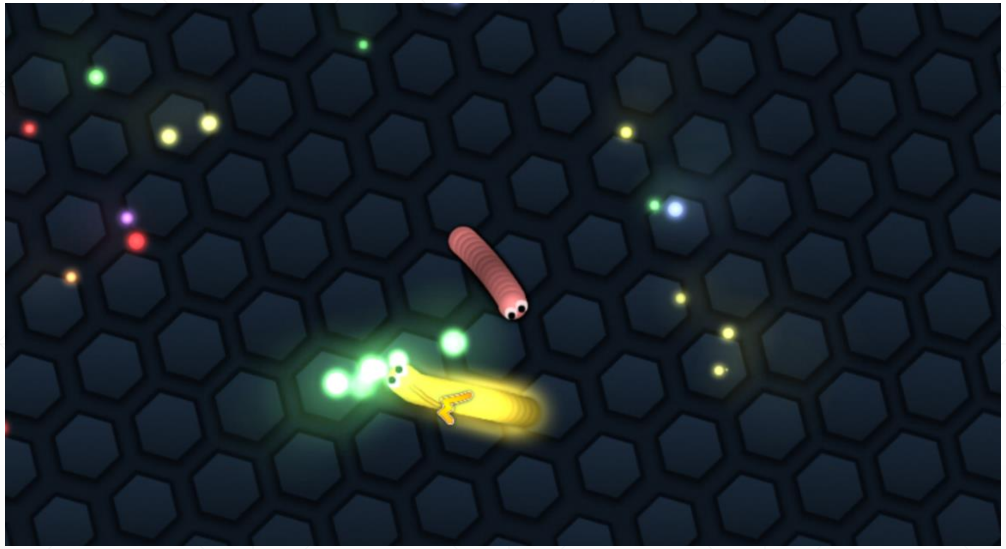
# Deep Slither.io

A Deep Q Learning Approach to Slither.io

Brian Voter [bv85]

# The Game



- Slither.io – a massively multiplayer browser game
- Mouse controls a worm
- Consuming glowing “food” pellets increases worm size and score
- Cut off other worms to make them disintegrate into food
- Goal: Become as large as possible, as fast possible

# Objective and Constraints

- Learn to play optimally using only the game pixels as inputs
- Reward signal is calculated each transition: $\mathrm{r} = \mathrm{score}(s') - \mathrm{score}(s)$
- Inspired by Minh et al. 2013: Playing Atari with Deep Reinforcement Learning
- Key Differences:
  - Not an offline emulator – can't control flow of states
  - Instead must poll current state from the browser at a regular interval

# High-level Approach

- Game is controlled using browser automation software
- Neural Net accepts pixel matrix as input
- Output: predicted quality of each action for the state
- The action of the highest quality is performed, yielding a reward and transitioning to a new state

# Background: Q-Learning

- Foundation of approach is Q-learning
- Basic Q function: $Q(s, a)$ is the "quality" of performing action $a$ in state $s$

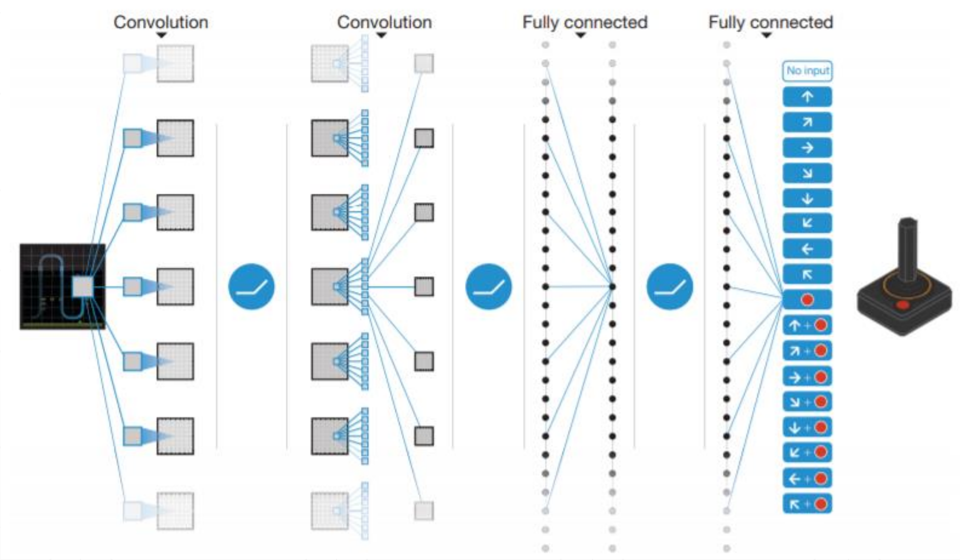
- **Problem**: Slither.io has infinitely many states
  - Can't learn quality of all possible states
  - Need to approximate

# Background: DQN

- **Solution:** Approximate $Q(s, a) \approx Q(s, a; \theta)$
  - Right side uses neural network with parameters $\theta$ to approximate left side
- Introduced by Minh et al. 2013
- Deep Q Network = DQN
- DQN is the foundation for my approach

# Background: DQN Architecture

- Original DQN accepts $[4 \times 84 \times 84]$ images
- Faster processing because images are downscaled from $[210 \times 160]$
- 4 images are stacked on the channel-dimension to give state history
  - Consider single *Pong* frame
  - For Atari 2600 games, changes POMDP → MDP




Mnih et al. 2015, *Nature*

# Background: Experience Replay

- **Problem**: as a DQN is trained on fresh states, it will "forget" transitions it learned earlier
- **Solution**: *Experience replay*
  - Don't just train on fresh states
  - With a replay buffer $B$ holding up to N states, instead train on sampled batch: $b \sim B$
  - Default mode is uniform random sampling

# Background: Prioritized Experience Replay

- Replay buffer $B$ holding up to N states, train on sampled batch: $b \sim B$
- The probability that a transition is sampled depends on its “surprise factor” – how much the network learned from the transition
  - error $= Q(s,a) - \left(r + \gamma \max_{a'} Q(s',a')\right)$
  - $\text{priority} = |\text{error}| + \epsilon$
- But more difficult to store states and sample efficiently:
  - Replacing e.g. a ring buffer, authors recommend a *sum tree* achieving O(log n) sampling
- Used in Slither system

Schaul et al. 2015

# Background: Double DQN

- **Problem:** DQN can significantly overestimate Q values
- **Solution:** Double DQN
  - Use a target network, initialize parameters: $\theta^- = \theta$ at startup
  - At (long) regular intervals, copy $\theta^- \leftarrow \theta$ (to the target network from the online network)
  - Replace target values: $Y_t^{\text{DQN}} \equiv R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \theta_t)$

    with: $Y_t^{\text{DoubleQ}} \equiv R_{t+1} + \gamma\, Q(S_{t+1}, \operatorname{argmax}_a Q(S_{t+1}, a; \theta_t); \theta_t^-)$.
  - That is, *choose* the best action for state $S_{t+1}$ using online network, but *evaluate* it using the target network
  - Reduces overestimation and is implemented in Slither system

van Hasselt et al. 2015

# Slither System Overview

- Written in Java
- Browser control with Selenium WebDriver
- Neural network implemented in DL4J / ND4J
- Image capture / processing with OpenCV/JavaCV
- Trained with 16GB RAM system with CUDA on NVIDIA 980TI GPU

# Neural Network Architecture

- Similar to that of the original DQN
  - Larger networks take longer to train
- Input: [1x120x60]
- Convolutional Layers
  - Layer 1: 16 $8 \times 8$ filters, stride 4
  - Layer 2: 32 $4 \times 4$ filters, stride 2
- Layer 3: Fully connected, 256 outputs
- Output: numOutputs = numActions (explained later)

# Neural Network Architecture

- Why just one input image?
  - Using multiple images for each transition seems to require either more memory (preprocess once) or recurring computation (combine images at training time)
  - Neither is desirable
  - Direction of a worm = direction of head
- Why the different input size?
  - DQN used 84x84, apparently because their NN library preferred square images
  - Instead we have a similar image area and retain the same aspect ratio

# Neural Network Details

- Output layer has linear activation
- All other layers have ReLU (rectified linearity) activation
- Stochastic Gradient Descent
- Updater: Adam ($\alpha = 0.0005,\ \beta_1 = 0.9,\ \beta_2 = 0.999,\ \epsilon = 1 \times 10^{-8}$)
  - As recommended by Kingma and Ba (2014) except for $\alpha$
- Loss: Mean Squared Error
  - Good alternatives for further study include Huber Loss and LogCosh Loss

# Neural Network Details

- Weights are initialized according to He et al. (2015):
  - Issue: Xavier initializer assumes activations are linear (they aren't)
    - Can slow or prevent convergence
  - Proposed ReLU initializer ~ Normal with mean 0, stdev $\sqrt{\frac{2}{\hat{n}_l}}$ where $\hat{n}_l = k_l^2 d_l$, $l$ is the layer index, $k$ is the filter size, and $d$ is the number of filters

# Image Processing

- Screenshots of the game are taken rapidly using JavaCV's FFmpeg extension, as opposed to Java's Robot class
  - Robot class too slow for realtime inference
- Images are downscaled to $[120 \times 60]$
  - Raw images roughly $[1800 \times 900]$
- 3 Channels are reduced to 1
- Pixel values are normalized from $[0, 255]$ to $[0, 1]$

# Action Set Discretization

- **Problem**: Slither.io admits a nearly continuous set of inputs
  - $1800 \times 900 \times 2 = 3{,}240{,}000$ actions.
  - Recall numOutputs = numActions

# Action Set Discretization

## "Framing" Approach

- Preselect a frame of absolute coordinates
- Mouse "jumps" to coordinates
- Leads to jittery movements

## "Sliding" Approach

- $a_x, a_y \in \{-100, 0, +100\}, a_{\text{boost}} \in \{0,1\}$
- Mouse slides $\leq$ 100 pixels in any direction per timestep
- Encourages more commitment to one direction

# Gameplay Algorithm

- See pseudocode in report
- Agent acts under $\epsilon$- greedy policy
  - $\epsilon$ is linearly annealed from 1.0 to ~0.1 over thousands of timesteps
  - If $\mathrm{rand}() < \epsilon$: use random action; else: use best action
- Every ~ 5 timesteps: train on sample transition batch
  - Popular batch size = 32
- Every ~ 1000 timesteps: copy $\theta_t^- \leftarrow \theta_t$

# Evaluation

- Unfortunately, agent performs below human skill level
- Agent scores generally < 1000 points
- A small convenience sample of human scores was obtained by joining multiple game sessions
  - Observed score of players at position 1 of the leaderboard
  - N = 7
  - Mean = 34,157
  - Median = 32,099
  - Stdev: 16,640
- However YouTube videos allege scores of 100,000+

# Conclusion

- Agent performance was worse than one might hope
- Agent has not discovered the crucial “cut-off” and “encircling” strategies
- I discovered that Mnih et al. trained their Atari 2600 agent, for each game, on 10M frames
- Slither system can process ~ 12,500 frames / hour, requiring 30+ days to reach 10M
- However, learning efficiency (at least on Atari 2600) can be further improved as shown by Hessel et al. (2017) in the *Rainbow* agent
  - Using all of: double learning, prioritized replay, dueling networks, multi-step learning, distributional RL, and noisy nets
  - Independent DQN extensions can be combined for greater overall performance

# Conclusion

- A unique test-bed for RL algorithms was created
- The Slither system is distinct from the Atari literature as the game has highly limited observability
  - Agent only seems a small portion of the map at a time
- Further study: is DQN or a variant thereof well suited for partially observable games?
  - How distinct is flickering Pong from Slither.io?
  - See also: recurrent DQN (Hausknecht et al. 2015)

# Complete References


[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller, Playing Atari with Deep Reinforcement Learning, ArXiv13125602 Cs. (2013). http://arxiv.org/abs/1312.5602 (accessed May 19, 2018).

[2] T. Schaul, J. Quan, I. Antonoglou, D. Silver, Prioritized Experience Replay, ArXiv151105952 Cs. (2015). http://arxiv.org/abs/1511.05952 (accessed May 19, 2018).

[3] S. Zhang, R.S. Sutton, A Deeper Look at Experience Replay, ArXiv171201275 Cs. (2017). http://arxiv.org/abs/1712.01275 (accessed May 19, 2018).

[4] H. van Hasselt, A. Guez, D. Silver, Deep Reinforcement Learning with Double Q-learning, ArXiv150906461 Cs. (2015). http://arxiv.org/abs/1509.06461 (accessed May 19, 2018).

[5] D.P. Kingma, J. Ba, Adam: A Method for Stochastic Optimization, ArXiv14126980 Cs. (2014). http://arxiv.org/abs/1412.6980 (accessed May 20, 2018).

[6] K. He, X. Zhang, S. Ren, J. Sun, Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification, ArXiv150201852 Cs. (2015). http://arxiv.org/abs/1502.01852 (accessed May 20, 2018).

[7] M. Hausknecht, P. Stone, Deep Recurrent Q-Learning for Partially Observable MDPs, ArXiv150706527 Cs. (2015). http://arxiv.org/abs/1507.06527 (accessed May 20, 2018).

[8] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, D. Silver, Rainbow: Combining Improvements in Deep Reinforcement Learning, ArXiv171002298 Cs. (2017). http://arxiv.org/abs/1710.02298 (accessed May 20, 2018).