BroadMind: A Better Platforming Agent

Abstract

Recent work in reinforcement learning has focused on building generalist video game agents, as opposed to focusing on a particular genre of games. We aim to build a more specialized high-performance agent focused on the more challenging genre of platform games, which has received less attention. Utilizing symbolic representations of game state, we are training fully connected Neural Q-Network agents to successfully learn to play games with long term rewards and complex dynamics.

1. Background

Platform games involve a free-running avatar that jumps between suspended platforms and avoid obstacles to advance through levels in the game. As a result of the array of environments to parse and the huge decision space, these games are very difficult for learning agents to play well. We are building an agent does not have to simultaneously recognize the screen-space pixels of the game and decide optimal policies. Instead, we have decoupled these two problems, using symbolic state representations from the environment encoded for the agent. However, this symbolic representation is still very large, motivating us to utilize neural network-based Q-learning approaches.

2. Approach

2.1. Experimental Setup

We have developed a Neural Q-Network algorithm that can be extended as a reinforcement learning agent in multiple gaming environments including Generalized Mario and the Arcade Learning Environment (ALE). We utilize the RL-Glue framework to allow our agents and experiments to be used across these environments.

RL-Glue is a socket-based API that enables reinforcement learning experiments to work with software across multiple languages (Tanner & White, 2009). Is allows our experiments to connect reusable Python agents across many open source environments written in languages including C++ and Java. It also enables us to customize our experiments, such as optional game visualization, loading and saving trained policies, adjusting the game difficulty, etc.

We have started evaluating our agents in the Generalized Mario environment. This is part of the 2009 AI Competition software package and is RL-Glue compatible (Togelius et al., 2010). The Generalized Mario game has a total control input space of 12, which we encode to integers. The raw state is made up of left/right/none motions, on/off for jump, and on/off for run. The observation interface provides the 2D screen position and velocity of the mario actor, as well as a 22x16 tile grid semantically describing the screen space with the location of coins, pipes, blocks, etc. Separately, it has a list of all enemy positions on-screen, with similar position and velocity information as provided about Mario.

We have leveraged the Arcade Learning Environment, an RL-Glue compatible framework built on top of an Atari 2600 emulator (Bellemare et al., 2013). It provides an API for interfacing with the raw pixels, the current score, and the controller input of the game. This has allowed us to use original Atari games to train and evaluate our agents. We currently support 7 Atari platformers in our experimental setup:

1. Montezumas Revenge
2. Kung-Fu Master
3. Frostbite
4. Kangaroo
5. Pitfall!
6. Pitfall! 2
7. H.E.R.O.

2.2. Learning Algorithms

2.2.1. State Representation in Mario

It is challenging to find an effective representation of the Mario game state that enables effective Q-learning. The environment observations provided by Generalized Mario contains a wealth of symbolic information, but there are many possible encodings that we are investigating. Currently, our representation is a tiled grid of integers, 20x12 tiles in size. The relative value in each tile is given as a “measure of goodness”, enemies are -2, obstacles are -1, coins are +2, etc. The grid is centered on Mario’s current location. We intend to represent the state with separate substrates for each class of objects, as in (Hausknecht et al., 2013), by breaking it out into separate background, enemy, reward, and actor layers. We hope to find that this repre-

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

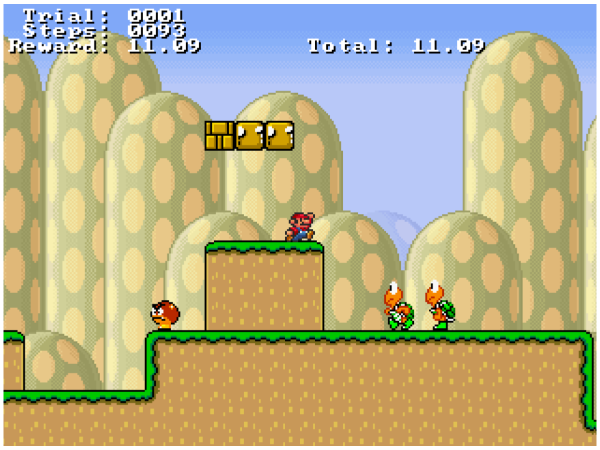

Figure 1. Raw pixels representing the state of the mario game, as it would be seen by a player.

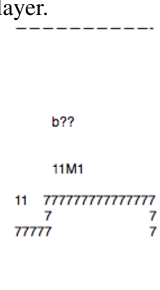

Figure 2. The equiivalent game state encoded as symbolic characters, as provided to the learning agent by the Mario environment in each timestep.

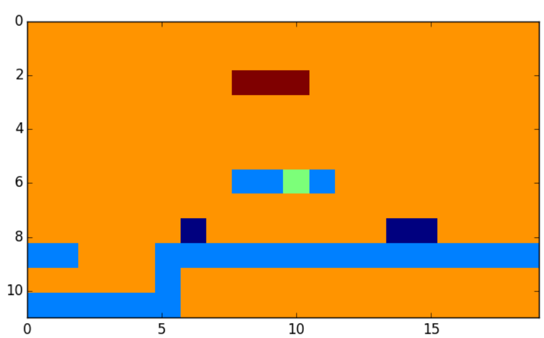

Figure 3. Our encoding of the game state in a 20x12 grid with Mario at the center, as provided to the Neural Q-Network. Larger postive values correspond to positive symbols such as coins and free space, and negative values correspond to negative symbols such as enemies and obstacles.

Algorithm 1 Neural Q-Network with Impactful Experience Replay

Initialize Neural Q-Network with random weights
Initialize Experience Pool to $\{\}$
for episode$= 1, m$ **do**
  Initialize previous state s_0 and previous action a_0 to NULL
  for $t = 1, T$ **do**
    Compute the mean and standard deviation of absolute value of the rewards in the experience pool, r_{ave}, r_{dev}.
    Observe state s_t from the emulator
    With probability ϵ_a, select random action a_t
    Else, set $a_i = \max_a Q(s_t, a)$ by forward propagating s_t through Q
    if s_{t-1} and a_{t-1} are not NULL **then**
      Observe reward r from emulator
      With probability $\epsilon_r * (1 + \frac{|r| - r_{ave}}{r_{dev}})$, store the experience $\{s_{t-1}, a_{t-1}, r, s_t, a_t\}$ in the pool
    end if
    for re-experience$= 1, ex$ **do**
      Randomly sample experience $\{s'_0, a'_0, r', s'_1, a'_1\}$ from the pool
      if using SARSA update rule **then**
        Compute $v = r' + \gamma Q(s'_1, a'_1)$
      else
        Compute $v = r' + \gamma \max_a Q(s'_1, a)$
      end if
      Update Q through backpropagation of value v on output a'_0 with state s'_0
    end for
    Apply action a_i to emulator
    Update state $s_{t-1} = s_t$ and action $a_{t-1} = a_t$
  end for
end for

sentation will be universal across platform games. None of these representations encode the important velocity information about the actors, which is important in platformers where the characters move with some inertia.

2.2.2. Neural Q-Learning

Typically, Q-Learning approaches use a table representing the Q-function. For very large state/action spaces such as platforming games, this is impractical as the space would take too many trials to explore and converge on an optimal policy. Even using optimizations such as nearest neighbor ran out of memory for our Generalized Mario state representation. We have implemented a neural-network based Q-learning algorithm (see Algorithm 1) to allow us to learn on large state/action spaces with reasonable memory utilization by finding a useful hidden layer.

Inspired by DeepMind's approach (Mnih et al., 2013), we have avoided multiple forward propagation steps when selecting optimal actions by using only the state as the input to the network, and a weight for each action as the output. Thus, we can select an optimal action for a state by propagating once, and selecting the max argument from the outputs. Also like DeepMind, we include an Experience Replay step that stores a history of events to re-learn. This avoids overfitting to the current situation and unlearning good behavior from earlier episodes. Our algorithm can optionally use the standard Q-Learning update, or the SARSA calculation.

We have adjusted the experience rememberance process to prioritize memorization of meaningful experiences. We observe that experiences corresponding with large positive or negative rewards are rare, but these are the experiences that provide the most value to the agent. To do this, we modify the probability of rememberance, multiplying by $1 + \frac{|r| - r_{ave}}{r_{dev}}$.

2.2.3. Extension to Atari Platformers

We have setup the ALE environment, and connected default agents to it. However, we have yet to attempt to port our Mario agents to these problems. Initially, we will use 3 colored pixel substrates as the state representation, as in (Hausknecht et al., 2013), but we hope to reconstruct an object layer as well.

3. Results

We have found that the initial random weights in the neural network can make the early policies very flawed. To counter this, we use heavy exploration bias ($\epsilon = 1.0$) for early episodes, while transitioning to exploitation policies ($\epsilon = 0.1$) at later episodes. We have also biased the random action to prefer motion to the right, helping the agent to explore more of the game. We trained agents over 1000 episodes in the Mario environment using the state encoded in the format described in section 2.2.1, and using the Neural Q-Network with experience replay of Algorithm 1. We use Mario level of difficulty 1 so that the levels provide a challenge with enemies, but not so hard that a random agent cannot make considerable progress.

Due to the variation in available rewards between each Mario level, we have compared all performance results against an agent that acts randomly. To reduce noise in our plots, we compute a running average of total rewards over 100 episodes. The Mario environment provides a substantially large reward for reaching the end of the level. When averaging this reward over 100 levels, it provides too much weight that de-emphasizes the results of the other episodes. For this reason, we have capped the max reward that a sin-

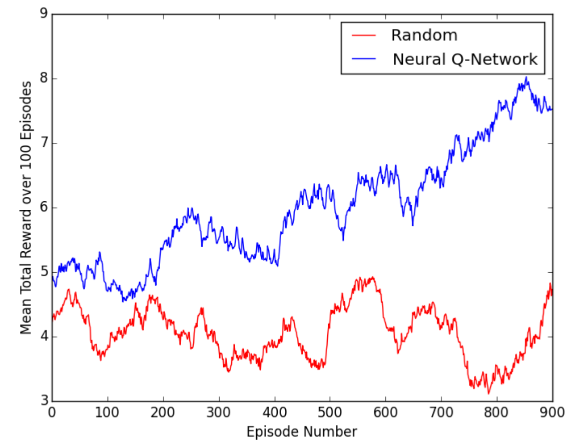

Figure 4. Mario agent trained with a neural q-network with a hidden layer of 126 nodes. The agent was trained for 1000 episodes of the same level seed 3 and difficulty 1. The initial exploration factor was 1.0, and this decreased by 0.05 every 100 episodes, until stopping at 0.1. The total reward gained by the agent was summed over each episode, and the running average of 100 episodes is shown here.

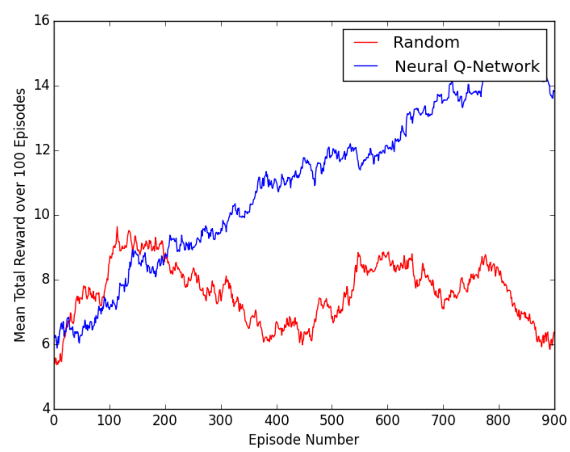

Figure 5. The same test performed with level seed 8. At its peak, the mean reward earned for the learned agent is roughly double that of random behavior

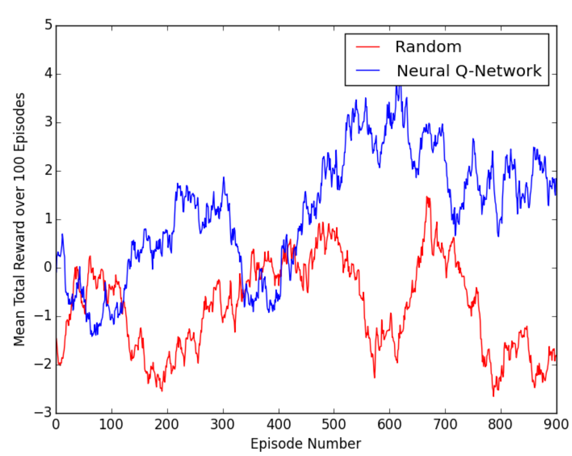

Figure 6. Another test with level seed 20. This level has less available positive rewards, so the learned agent does not show the same level of performance improvement.

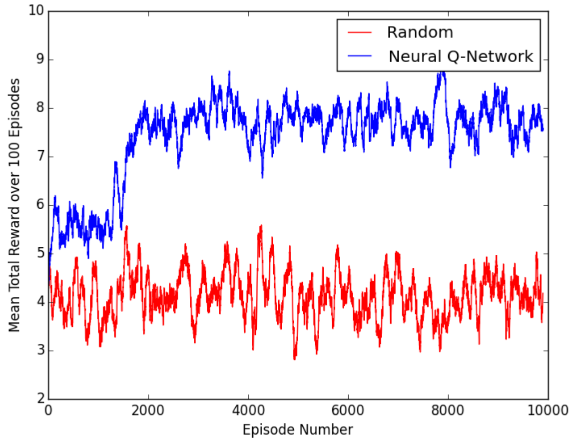

Figure 7. Level seed 3 continued out to 10,000 episodes. Performance ceases to improve past episode 2,000.

gle episode can add to the running average.

3.1. Transfer Learning Between Levels

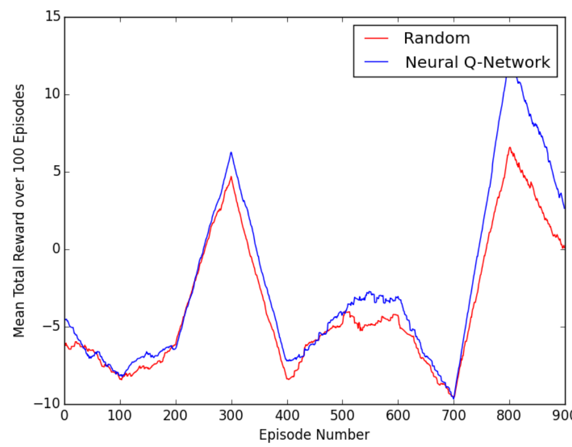

Figure 8. This test matches earlier experiments, though every 100 episodes the Mario envrionment changes to a new level. Episode number 300 and 800 correspond to the results of averaging 100 episodes of only level seed 3 and 8 respectively. The performance of the learned agent is consistent with what we observed in earlier tests with training an agent only on these levels.

Acknowledgments

None.

References

Bellemare, Marc G, Naddaf, Yavar, Veness, Joel, and Bowling, Michael. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, 2013.

Hausknecht, M., Lehman, J., Miikkulainen, R., and Stone, P. A neuroevolution approach to general atari game playing. *IEEE Transactions on Computational Intelligence and AI in Games*, 2013.

Mnih, Volodymyr, Kavukcuoglu, Koray, Silver, David, Graves, Alex, Antonoglou, Ioannis, Wierstra, Daan, and Riedmiller, Martin. Playing atari with deep reinforcement learning. *NIPS Workshop on Deep Learning and Unsupervised Feature Learning*, 2013.

Tanner, Brian and White, Adam. Rl-glue: Language-independent software for reinforcement-learning experiments. *Journal of Machine Learning Research*, 10: 2133–2136, 2009.

Togelius, J., Karakovskiy, S., and Baumgarten, R. The 2009 mario ai competition. *Evolutionary Computation (CEC), 2010 IEEE Congress on.*, pp. 1–8, 2010.