial temporally extended navigation is helpful. In other environments like Leaper and Ninja, our LSTM baseline is notably unstable. In most environments, frame stack agents perform similarly to baseline agents, though in a few environments the difference is noticeable.

I. Easy Difficulty Baseline Results

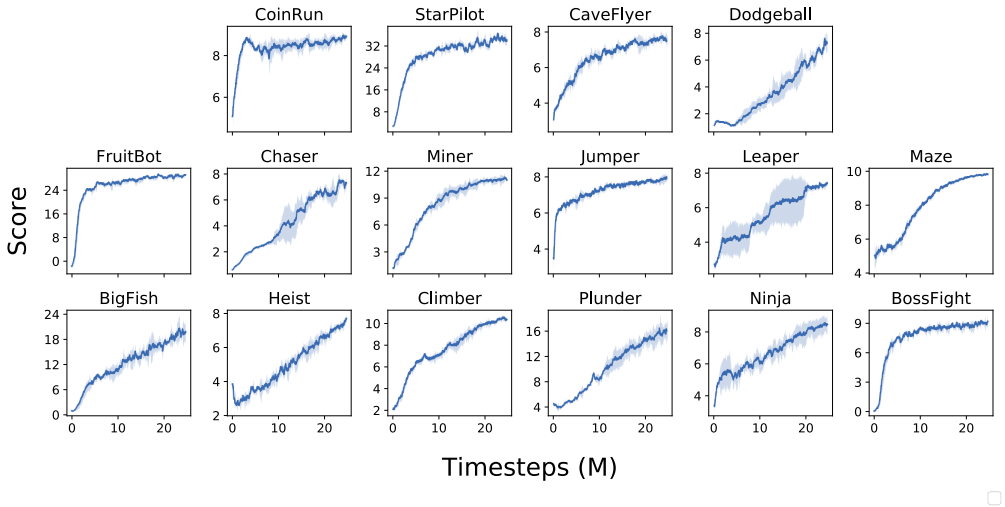

Figure 12. Performance of agents on easy difficulty environments, trained and evaluated on the full distribution of levels.

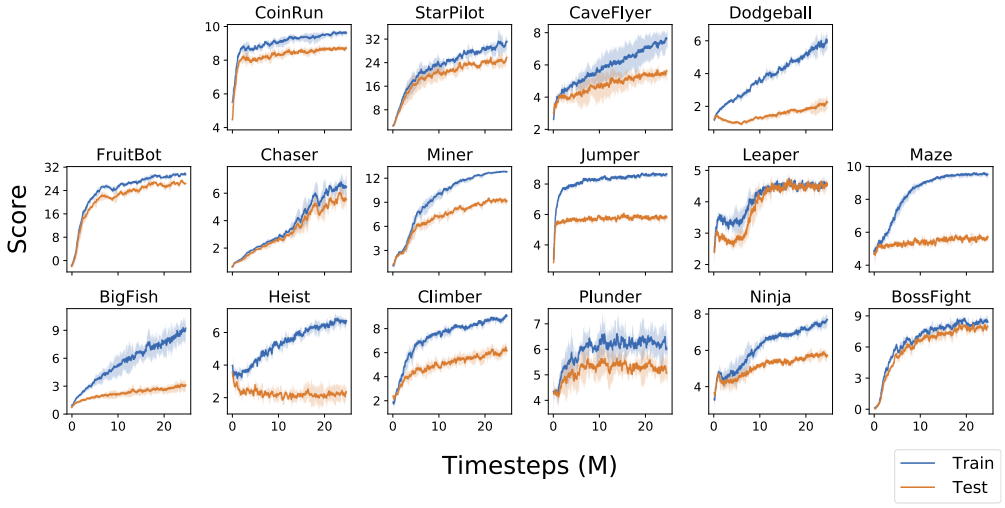

Figure 13. Performance of agents on easy difficulty environments, trained on 200 levels and evaluated on the full distribution of levels.