Learning to play SLITHER.IO with deep reinforcement learning
Joan Creus-Costa & Zhanpei Fang, {jcreus, zhanpei}@stanford.edu

Introduction and motivation
This project uses deep reinforcement learning (RL) to train an agent to play the massively multiplayer online game (MMO) SLITHER.IO, in which players attempt to maximize “snake” length by consuming multicolored food pellets while avoiding other players’ snakes. The goal of the project was to achieve human-like performance on the game, which presents a good target due to the relative simplicity of game mechanics and the possibility of complex emergent behavior. Introduced in 2015, deep Q-learning has been applied with great success to a wide variety of gameplay scenarios, such as Atari games Mnih et al. [2015].

Dataset and features
We collect training data using an OpenAI Universe environment, which interacts with the online game via a simulated screen and virtual keyboard. Our inputs are the raw frames from the remote desktop, and reward signals corresponding to change in score (length of snake).

Pre-processing steps include:
- Crop and downsize image, from 768 × 1024 × 3 pixel RGB image → 150 × 250 × 3 RGB pixels.
- Frame skipping from 60 → 5 frames per second
- 8-bit color range (0–255) → floating-point values (0–1).
- Encode difference from previous frame by computing Δ(previous frame – current frame) for each channel, for a total of 150 × 250 × 6 input features.

Model: Deep Q-Learning
Q-learning learns the action-value function $Q(s, a)$ from pixels $s$:

$Q^*(s, a) = E_{s’ \sim p(s’|s, a)} \left[ \gamma r + \max_{a’} Q^*(s’, a’) \right]$

Q-learning update at iteration $i$ uses Huber loss:

$L_\text{huber}(a) = \frac{1}{2} \left| r + \gamma \max_{a’} Q(s’, a’) - Q(s, a) \right|^2 \text{if } |\epsilon| \leq \delta,$

$= (r + \gamma \max_{a’} Q(s’, a’) - Q(s, a))^2 \text{otherwise.}$

A variety of tricks were necessary to be able to learn a policy:

- Learn from human demonstrations to facilitate training, as in [Hester et al., 2018]. We collected around 50 demonstrations to pretrain the network. We add the following term to the cost function

$L_E(Q) = \max_{a \in A} Q(s, a) + \epsilon a \alpha - Q(s, a)$

where $\epsilon a$ is a margin function that’s 0 when the actions are equal.

- Reward shaping disincentivizes ending the episode [Ng et al., 1999]:

$r_i’ = \begin{cases} -10 & T - r < 10 \\ r_i & \text{otherwise} \end{cases}$

- Prioritize experience replay to sample transitions with or near a reward to compensate for sparsity of rewards and mitigate instability.

Results

<table>
<thead>
<tr>
<th>Model</th>
<th>Median score</th>
<th>Average reward</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random</td>
<td>$3^{1/2}$</td>
<td>0.08</td>
</tr>
<tr>
<td>Human</td>
<td>$145^{3/2}$</td>
<td>0.68</td>
</tr>
</tbody>
</table>

No human demonstrations, ϵ-greedy, $K = 1.5 \times 10^5$ batches
Pretrain on human demonstrations, $K = 1.5 \times 10^5 \times$ followed by $K_h = 3 \times 10^5$ collected batches

* Estimated 50% confidence interval using the bootstrap, using median instead of mean to avoid bias from outliers. **Human performance was tested in the simulated environment instead of the website, and was therefore worse than normal play due to the environment limitations (low resolution, low frame rate).

Discussion

- Realtime Internet playing introduces significant bottleneck to training. Combined with the complexity of the game, this results in high difficulty in training a better-than-human policy.
- Using human demonstrations results in a tremendous speedup in training as well as in a much better model. 15,000 pretraining steps yields a better model than 150,000 ϵ-greedy steps; combining both yields the best model in 45,000 iterations.
- Reward shaping and prioritized replay also helped get most of the (expensive) transitions to acquire. Using PyTorch’s CUDA integration allows for significantly faster training.
- Limited model capacity since we only use a single delta frame, and implement vanilla DQN.

Future work

- Switch to better RL methods, given that more training with the current model did not yield significant improvements, such as recurrent network architecture, or actor-critic models or Rainbow agents which have also shown promise in similar environments.
- Use cloud infrastructure to allow for faster training to avoid the limitations of the network-based game. Use asynchrononous methods (e.g. A3C) to distribute parameters.

Selected references:

Poster template based on the UCL Beamer template. https://github.com/UCL/ucl-beamer