LEARNING TO PLAY MOBA GAMES USING NEURAL NETWORKS AND REINFORCEMENT LEARNING

RESEARCH PROJECT FOR THE DEGREE OF MASTER OF SCIENCE IN COMPUTER SCIENCE

JANINE WEBER

SUPERVISORS: ROLF FAGERBERG, MARCO CHIARANDINI

UNIVERSITY OF SOUTHERN DENMARK

JANUARY, 2017

OUTLINE

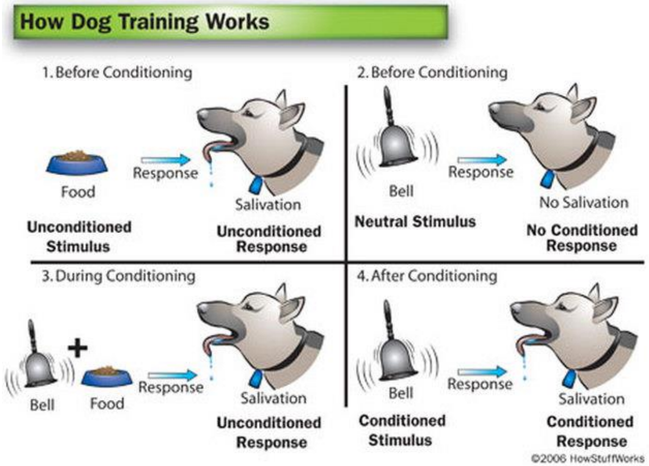
- Introduction
- Related Work
- Methods
- Application
- Experiments and Results
- Conclusion and Future Work

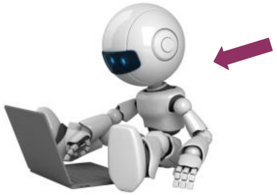
INTRODUCTION

MOTIVATION

- Market research (2014)
  - ~1.7 billion people playing video games
  - ~1.6 billion people active in traditional sports
- MOBA (multiplayer online battle arena)
  - Real-time strategy
  - Fast paced action
- Hard-coded AI

+





INTRODUCTION

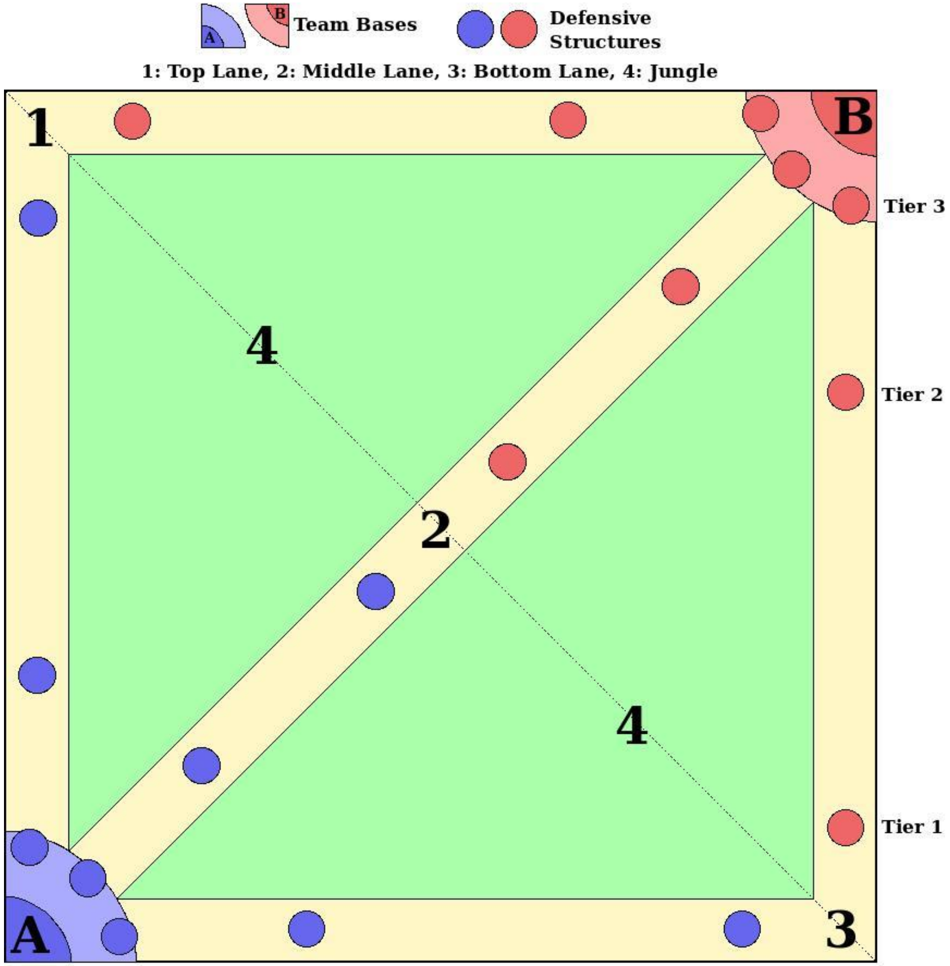
AIM OF THESIS

- Objective
  - Investigate if and to what degree agents can be taught to play a MOBA game.
- Methods
  - Implement a learning agent
  - Train through self-play
  - Test value of agent against original game AI
- Results
  - **First learning agent for a MOBA game**
  - Reinforcement learning to learning from experience
  - Neural networks to approximate win prediction
  - **Competitive with hard-coded Game AI (65+% win percent)**

INTRODUCTION

HEROES OF NEWERTH

- Game Objective
  - Entities
- Heroes of Newerth as Testing Grounds
  - Open code base
  - Access to game console
- Existing AI
  - Pre-defined behaviors (general and hero-specific)
  - Behavior priority ordering by computing utility values



OUTLINE

- ✓ Introduction
- Related Work
- Methods
- Application
- Experiments and Results
- Conclusion and Future Work

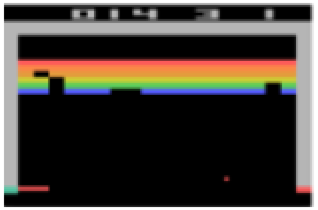
RELATED WORK

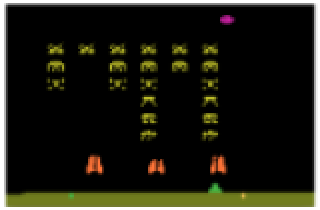
- *TD-Gammon* (Tesauro, 1995)

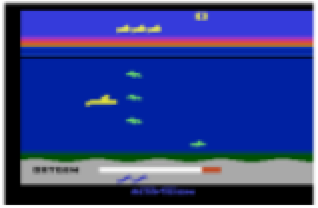
  - Neural Network to evaluate players chance of winning from current board state
  - Reinforcement Learning (self-play only)

- *AlphaGo* (Silver et al, 2016)
  - Deep Neural Networks
  - Tree Search using Monte-Carlo Simulation
  - Supervised & Reinforcement Learning (expert & self-play)

- Playing Atari Games (Mnih et al, 2013)
  - Convolutional Neural Network
    - Applied to 7 different games
  - Reinforcement Learning







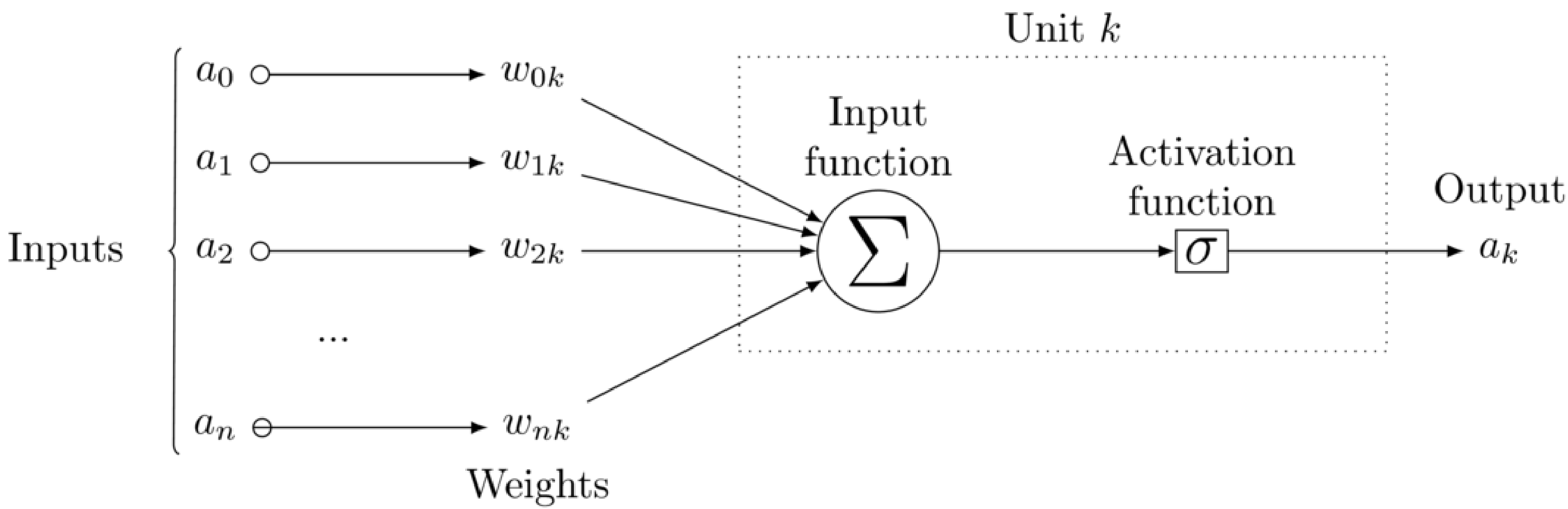
OUTLINE

- ✓ Introduction
- ✓ Related Work
- Methods
  - Neural Networks (Supervised Learning)
  - Reinforcement Learning
- Application
- Experiments and Results
- Conclusion and Future Work

METHODS

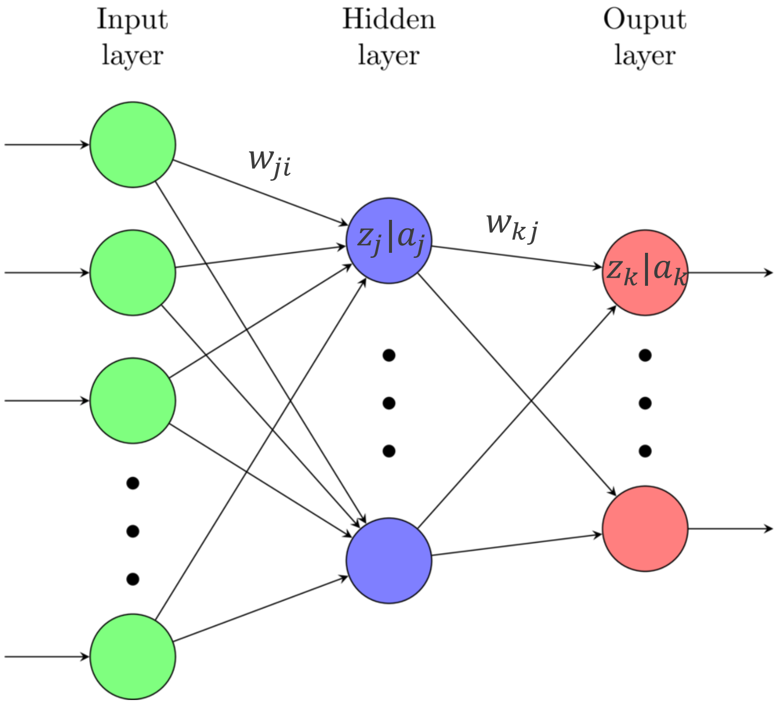
NEURAL NETWORKS

A Neural Network Neuron



METHODS

NEURAL NETWORKS



METHODS

NEURAL NETWORKS

- Back-propagation

1. Receive input $\vec{x}$ and target $\vec{y}$
2. Feed-forward

$$a_k = \sigma\left(\sum_j \sigma\left(\sum_i a_i\, w_{ji}\right) w_{kj}\right)$$

3. Calculate Prediction Error $(\vec{y} - \vec{a})$
4. Back-propagate Error

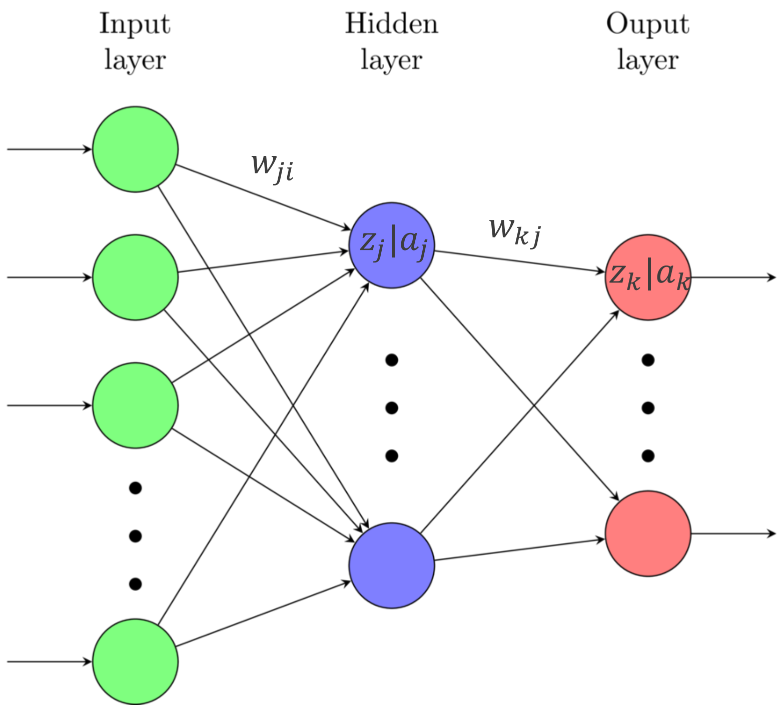
$$\delta_k = \sigma'(z_k)(y_k - a_k)$$

$$\delta_j = \sigma'(z_j) \sum_k \delta_k\, w_{kj}$$

5. Update Weights

$$w_{ih} \leftarrow w_{ih} - \alpha \frac{\partial E}{\partial w_{ih}}$$

$$\leftarrow w_{ih} - \alpha a_h \delta_i$$



METHODS

NEURAL NETWORKS

- Back-propagation
  1. Receive input $\vec{x}$ and target $\vec{y}$
  2. Feed-forward

$$a_k = \sigma\left(\sum_j \sigma\left(\sum_i a_i\, w_{ji}\right) w_{kj}\right)$$

  3. Calculate Prediction Error $(\vec{y} - \vec{a}$
  4. Back-propagate Error

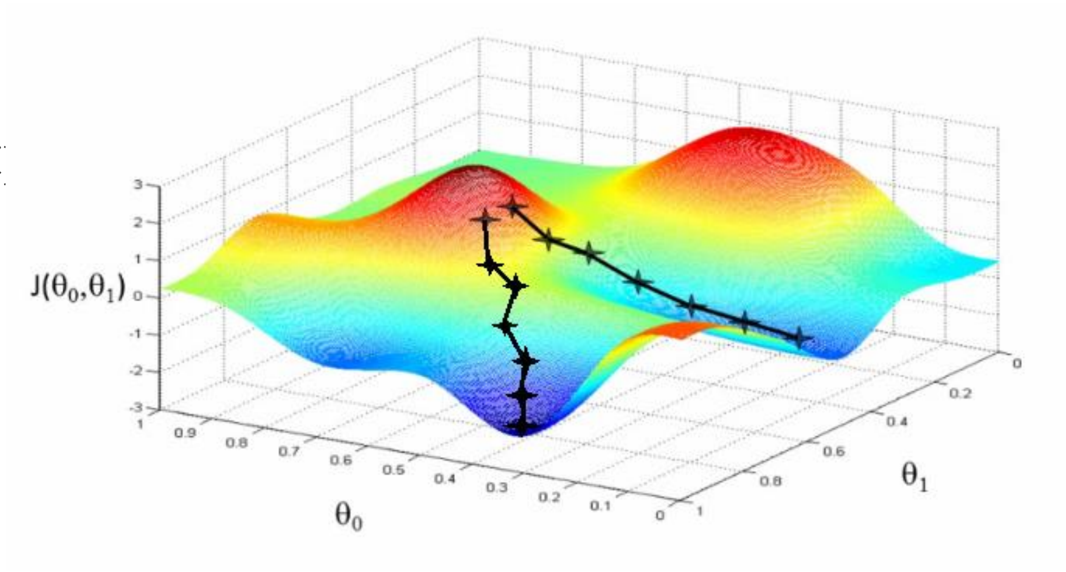
$$\delta_k = \sigma'(z_k)(y_k - a_k)$$

$$\delta_j = \sigma'(z_j)\sum_k \delta_k\, w_{kj}$$

  5. Update Weights

$$w_{ih} \leftarrow w_{ih} - \alpha \frac{\partial E}{\partial w_{ih}}$$

$$\leftarrow w_{ih} - \alpha a_h \delta_i$$



METHODS

NEURAL NETWORKS

- Back-propagation

1. Receive input $\vec{x}$ and target $\vec{y}$
2. Feed-forward

$$a_k = \sigma\left(\sum_j \sigma\left(\sum_i a_i\, w_{ji}\right) w_{kj}\right)$$

3. Calculate Prediction Error $(\vec{y} - \vec{a})$
4. Back-propagate Error

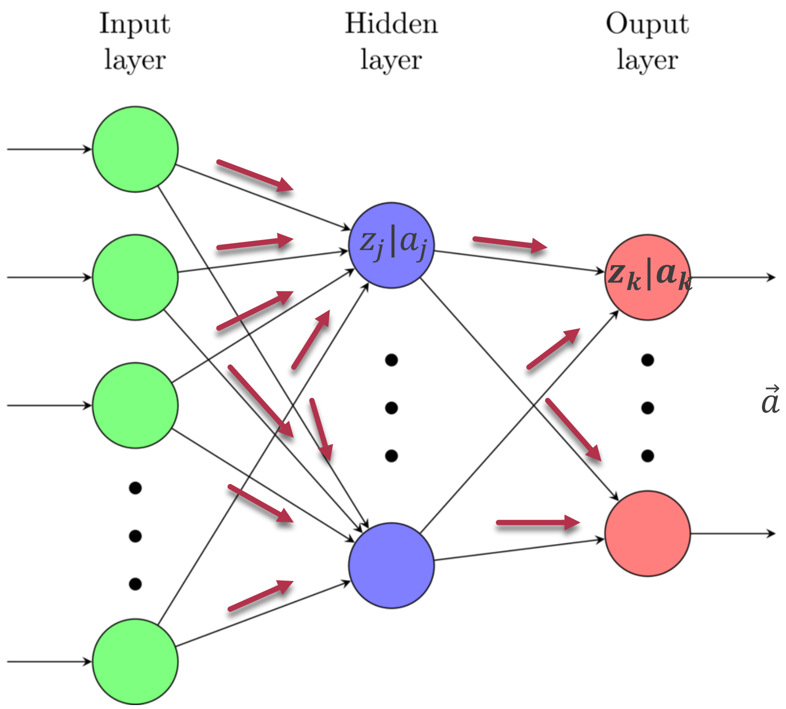
$$\delta_k = \sigma'(z_k)(y_k - a_k)$$

$$\delta_j = \sigma'(z_j) \sum_k \delta_k\, w_{kj}$$

5. Update Weights

$$w_{ih} \leftarrow w_{ih} - \alpha \frac{\partial E}{\partial w_{ih}}$$

$$\leftarrow w_{ih} - \alpha a_h \delta_i$$



METHODS

NEURAL NETWORKS

- Back-propagation
  1. Receive input $\vec{x}$ and target $\vec{y}$
  2. Feed-forward

     $$a_k = \sigma\left(\sum_j \sigma\left(\sum_i a_i\, w_{ji}\right) w_{kj}\right)$$

  3. Calculate Prediction Error $(\vec{y} - \vec{a})$
  4. Back-propagate Error

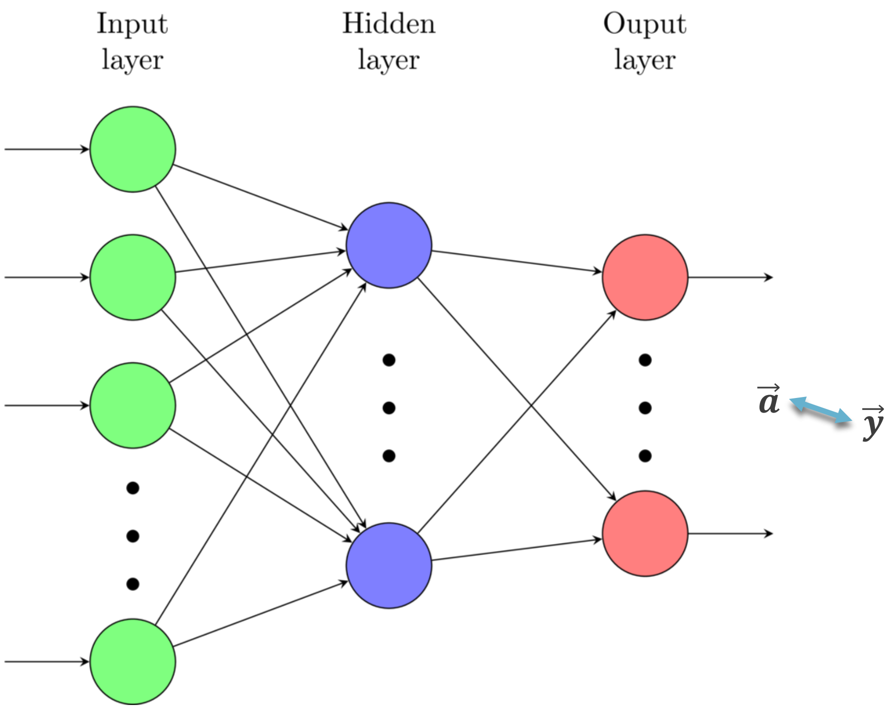
     $$\delta_k = \sigma'(z_k)(y_k - a_k)$$

     $$\delta_j = \sigma'(z_j) \sum_k \delta_k\, w_{kj}$$

  5. Update Weights

     $$w_{ih} \leftarrow w_{ih} - \alpha \frac{\partial E}{\partial w_{ih}}$$

     $$\leftarrow w_{ih} - \alpha a_h \delta_i$$



METHODS

NEURAL NETWORKS

- Back-propagation

1. Receive input $\vec{x}$ and target $\vec{y}$
2. Feed-forward

$$a_k = \sigma\left(\sum_j \sigma\left(\sum_i a_i w_{ji}\right) w_{kj}\right)$$

3. Calculate Prediction Error $(\vec{y} - \vec{a})$
4. Back-propagate Error

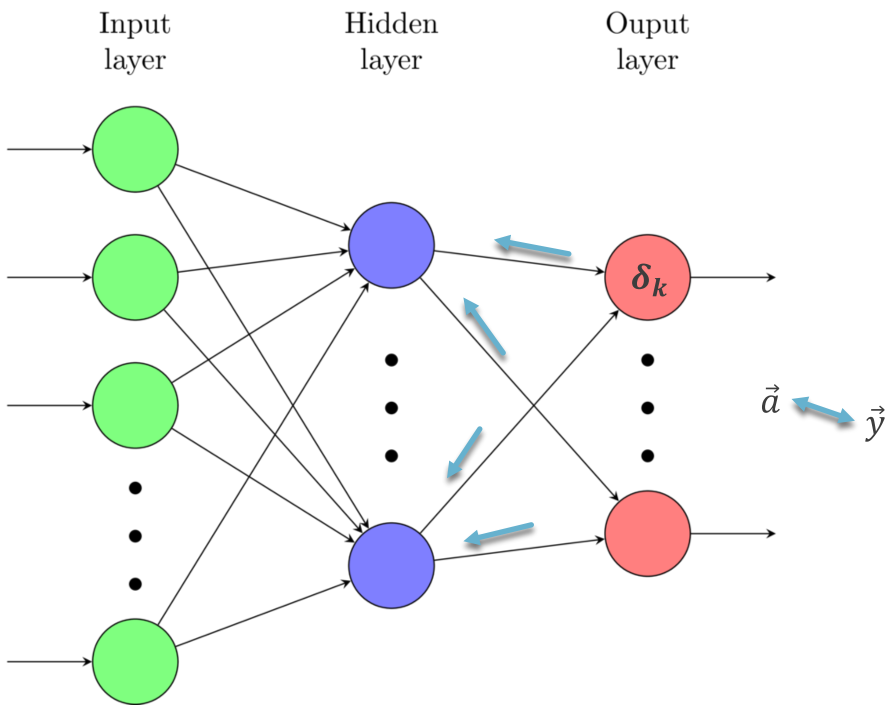
$$\delta_k = \sigma'(z_k)(y_k - a_k)$$

$$\delta_j = \sigma'(z_j) \sum_k \delta_k w_{kj}$$

5. Update Weights

$$w_{ih} \leftarrow w_{ih} - \alpha \frac{\partial E}{\partial w_{ih}}$$

$$\leftarrow w_{ih} - \alpha a_h \delta_i$$



METHODS

NEURAL NETWORKS

- Back-propagation

1. Receive input $\vec{x}$ and target $\vec{y}$
2. Feed-forward

$$a_k = \sigma\left(\sum_j \sigma\left(\sum_i a_i w_{ji}\right) w_{kj}\right)$$

3. Calculate Prediction Error $(\vec{y} - \vec{a})$
4. Back-propagate Error

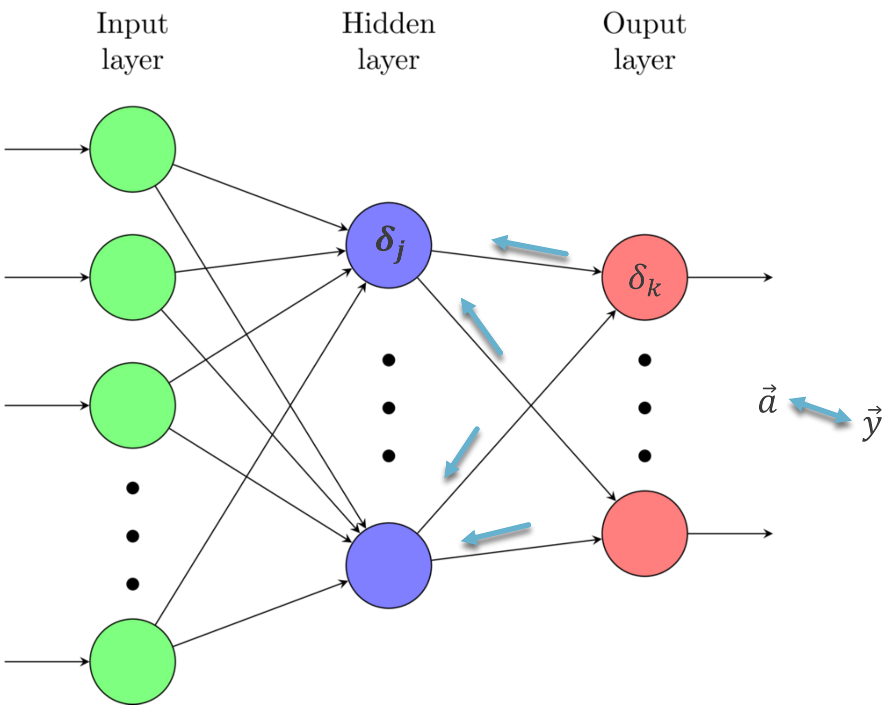
$$\delta_k = \sigma'(z_k)(y_k - a_k)$$

$$\delta_j = \sigma'(z_j) \sum_k \delta_k w_{kj}$$

5. Update Weights

$$w_{ih} \leftarrow w_{ih} - \alpha \frac{\partial E}{\partial w_{ih}}$$

$$\leftarrow w_{ih} - \alpha a_h \delta_i$$



METHODS

NEURAL NETWORKS

- Back-propagation

1. Receive input $\vec{x}$ and target $\vec{y}$
2. Feed-forward

$$a_k = \sigma\left(\sum_j \sigma\left(\sum_i a_i w_{ji}\right) w_{kj}\right)$$

3. Calculate Prediction Error $(\vec{y} - \vec{a})$
4. Back-propagate Error

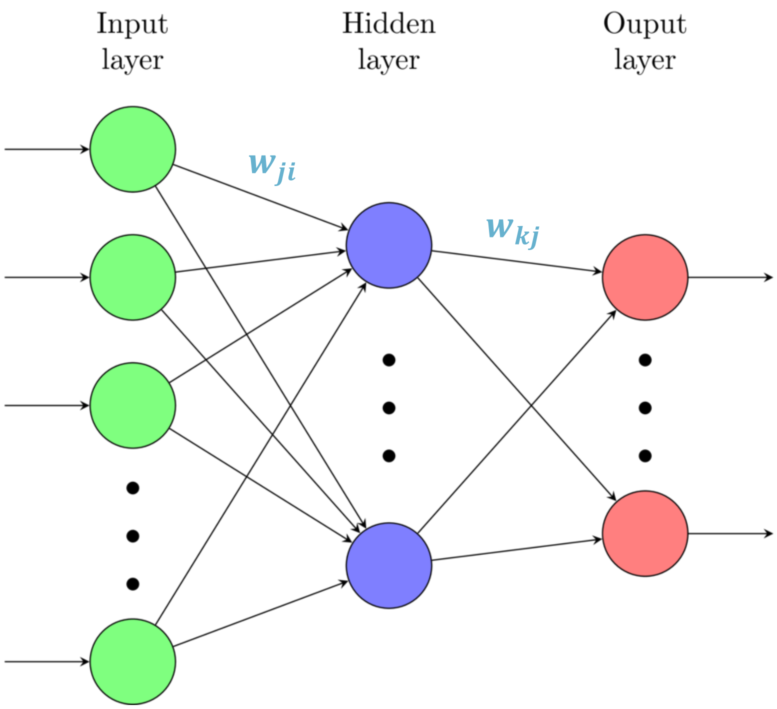
$$\delta_k = \sigma'(z_k)(y_k - a_k)$$

$$\delta_j = \sigma'(z_j) \sum_k \delta_k w_{kj}$$

5. Update Weights

$$w_{ih} \leftarrow w_{ih} - \alpha \frac{\partial E}{\partial w_{ih}}$$

$$\leftarrow w_{ih} - \alpha a_h \delta_i$$



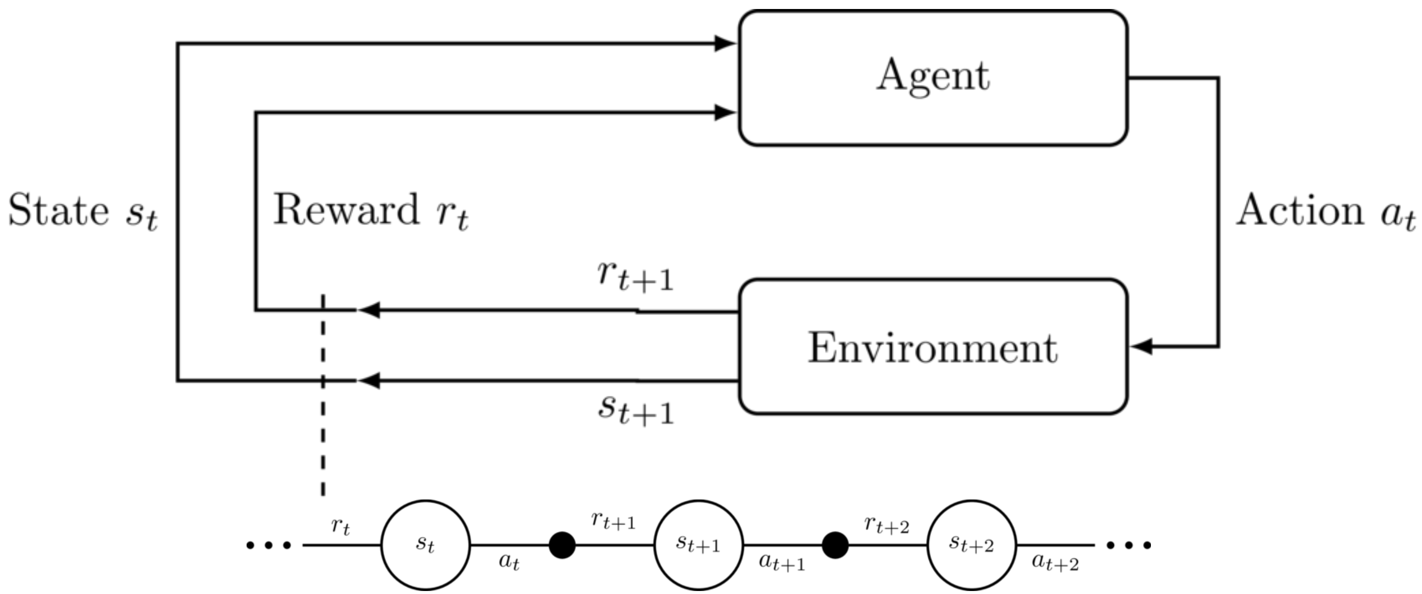
OUTLINE

- ✓ Introduction
- ✓ Related Work
- Methods
  - ✓ Neural Networks (Supervised Learning)
  - Reinforcement Learning
- Application
- Experiments and Results
- Conclusion and Future Work

METHODS

REINFORCEMENT LEARNING

Agent-Environment Interaction in Reinforcement Learning



METHODS

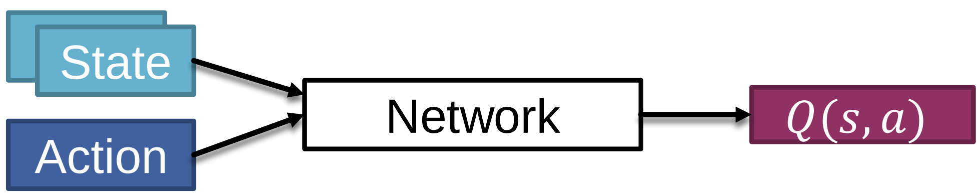
REINFORCEMENT LEARNING

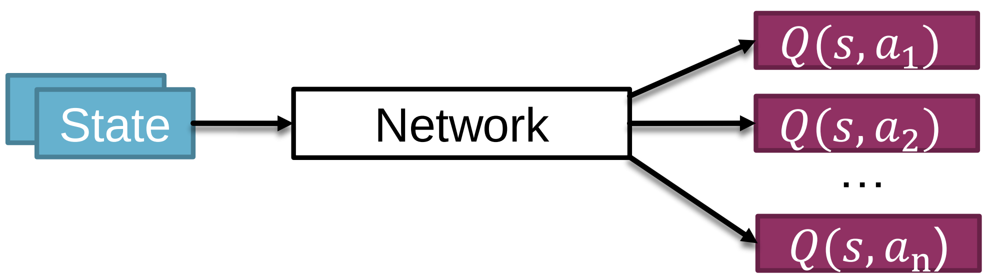
- Policies
  - Strategy which tells the agent which action is best to take in all situations
  - Mapping: *state* → *action*
- Reward Function
  - Defines goals
  - Mapping: *state* → *reward*
- Value Functions
  - Prediction of rewards
  - Mapping: *state* → *value*
  - Used to determine policy

REINFORCEMENT LEARNING

State-Value Function $V(s)$

Action-Value Function $Q(s, a)$



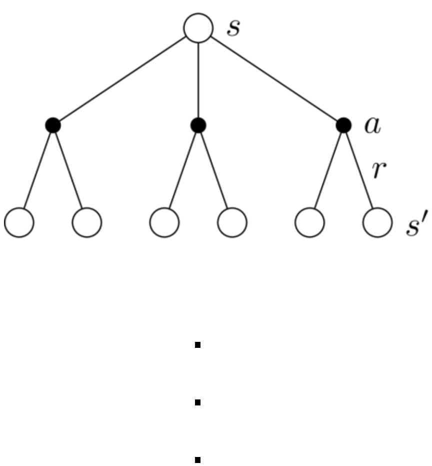


METHODS

REINFORCEMENT LEARNING

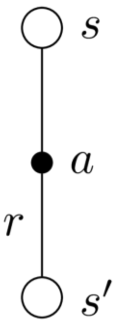
Value function approximation methods

Search Tree Evaluation



Monte-Carlo Methods

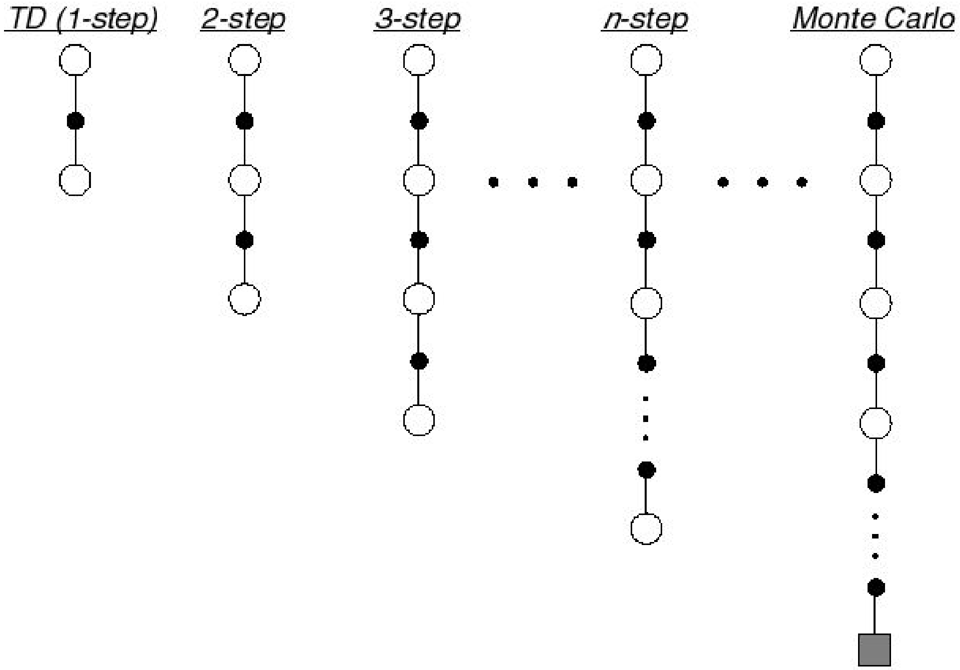
Temporal Difference, TD(0)



METHODS

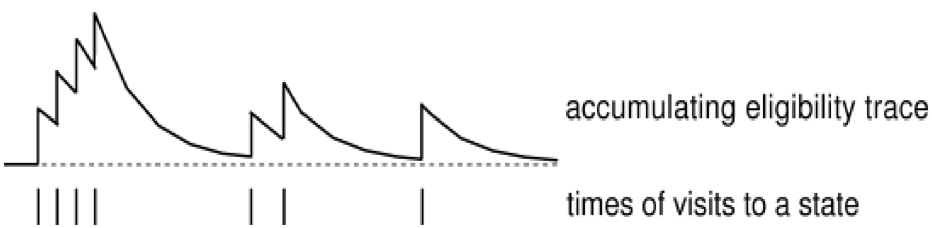
REINFORCEMENT LEARNING

- TD(λ)
  - Seeks to unify Monte-Carlo and Temporal Difference methods

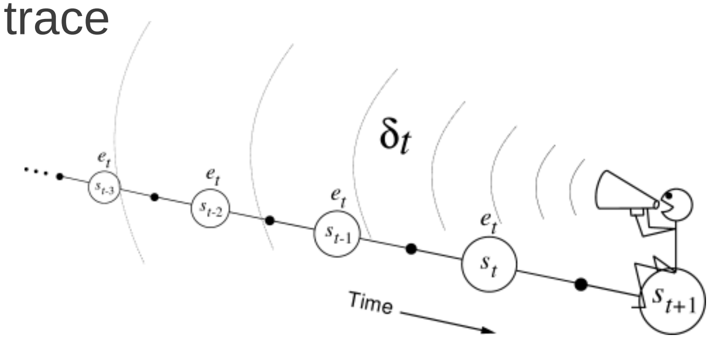


REINFORCEMENT LEARNING

- TD(λ)
  - TD error
    - Prediction error
  - Eligibility traces
    - Temporary records of event occurrences
    - λ-decay



1. Execute action
2. Calculate TD error (state prior vs post action)
3. Assign error backward in accordance to state's eligibility trace



4. Update

OUTLINE

- ✓ Introduction
- ✓ Related Work
- ✓ Methods
- Application
- Experiments and Results
- Conclusion and Future Work

APPLICATION

RL MODEL & NN

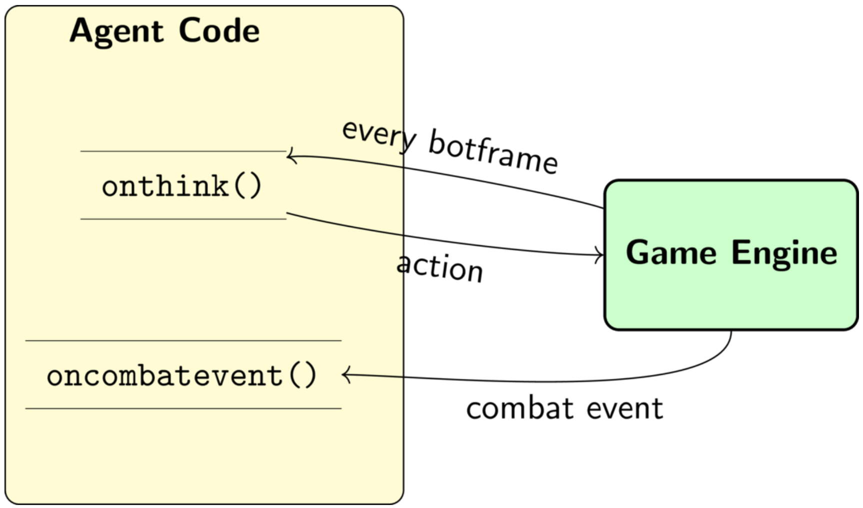
- Environment
  - **General**
    - Health
    - Mana
    - Auto-Attack Reader & Range
    - Distance (between heroes and to center)
    - Creep aggro & health
    - Creep score
    - Tower aggro/range & health
  - **Events**
    - Action Availability
    - Action inrange
    - Special events (weakening or strengthening)
    - Last hit & deny potential
    - Creep spawn
- Actions
  - 4 Abilities
  - Auto-attack (opponent, creep, tower)
  - Movement (Flee, Pursue, Hold)
- Rewards
  - Delayed, at the end
    - 0 (loss), 0.5 (draw), 1 (win)
- Value Function $Q(s, a)$
  - Given the current state *s* of the game, what is the predicted outcome of the match given that I take action *a*?

56 = 45 (*environment*) + **11** (*actions*) **inputs**
25 hidden units (*1 hidden layer*)
1 output

APPLICATION

IMPLEMENTATION

1. Pick an action
   - Random, ε-greedy
2. Execute action
3. Observe successor state
   - Increase eligibility trace
4. Observe reward
   - Prediction error, match outcome
5. Backpropagation & network weights update
   - Decay eligibility trace



GAME MECHANISMS

- Game Mechanisms
  - Instant vs channel actions
  - Temporary inability to act

0	Match Start
250	Action a_1 (Start & End), instantaneous
500	Action a_2 (Start), channeling time: 500
750	
1000	Action a_2 (End); Action a_3 (Start & End), instantaneous
1250	Hero Stunned (Start), duration: 750
1500	
1750	
2000	Hero Stunned (End); Action a_4 (Start), channeling time: 1000
2250	
$\dots$	
$\mathcal{M}$	Match End

OUTLINE

- ✓ Introduction
- ✓ Related Work
- ✓ Methods
- ✓ Application
- Experiments and Results
- Conclusion and Future Work

EXPERIMENTS AND RESULTS

EXPERIMENT STRUCTURE

- Limitations
- Stage 1
  - Is our agent learning?
  - Neural network structure?
- State 2
  - Can we detect incremental improvements?
  - First benchmark test!
- Stage 3
  - Extending the Game Scenario
  - What have we learned?
  - Benchmark test

EXPERIMENTS AND RESULTS

RESULTS

Stage 1

Are our agents learning?

Can we observe intelligence?

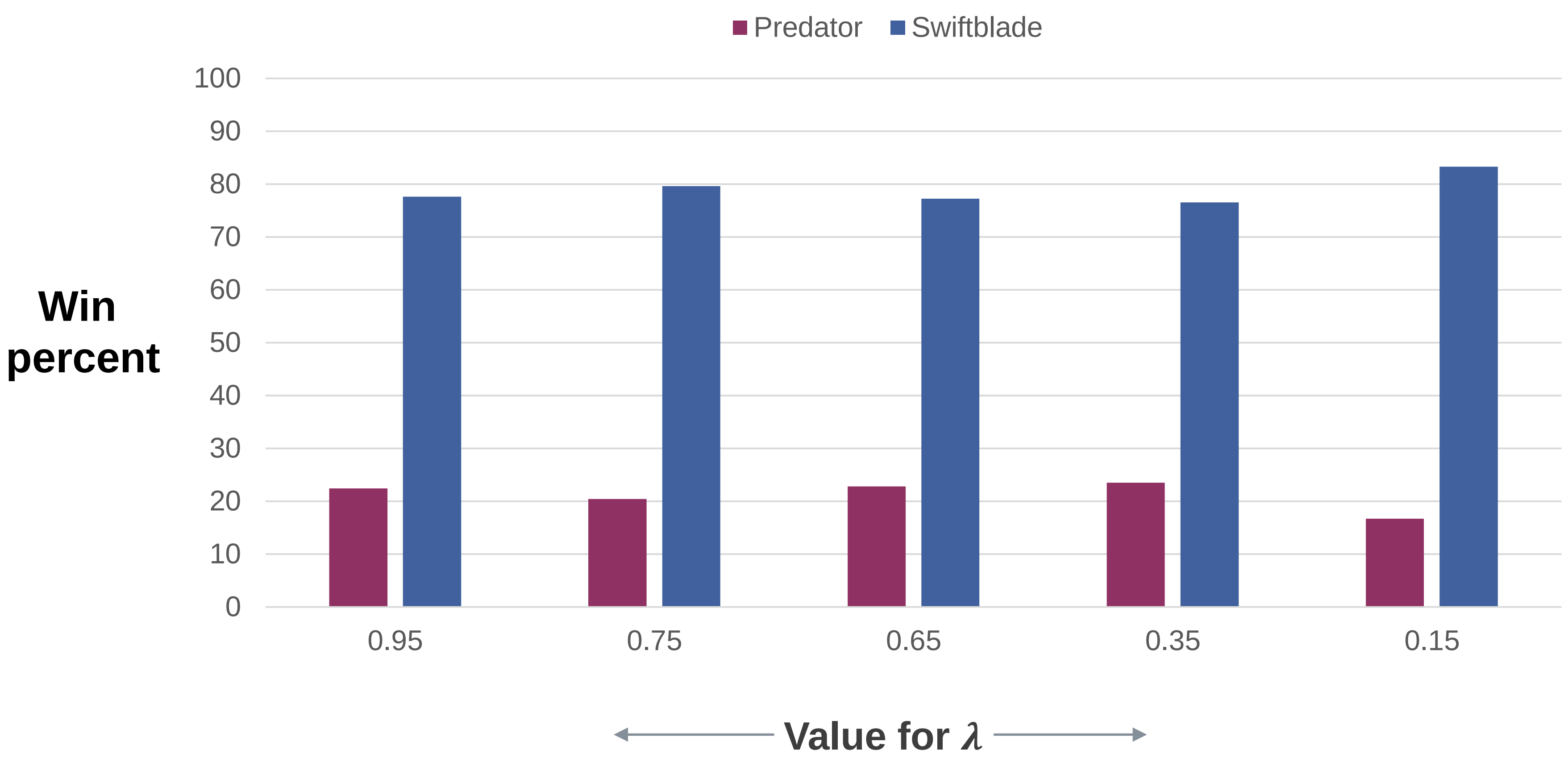
What should the network structure look like?

Self-Play: Predator vs Swiftblade

$(\alpha = 0.2, 40000 \text{ matches each})$

Self-Play: Predator vs Swiftblade

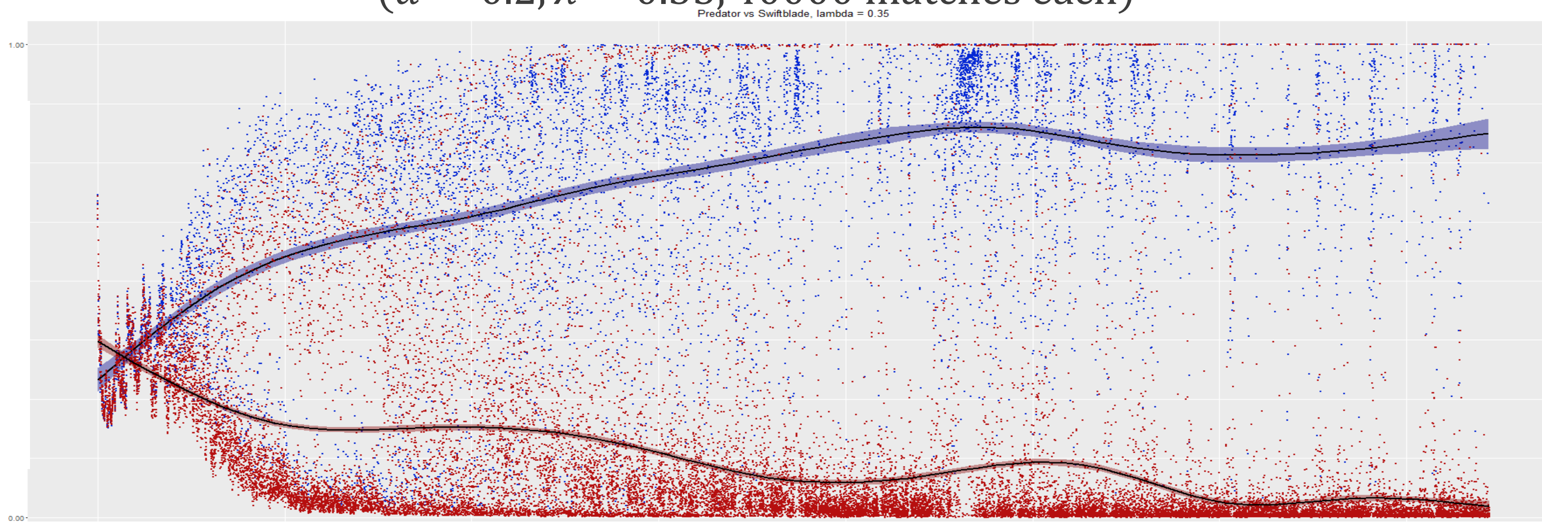
($\alpha = 0.2$, 40000 matches each)



Self-Play: Predator vs Swiftblade

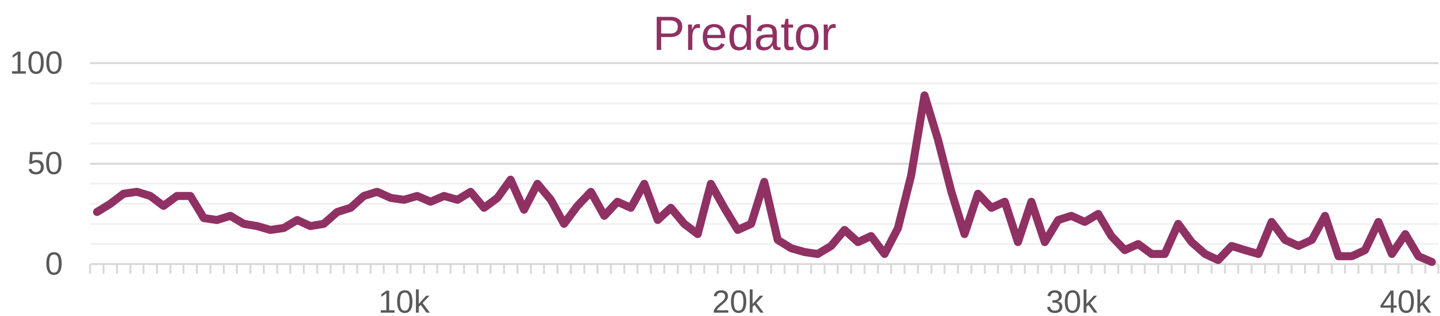
$(\alpha = 0.2, \lambda = 0.35, 40000 \text{ matches each})$

Win prediction of last executed action



Blue: wins; Red: losses

Match number

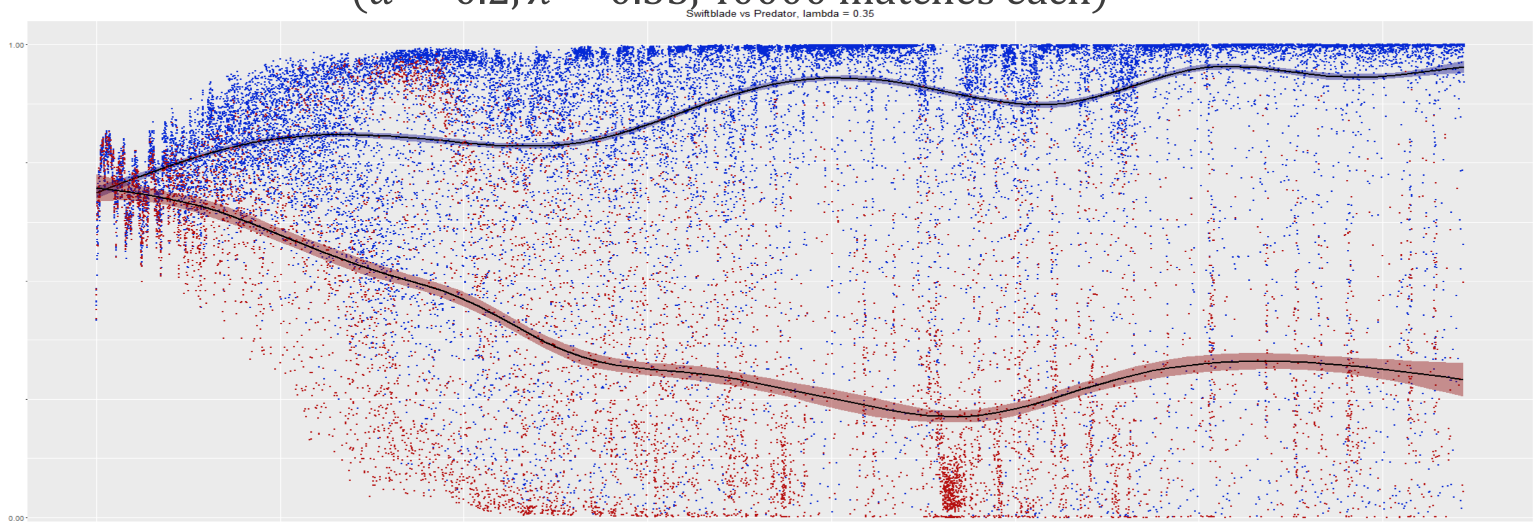


Averaging over win percent of 100 evenly sized groups of successive matches

Self-Play: Predator vs Swiftblade

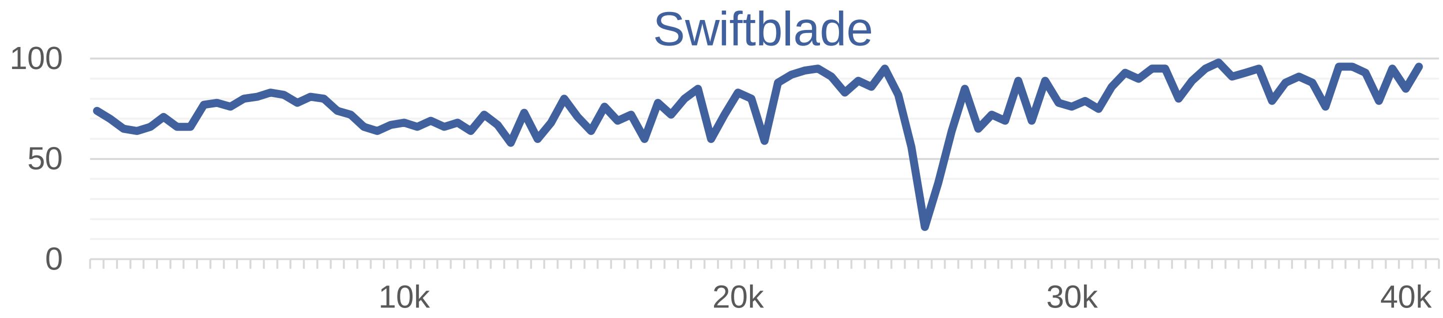
$(\alpha = 0.2, \lambda = 0.35, 40000 \text{ matches each})$

Win prediction of last executed action



Blue: wins; Red: losses

Match number

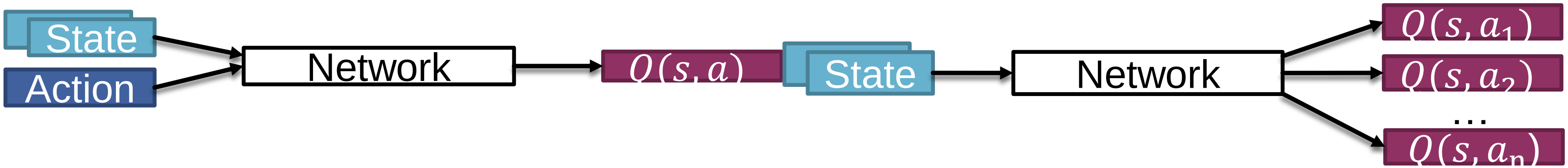


Averaging over win percent of 100 evenly sized groups of successive matches

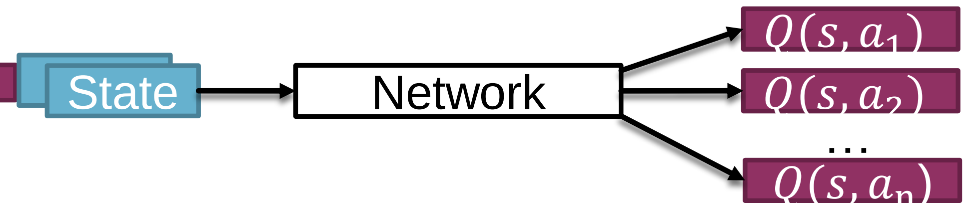
RESULTS

Balancing & Variations

PredatorA vs PredatorB $(\alpha = 0.2, 30000 \text{ matches each})$



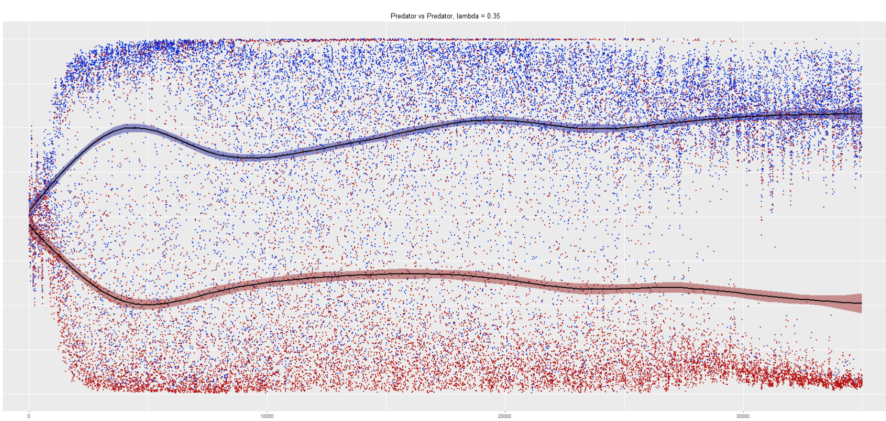
PredatorA vs PredatorB ($\alpha = 0.2$, 30000 matches each)

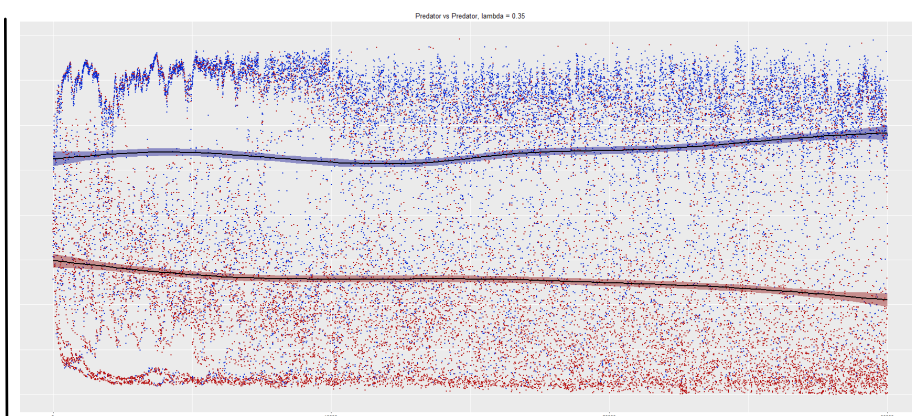


λ	Predator1	Predator2	Match Length
0.75	51.43%	48.57%	23.1s
0.35	55.3%	44.7%	22.2s

λ	Predator1	Predator2	Match Length
0.75	50.4%	49.6%	46.3s
0.35	50.5%	49.5%	39.4s

PredatorA, $\lambda = 0.35$





Blue: wins; Red: losses

***x* axis:** match number,

***y* axis:** win prediction of last action taken

RESULTS

Stage 2

Can we detect incremental improvement?

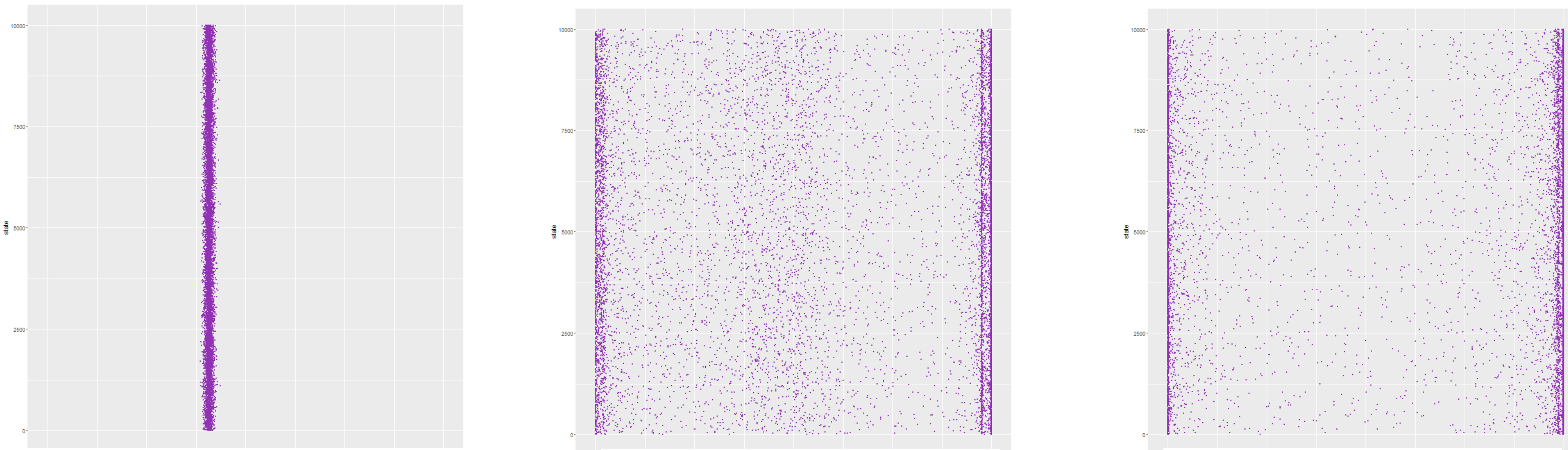
First benchmark test!

SuccubusA vs **SuccubusB** $(\alpha = 0.3, \lambda = 0.9, 60000 \text{ matches by selfplay})$

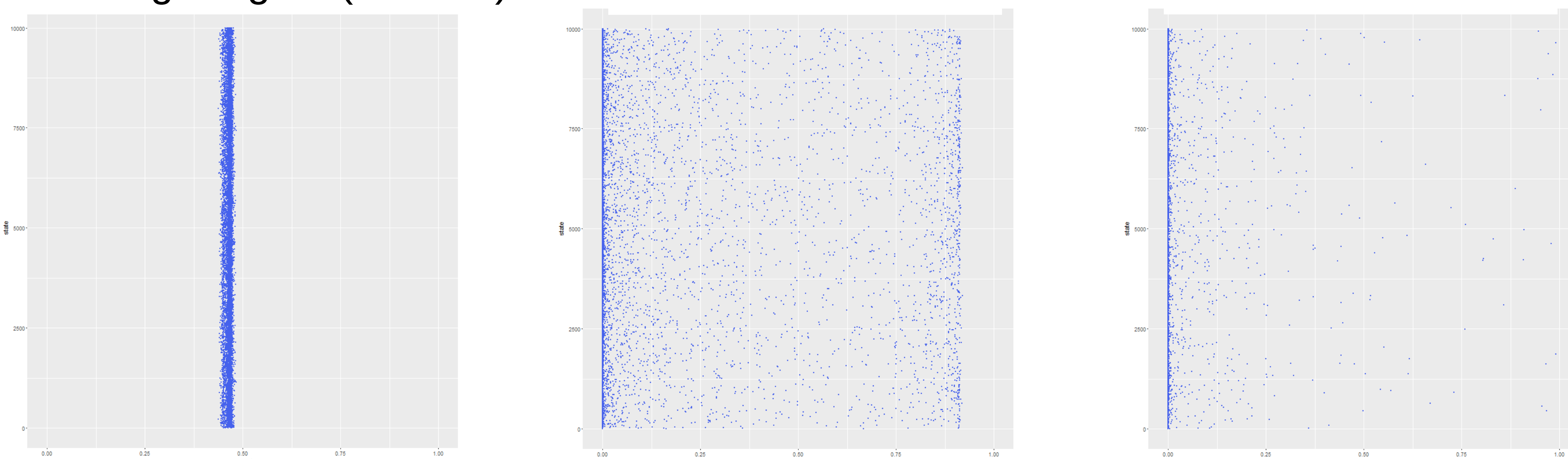
10000 states picked uniformly at random from 10000 random walk games.

SuccubusA vs **SuccubusB** $(\alpha = 0.3, \lambda = 0.9, 60000 \text{ matches by selfplay})$

10000 states picked uniformly at random from 10000 random walk games.



Starting weights (random) | After 30000 matches | After 60000 matches



x **axis:** network's prediction - y **axis:** random state number

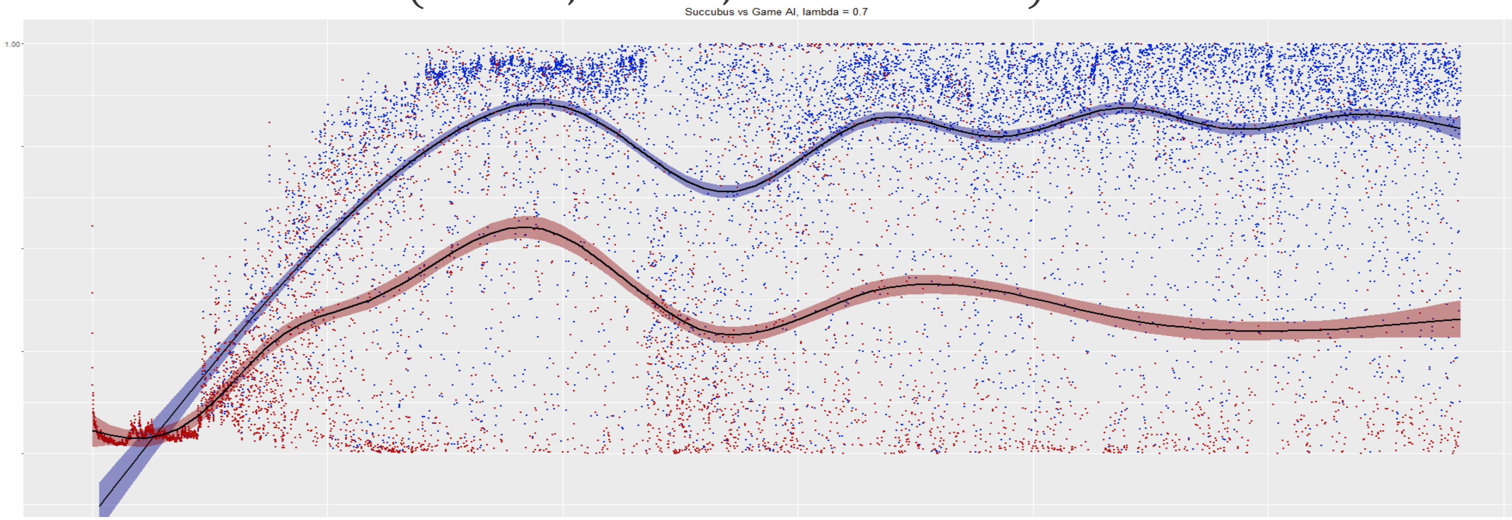
Succubus Learning Agent (starting with random weights) vs Game AI

$(\alpha = 0.3, \lambda = 0.7, 15000 \text{ matches})$

Succubus Learning Agent (starting with random weights) vs Game AI

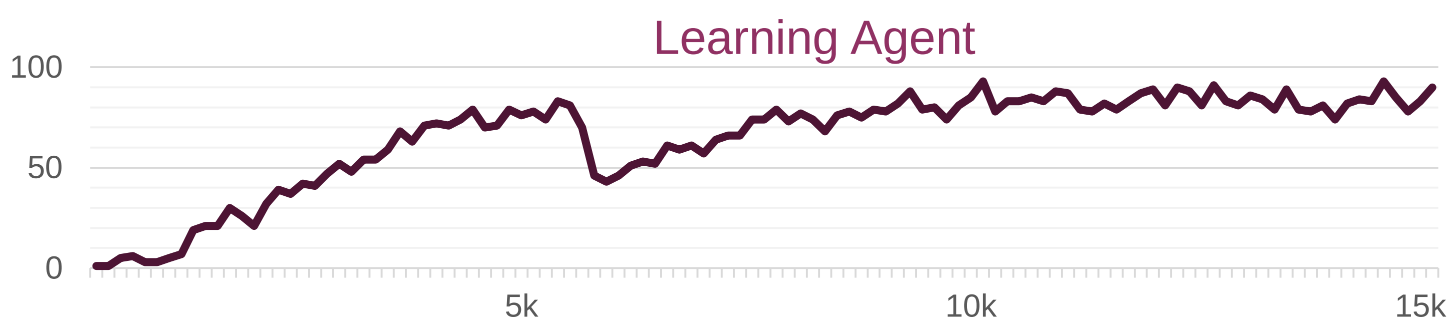
$(\alpha = 0.3, \lambda = 0.7, 15000 \text{ matches})$

Win prediction of last executed action



Blue: wins; Red: losses

Match number



Averaging over win percent of 100 evenly sized groups of successive matches

EXPERIMENTS AND RESULTS

RESULTS

Stage 3

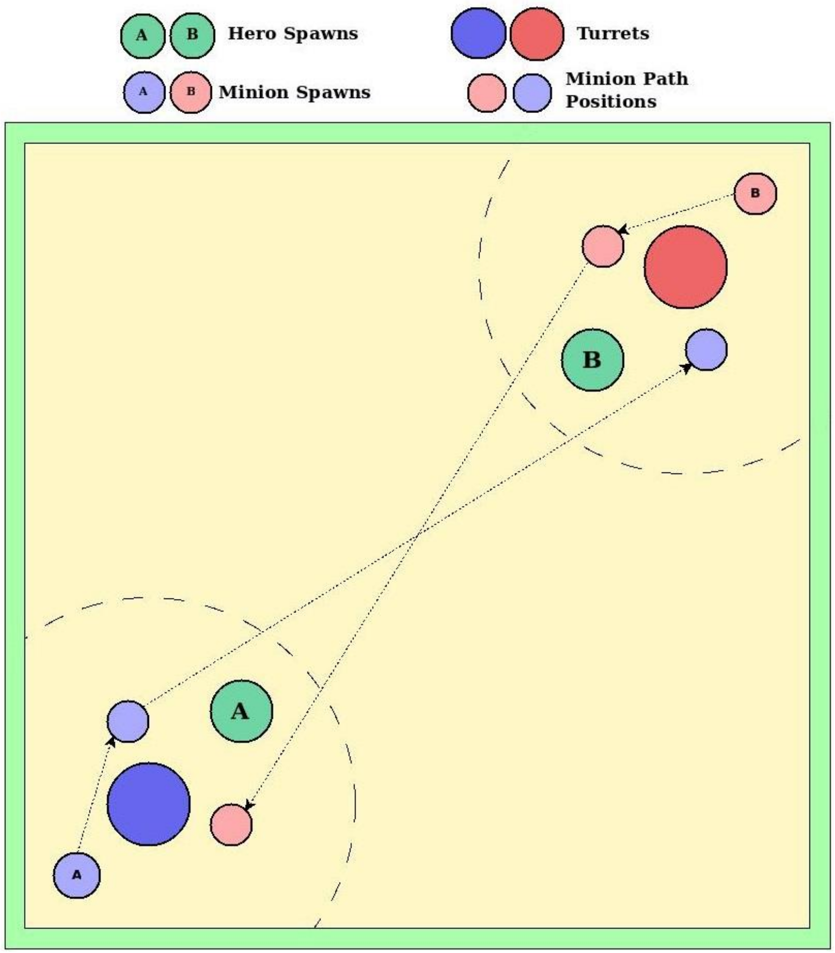
Increasing the difficulty.

Applying what we have learned.

EXPERIMENTS AND RESULTS

EXPERIMENTAL SETUP

- Win Conditions
  - Hero dies
  - 15 creep kills
  - Tower is destroyed



EXPERIMENTS AND RESULTS

EXPERIMENTAL SETUP

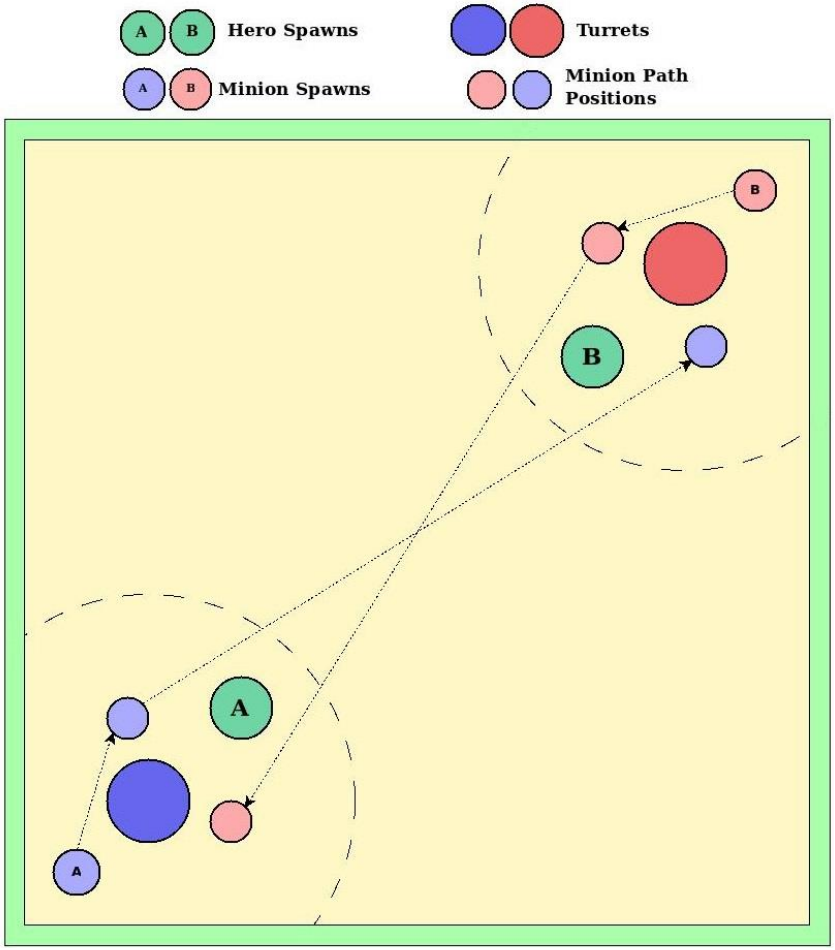
- The Hero: Succubus

Damage Reduction

Damage + Heal Self

Damage + Sleep

Damage + Stun + Mana Drain

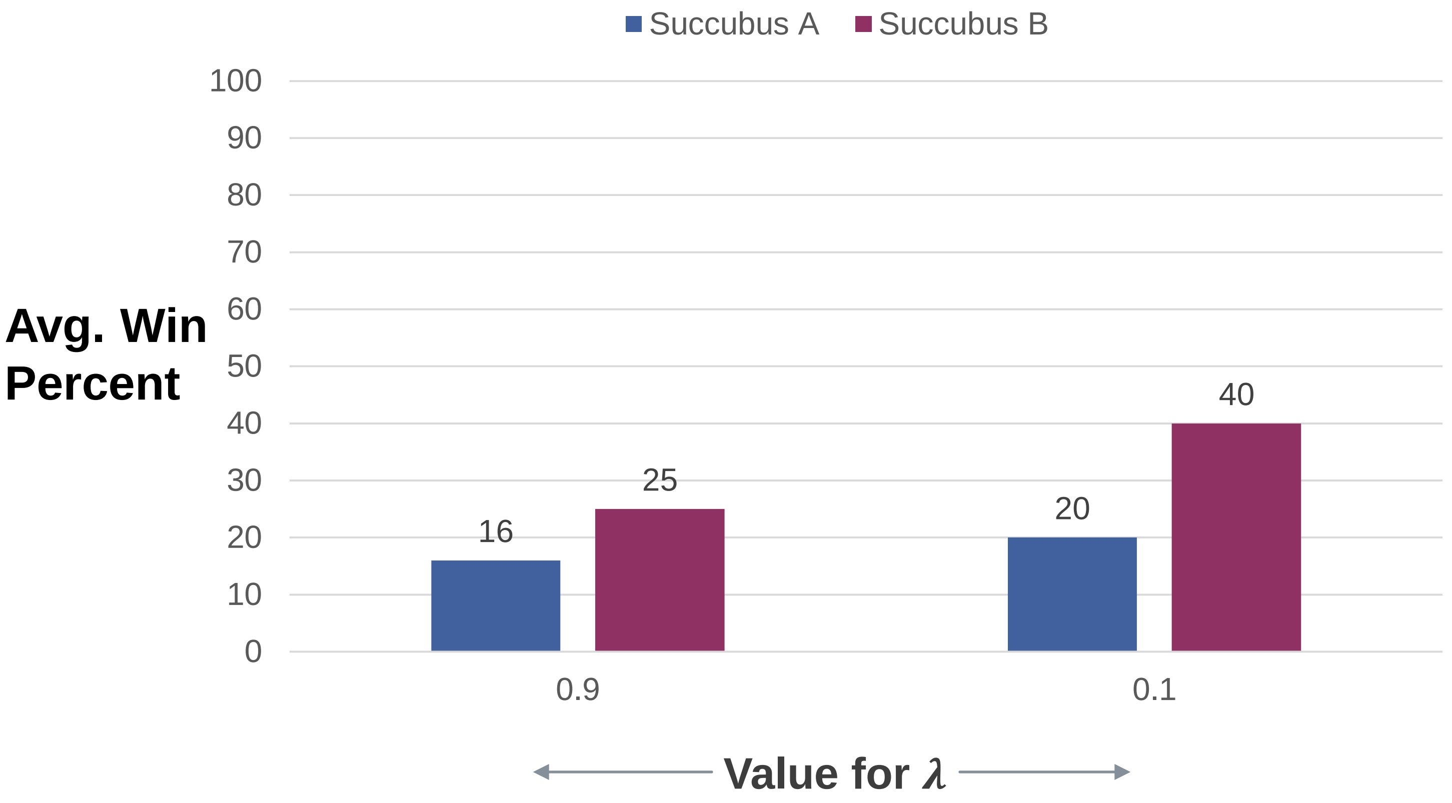


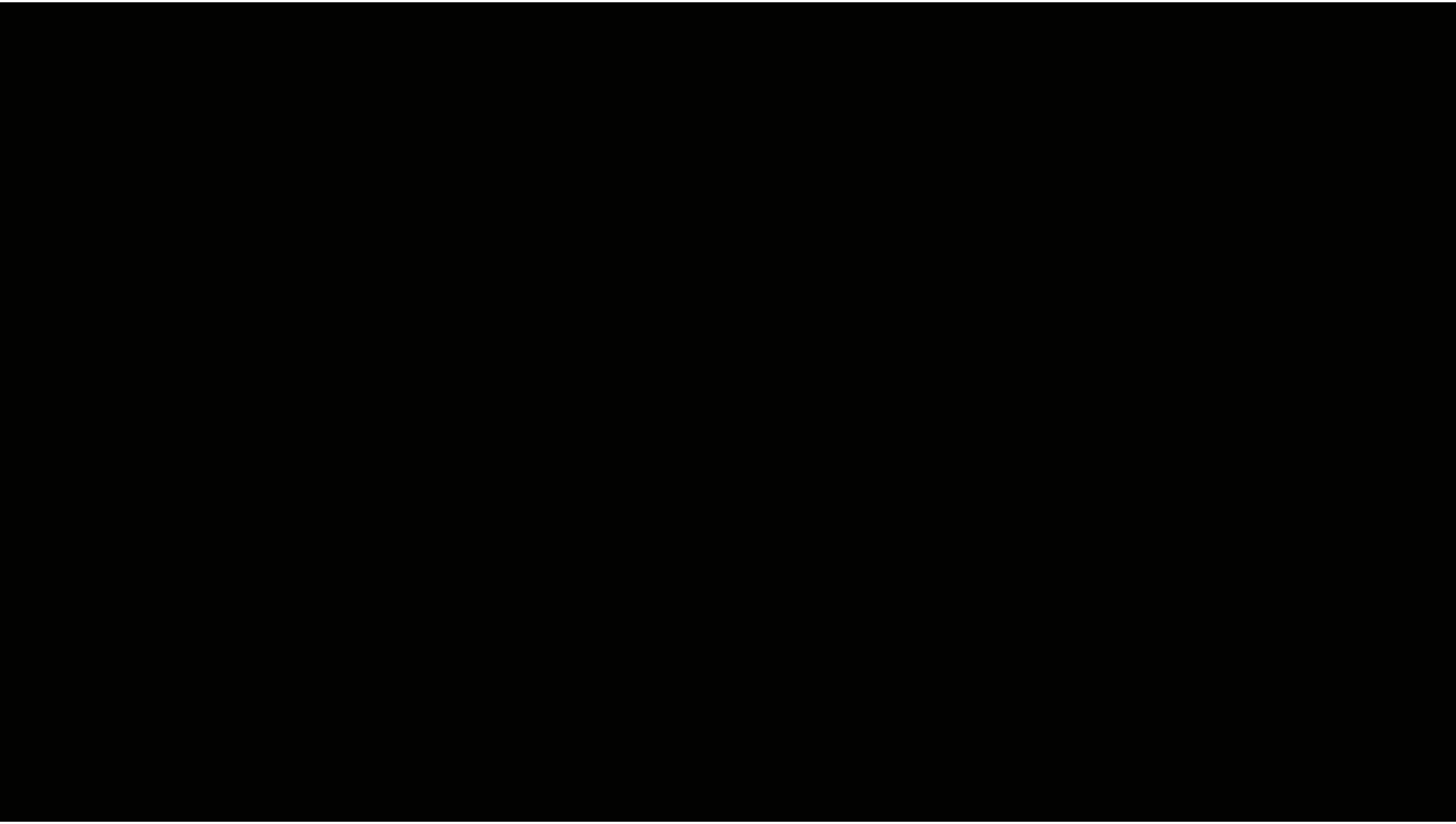
Learning Agent (using pretrained weights after 50000 self-play matches) vs Game AI

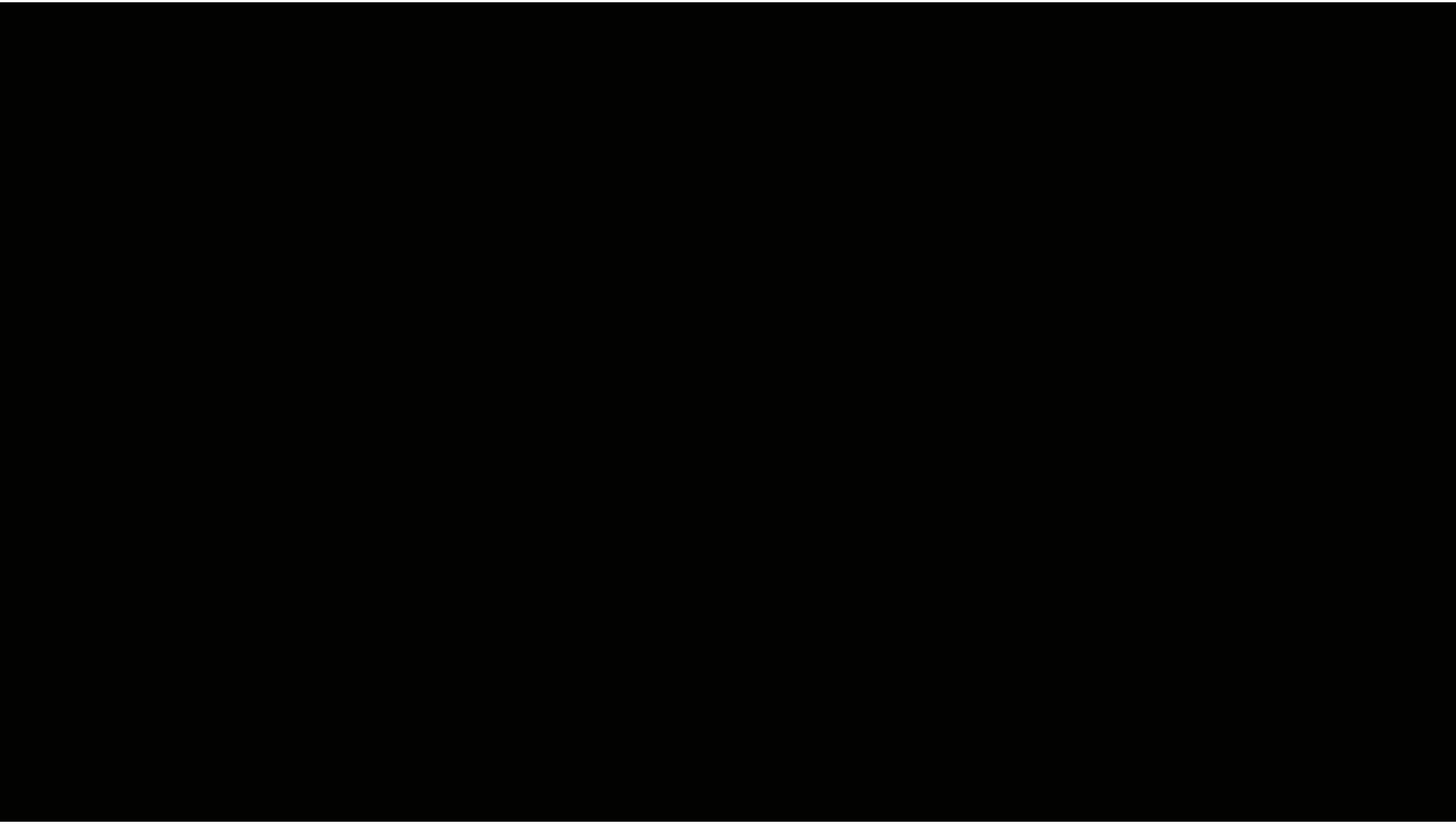
$(\alpha = 0.3, 100 \text{ matches each})$

Learning Agent (using pretrained weights after 50000 self-play matches) vs Game AI

$(\alpha = 0.3, 100 \text{ matches each})$







RESULTS

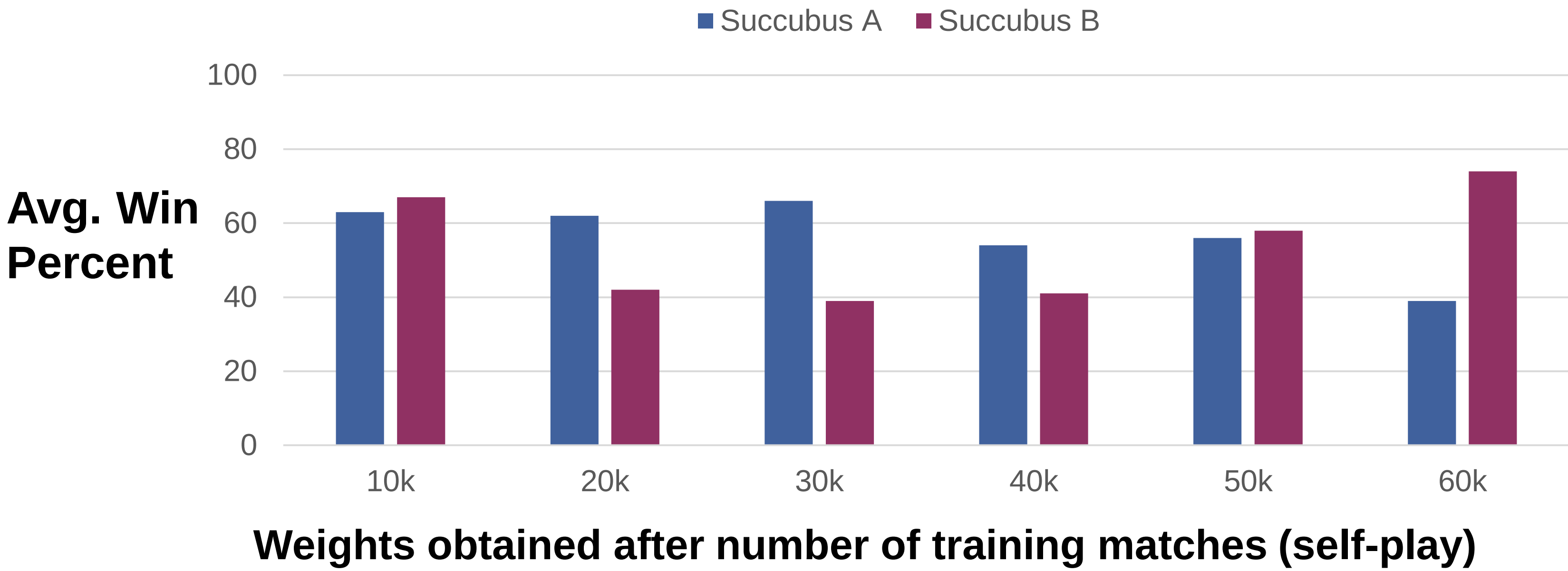
Adjustments

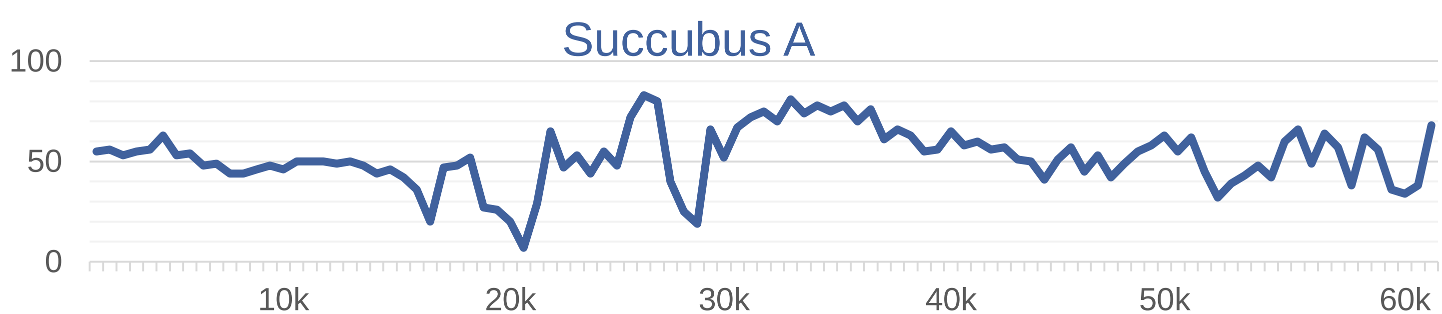
Learning Agent (using pertained weights) vs Game AI

$(\alpha = 0.3, \lambda = 0.9, 100 \text{ matches each})$

Learning Agent (using pertained weights) vs Game AI

$(\alpha = 0.3, \lambda = 0.9, 100 \text{ matches each})$

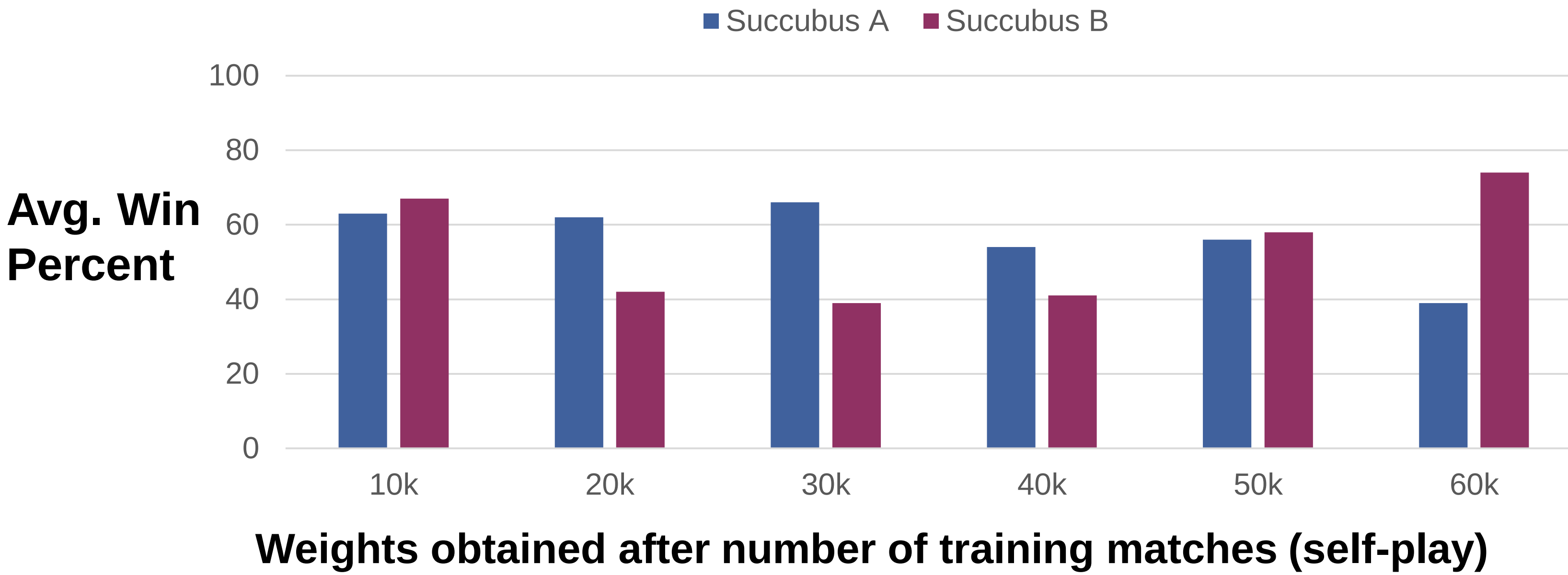


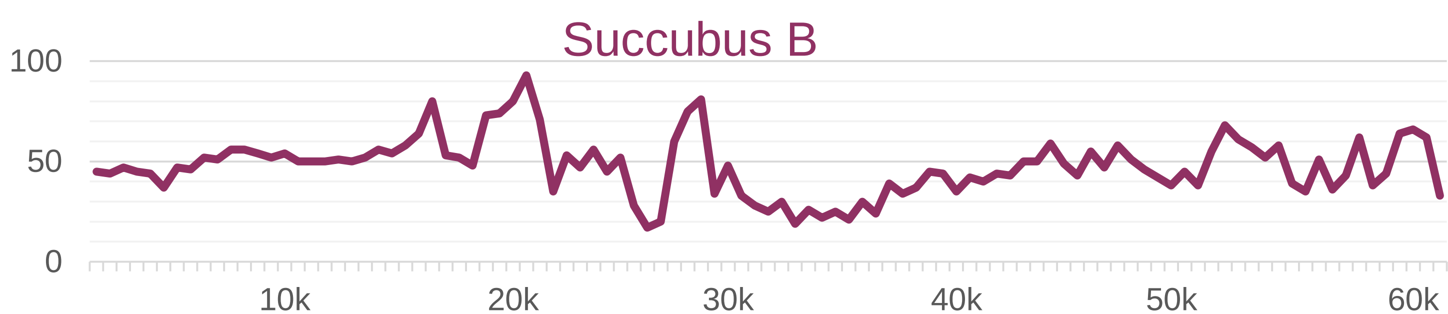


Averaging over win percent of 100 evenly sized groups of successive matches (self-play)

Learning Agent (using pertained weights) vs Game AI

$(\alpha = 0.3, \lambda = 0.9, 100 \text{ matches each})$





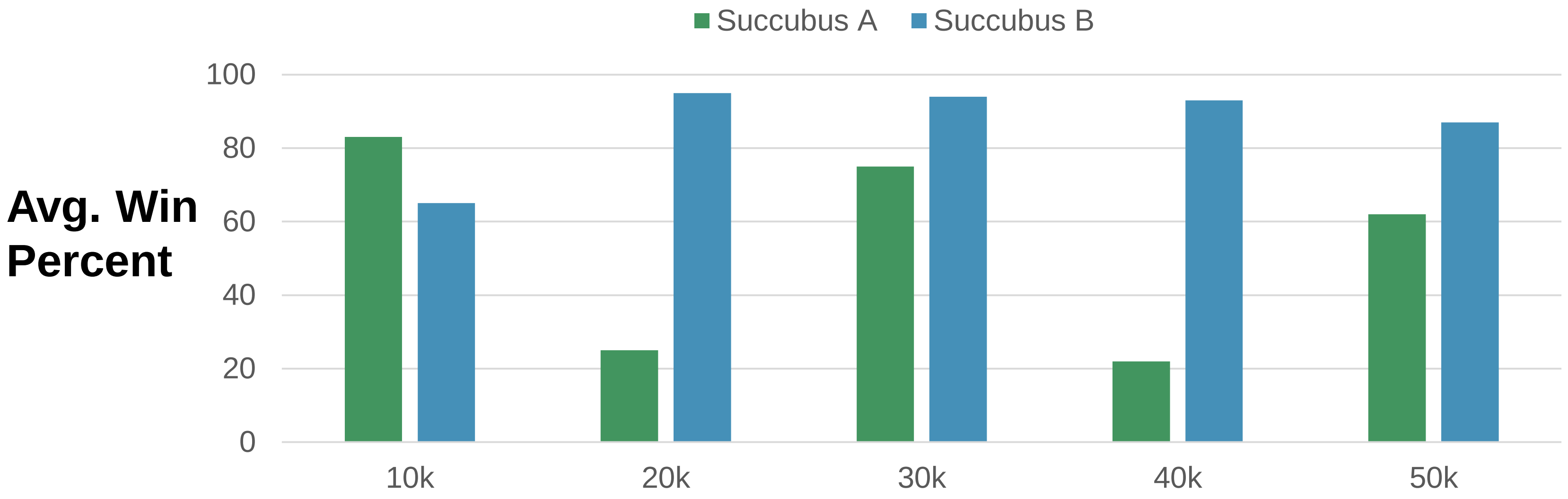
Averaging over win percent of 100 evenly sized groups of successive matches (self-play)

Learning Agent (using pertained weights) vs Game AI

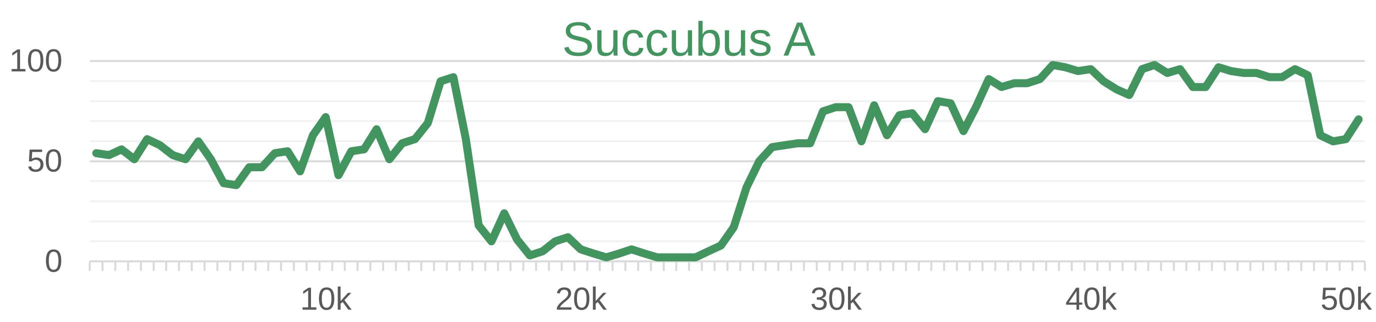
$(\alpha = 0.3, \lambda = 0.1, 100 \text{ matches each})$

Learning Agent (using pertained weights) vs Game AI

$(\alpha = 0.3, \lambda = 0.1, 100 \text{ matches each})$



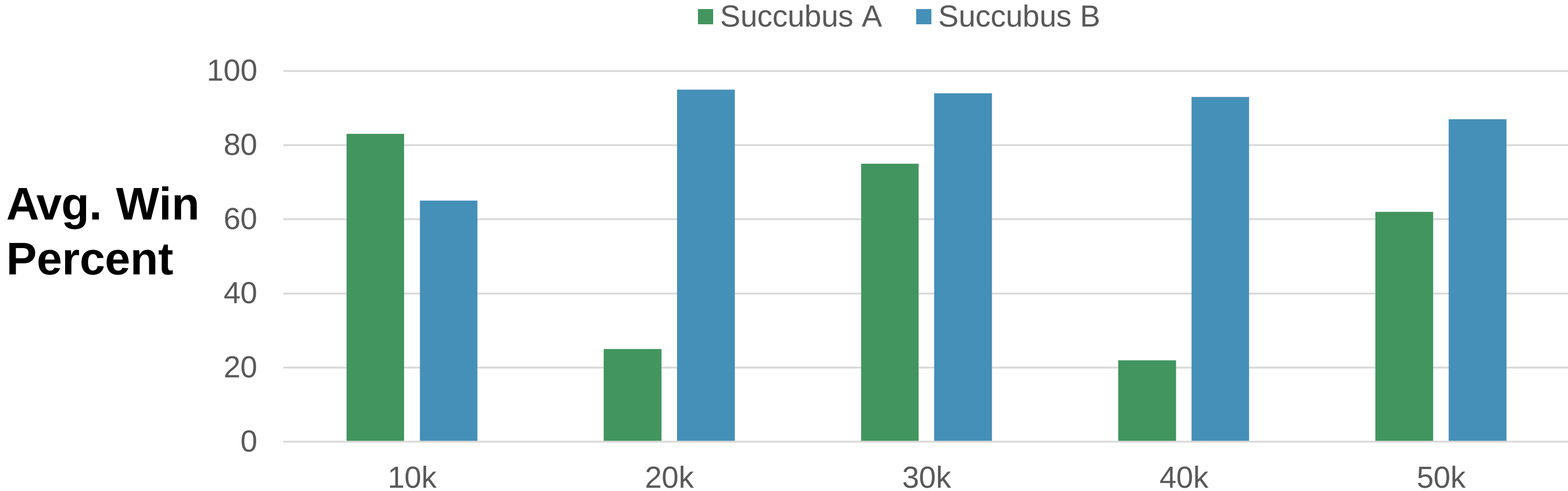
Weights obtained after number of training matches (self-play)



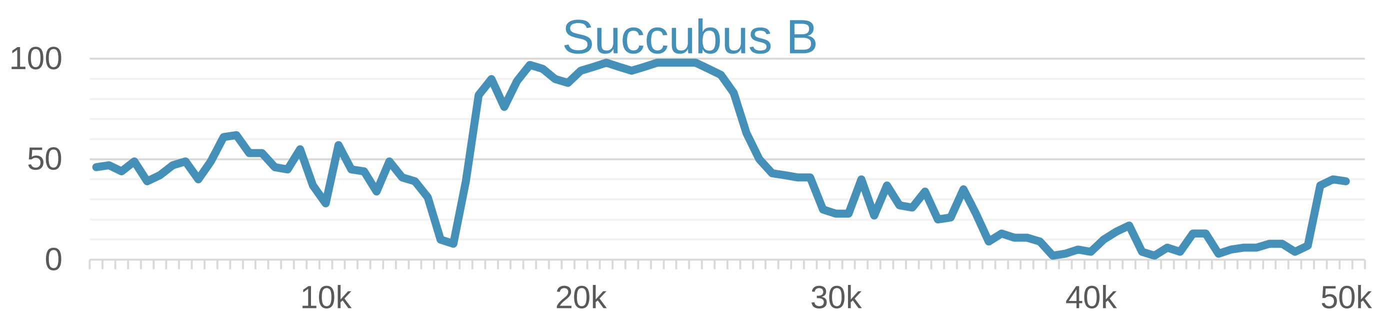
Averaging over win percent of 100 evenly sized groups of successive matches (self-play)

Learning Agent (using pertained weights) vs Game AI

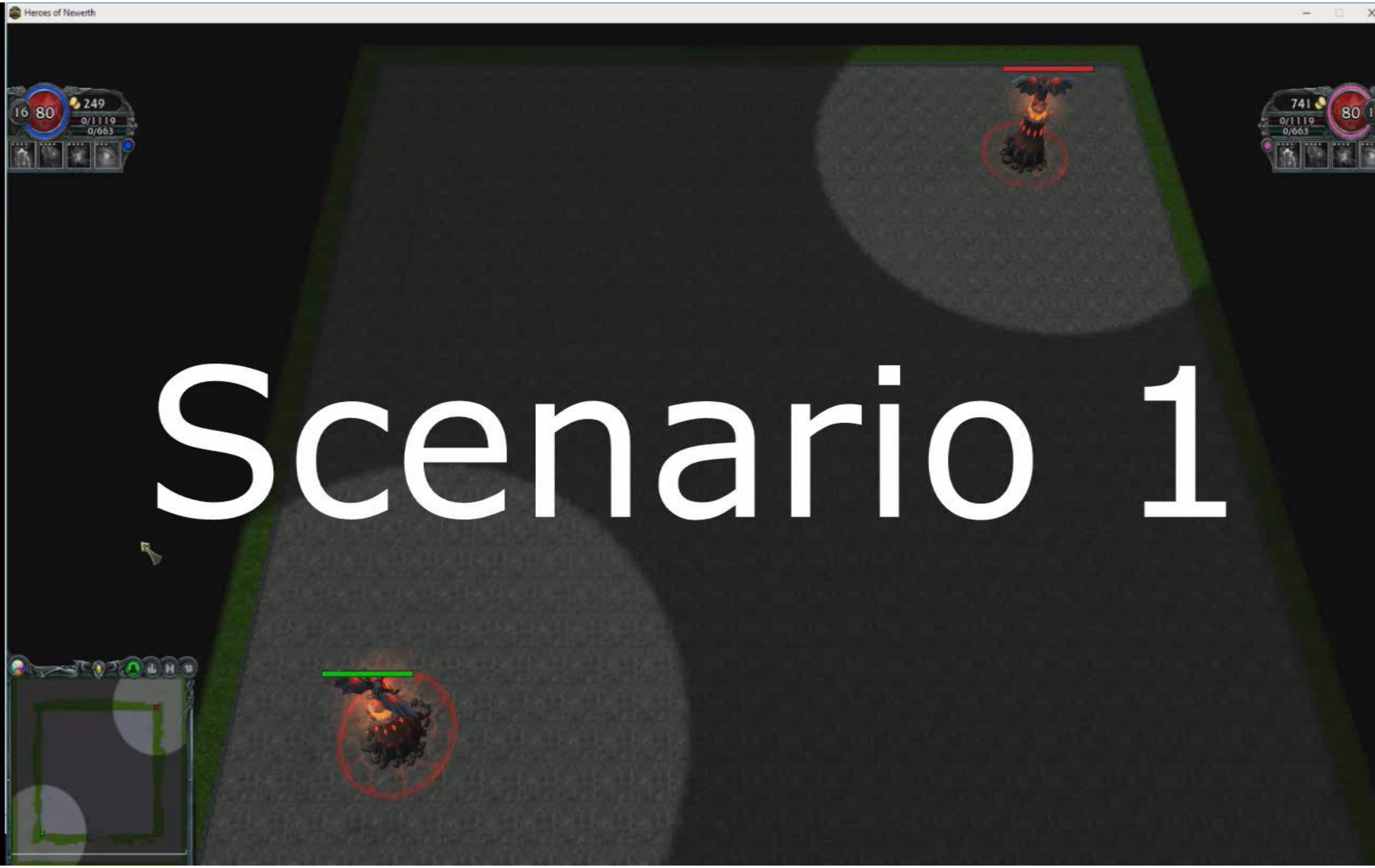
$(\alpha = 0.3, \lambda = 0.1, 100 \text{ matches each})$

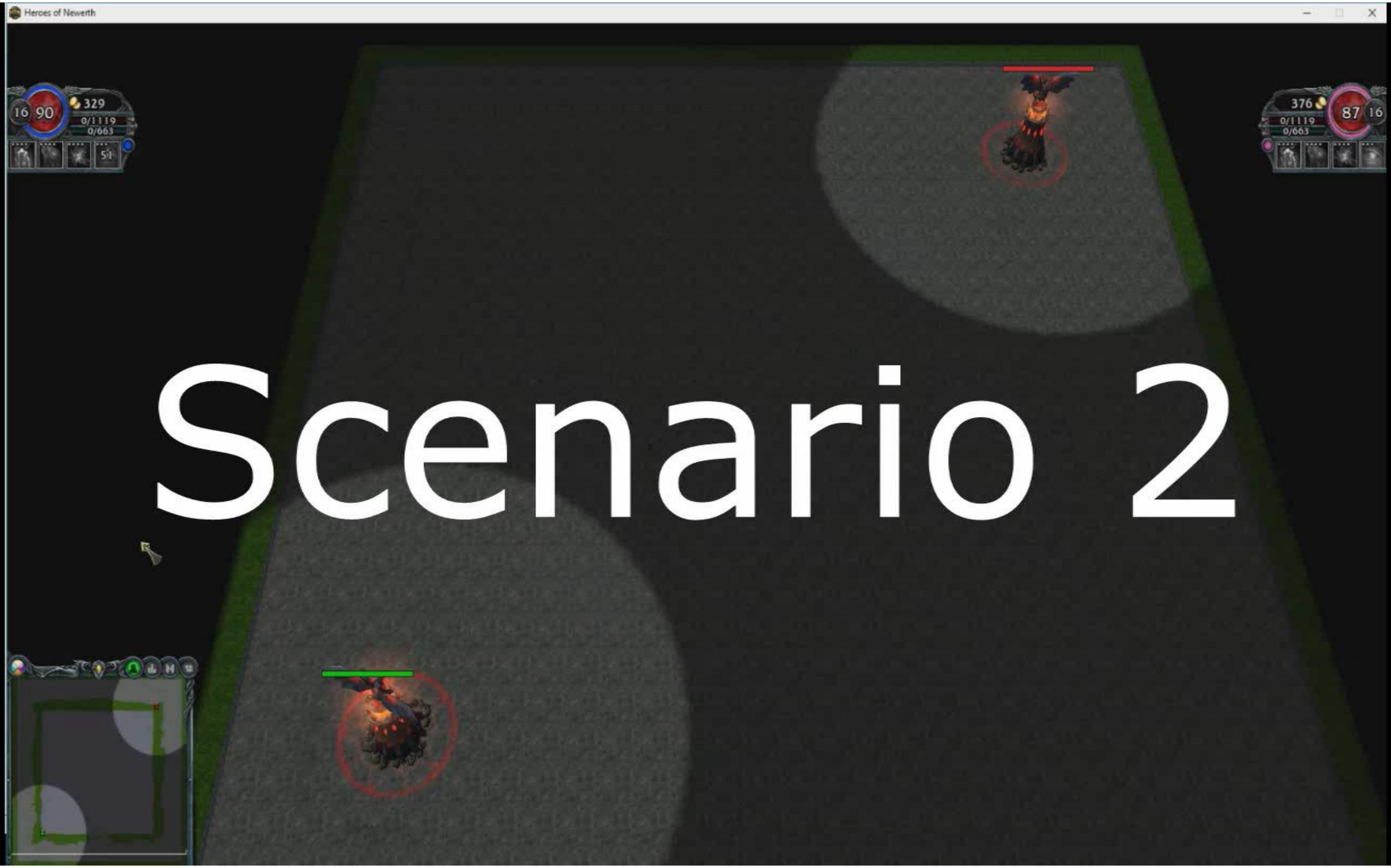


Weights obtained after number of training matches (self-play)



Averaging over win percent of 100 evenly sized groups of successive matches (self-play)


Heroes of Newerth
16
80
249
0/1119
0/663
741
80
16
0/1119
0/663
Scenario 1


Heroes of Newerth
Scenario 2
16 90 329
0/1119
0/663
376
87 16
0/1119
0/663

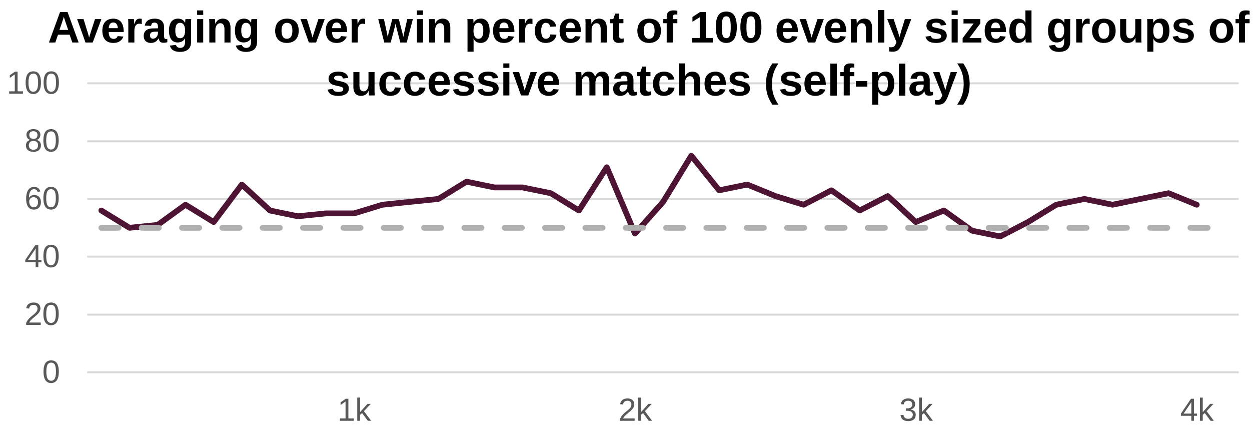
Scenario 3

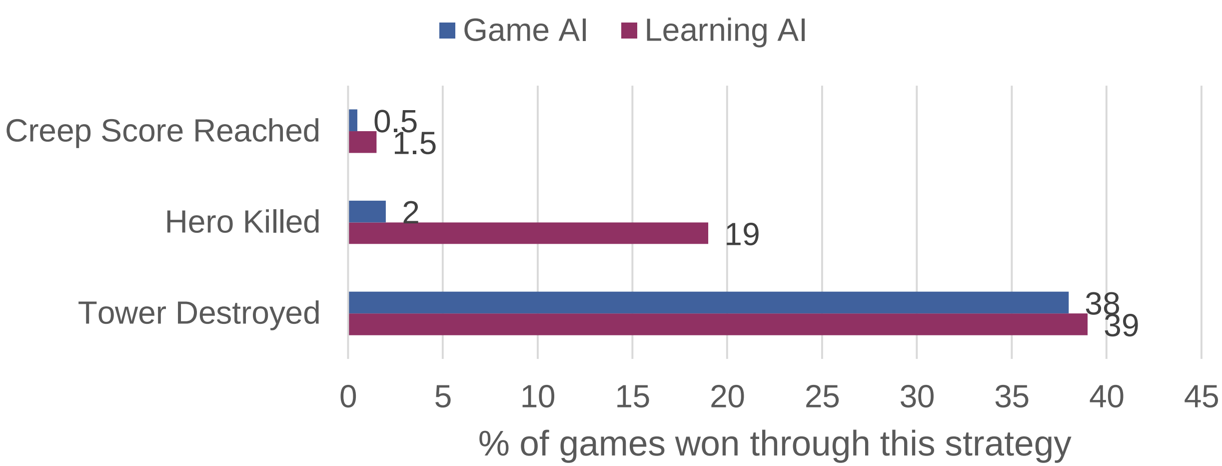
Learning Agent vs Game AI

1. Start from random weights; 10000 matches vs Game AI; $\alpha = 0.3, \lambda = 0.9$
2. 4000 evaluation games

Learning Agent vs Game AI

1. Start from random weights; 10000 matches vs Game AI; $\alpha = 0.3, \lambda = 0.9$
2. Further 4000 evaluation games






Heroes of Newerth
16
81
124
0/1119
0/663
929
84
16
0/1119
0/663

EXPERIMENTS AND RESULTS

OBSERVATIONS

- Game AI
  - Easily ”intimidated”
  - Not aware of own and opponents win conditions
  - Doesn’t understand game goals
  - Doesn’t alternate between behaviors well (e.g. farming and harrassing)
- Learning Agent
  - Understands the different game goals → different strategies!

OUTLINE

- ✓ Introduction
- ✓ Related Work
- ✓ Methods
- ✓ Application
- ✓ Experiments and Results
- Conclusion and Future Work

CONCLUSION

- Learning Agent for HoN
  - Reinforcement learning (TD(λ)) in combination with neural networks
  - Learning through self-play with *little* game knowledge
- Achievements
  - Diverse game-play strategies
    - Focus on harassing & killing the opponent
    - Focus on farming & pushing creep waves
  - Competitive at level of HoN Game AI

Simple Scenario	Extended Scenario

• Train vs Game AI
  • 85% win percent
    • $\alpha = 0.3, \lambda = 0.7$
	• Train by self-play
	  • 70% win percent
	    • $\alpha = 0.3, \lambda = 0.9$
	  • 90% win percent
	    • $\alpha = 0.3, \lambda = 0.1$
	• Train vs Game AI
	  • 65% win percent
	    • $\alpha = 0.3, \lambda = 0.9$

FUTURE WORK

- Testing
  - Further parameter variation
  - Optimal neural network features
  - Intermediate rewards
  - Playing against humans
  - Behaviors instead of actions
- Broader Implementation
  - Appliable against multiple opponent heroes
  - Even more exensive
- Different Concepts/Methods
  - Supervised Learning using human expert plays
  - From feed-forward to recurrent or convolutional multi-layer networks
- Other MOBA games

THANK YOU FOR YOUR ATTENTION