# Deep Reinforcement Learning for General Game Playing
# (Theory and Reinforcement)

Noah Arthurs (narthurs@stanford.edu) & Sawyer Birnbaum (sawyerb@stanford.edu)

**Abstract**— We created a machine learning algorithm that, given the rules of a two-player adversarial game, outputs a function Q such that Q(s, a) approximates the expected utility of action a in state s. Our Q-function is a neural network trained using reinforcement learning across data generated by self-play. We then create an agent for the game using the policy determined by the learned Q-function. Our algorithm is capable of creating very strong agents for small to medium sized games. It is scalable in that larger network sizes correspond to better play on a given game, and it easily generalizes across the games we have tested. We have made first steps toward determining the relationship between the complexity of a game and the complexity of the model required to play that game at a certain level. We have also found promising results using aggregate features to reduce input size and using weight sharing to jumpstart training.

## 1 Introduction

PAST reinforcement learning approaches to game playing have played individual adversarial games or general single-player games. We set out to combine these approaches in order to generalize across adversarial games. We take as input a vectorized representation of a game, and after training through self-play we output a neural network that approximates a Q-function. This function can then be used to create an agent for that game. We have made steps towards automating the process of selecting the structure of the neural network for a given game, which would be necessary for true General Game Playing[1].

## 2 Related Work

In the past there have been reinforcement learning algorithms applied to several adversarial games, notably TD-Gammon (Tesauro, 1992). There have also been successful attempts to apply reinforcement learning to general single player games, notably DeepMind's Atari project (2015). The success of the TD-Gammon algorithm depended on the stochastic nature of Backgammon. For this reason, we have modeled our algorithm after DeepMind's approach which is able to succeed in deterministic video game environments. The challenge then is adapting this approach to deterministic adversarial games.

## 3 Defining the Problem

### 3.1 Game Representation

Our game representation consists of a handful of functions that can be called during training to simulate games and store memories. For simulation, we have one function that given a state returns a list of legal moves, one function that given a state and a move returns the resulting state, and one function that given this resulting state checks to see if the game has ended. Each game also must provide a function that flips the perspective of the board between moves. This way our Q-function does not have to worry about determining which player it is playing as in each position. Each game also has a function that turns a state and an action into a single state-action vector. This is the format of the input to the Q-function and the format in which memories are stored.

As a specific example, our Tic-Tac-Toe state-action vector has 27 inputs. The first 9 represent the positions of the pieces of the player whose turn it is, the next nine represent the opponent's pieces, and the final 9 are a one-hot vector representing the action.

By default our state-action vector only contains indicator features, since this does not abstract away any information about the game.

[1] Our 221 project also involves General Game Playing. That project does not involve machine learning for finidng policies but instead is based on tree search algorithms. There is no code or data shared between the two projects, but we believe that combining these two approaches could result in a player stronger than either project created on its own.

### 3.2 Which Games?

We implemented games of small to medium size so that it would be easy to run many tests and verify good play. The games we have currently implemented are:

- *Tic-Tac-Toe*: The classic game of getting three in a row on a 3x3 grid. This is a small, simple game with very few states. We used it for initial testing in order to verify that our algorithm works.

- *Connect-4*: Drop pieces from the top in order to get four in a row in any direction. The rules are simple, but by varying the size of the board from 4x4 to 8x8 we could test a family of games with low to medium complexities.

- *Bidding Tic-Tac-Toe*: this is a variant of Tic-Tac-Toe in which each player has a certain amount of money that they can use to bid on the ability to place a piece on the board. We used this game to compare the use of indicator features to the use of aggregate features to represent amounts of money.

For convenience and ease of testing, we have only implemented two-player games, but we believe that our approach would generalize to three-plus-player and single-player games.

### 3.3 Measuring Effectiveness

We measure the effectiveness of an agent by taking its average score across 1000 games against an opponent, where a win is 1 point, a tie is 0.5 points, and a loss is 0 points. The initial test opponent was *Random*, an agent that makes random moves, but this resulted in our algorithm achieving very high win rates very quickly. As a result we switched to *Baseline*, an agent that looks one move ahead:

1. *Baseline* wins in one move if it can.
2. If *Baseline* cannot win immediately, it prunes moves that give its opponent the opportunity to win on the next turn.
3. *Baseline* then selects randomly from the remaining moves.

Intuitively *Baseline* represents a minimum human level of play, acting purely based on knowledge of the win conditions. Testing against *Baseline* has the following advantages:

- *Baseline* significantly outperforms Random.

- In order to beat Baseline, an agent needs to plan multiple moves in advance.

- During training it is easier to see incremental improvements of the agent in its score against *Baseline* than in its score against *Random* (5.1).

## 4 IMPLEMENTATION

### 4.1 The Pipeline

*Input*: Our input consists of a vectorized representation of a game (3.1) and a structure for the neural network that will be trained to approximate the Q-function (4.2).

*Initialization:* The weights of the network are initialized randomly using a truncated normal and the biases are initialized to zero. A bank of 10,000[2] memories (4.4) is generated through random play.

*Training:* Training alternates between two phases:

- First, the network plays 10 $\varepsilon$-greedy (4.4) games against itself, and the memories from these games overwrite the oldest memories in the memory bank.

- Second, we randomly sample 4 batches of 100 transitions from the memory bank, calculate targets (4.3), and backpropagate to update the weights in the network.

*Output:* After every 250 epochs we calculate the loss on the current memories and measure effectiveness by having the network play 1000 games against its testing opponent(s). If it is a new high score, the weights are saved. Training ends if no new high score is achieved for a stretch of 2500 epochs.

Note that, while the network is training against itself, we make the assumption that the best iteration of the network is the one that is most effective against *Baseline*. In the future we might measure effectiveness by having the network play against the current best set of weights, but we have not run into trouble from fitting to play against *Baseline*.

### 4.2 The Network

The network is a function from $\mathbb{R}^{SA}$ to $\mathbb{R}$ where SA is the dimension of the state-action vector. Currently we are using fully-connected layers with a ReLU activation function. Our final layer takes a sigmoid so that outputs will be between 0 and 1. In order to pass the network structure as input to the

[2] There are a handful of hyperparameters mentioned in this section that we did do detailed investigations of. All of these were set using trial and error during our early tests. Given more time, we would study these more rigorously and determine whether they should be set to the same values for every game or not.

pipeline, we specify how many hidden layers we want and how many units should be in each hidden layer. (When we refer to a 100x3 network, we mean one that has 3 layers of 100 hidden units each.) The network is implemented in TensorFlow and trained using TensorFlow's built in gradient descent optimizer with learning rate 0.01.

### 4.3 Bellman Loss

We store memories in the form $(s, a, r, s')$, representing a state, an action, an immediate reward, and a new state. Our approach is modeled after a variant of the Bellman Equation:

$$Q(s,a) = r + \gamma \max_{a'} Q(s', a')$$

which says that the quality of taking an action in a state is equal to the immediate reward plus the quality of taking the best action in the resulting state discounted by $\gamma$ (we used 0.9) If we use the right side of the equation as our target, we get the following loss function:

$$Loss = \left( Q(s,a) - \left( r + \gamma \max_{a'} Q(s', a') \right) \right)^2$$

Intuitively, minimizing this loss function creates continuity between consecutive outputs of the Q-function. At first Q acts randomly, but over time enforcing this continuity causes the rewards from the end states to trickle backwards.

### 4.4 Modifying DeepMind's Approach

We use two techniques that helped DeepMind generalize across tasks:

*Experience Replay*: Randomly sampling (s, a, r, s') transitions from a dynamic memory bank for back-propagation effectively decorrelates the data.

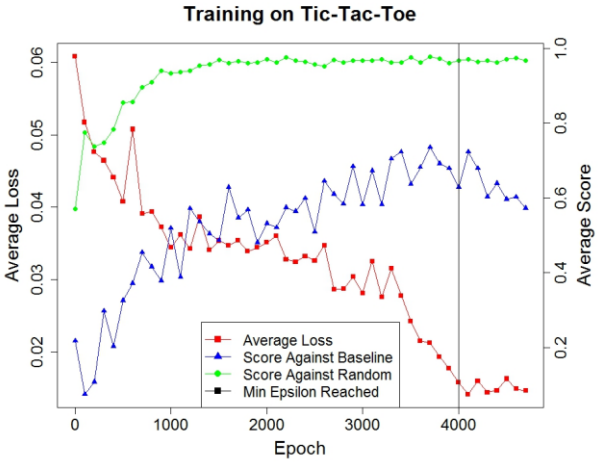
*ϵ-greediness*: In order to force exploration of new states, there is a probability of ϵ that any given move during training is made randomly. Like DeepMind, we slowly reduce the value of ϵ throughout training. Specifically, it starts at 1.0 and falls by 0.0002 every epoch until it reaches 0.2.

Unlike DeepMind, we are trying to learn a policy for an adversarial game rather than a single-player environment. We handle this by having s' represent the state reached after both the player and the opponent have moved. The opponent then ends up representing the environment. As ϵ decreases and our Q-function improves, this "environment" goes from being completely random to being a strong player. Because we are using self-play, this strong player is (secretly) our agent. Basically, we are learning a policy for a constantly evolving MDP which models how our agent plays at that point in training.

## 5 Results

### 5.1 Initial Testing

We started by training a network with one hidden layer of 100 units on Tic-Tac-Toe. Below we have graphed its effectiveness against *Baseline* and *Random*, and Bellman loss throughout training:




From these results we know:

- In general, lower loss correlates with better play. It is reasonable, however, that there are many times when it does not, because loss is calculated on the current memories, which are a constantly moving target.
- Testing against *Baseline* provides more information. *Random* peaks quickly and stays relatively constant, while *Baseline* is sensitive to small changes in loss throughout training.
- Beating *Baseline* means that our model can plan multiple moves in advance, which confirms that reward signals are being propagated backwards through the game.
- Both the win rate against *Random* and the win rate against *Baseline* indicate near optimal play, as *Random* will stumble into a tie a small percentage of the time and *Baseline* will often not give our agent the chance to set up a win.

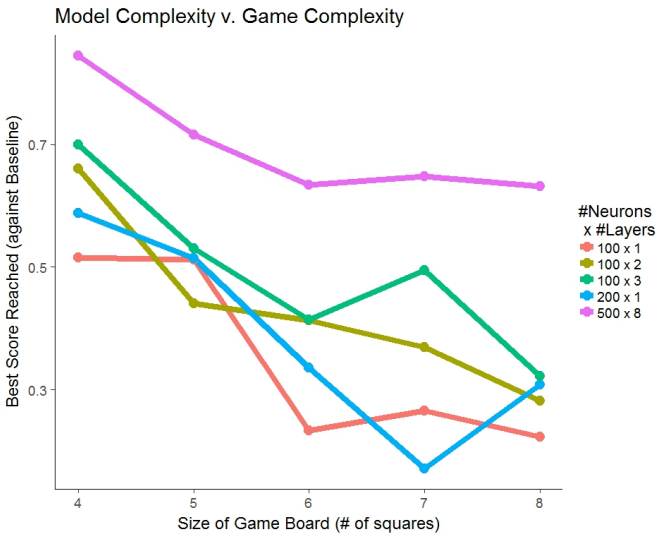
Basically these results indicate that our model is working in the way we want it to.

### 5.2 Investigating Model Complexity

Ultimately we would like to be able to set the dimensions of the neural network automatically, so we set out to explore the relationship between game complexity and model complexity.

We trained models with 1, 2, 3, and 8 hidden layers

on square Connect-4 grids ranging from 4x4 to 8x8. Below we have graphed the best score achieved by each model against game size:




Our results indicate:

- There is a roughly linear relationship between the performance of a given model and the size of the game. With data across a larger number of games, we might be able to learn this relationship.

- More complex models are almost uniformly more effective on a given game than less complex models.

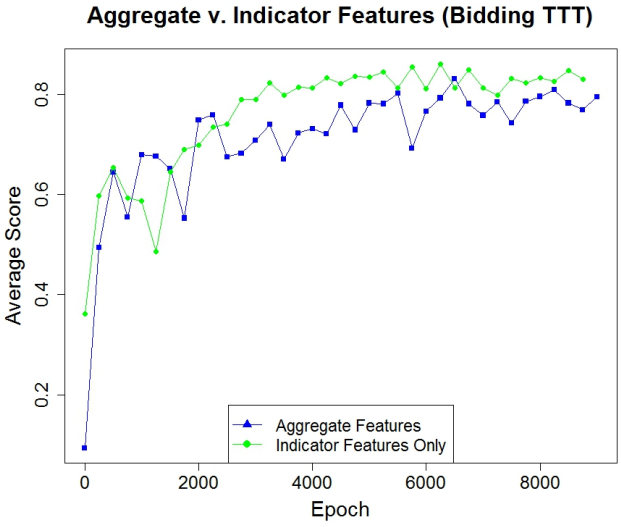
- Since two 100 unit layers generally outperformed one 200 unit layer, it seems that the depth of the network is more important than the breadth.

- Presumably, if we continue to ramp up the complexity, we could continue to drive up results on the larger boards.

## 5.3 Investigating Non-Indicator Features

We also wanted to explore whether using different types of features in the state-action vector had an effect on learning. For games like Tic-Tac-Toe and Connect-4, it is intuitive to use indicator features, but many games have a currency element which it is more natural to represent as a single number.

We tested Bidding Tic-Tac-Toe with two different feature extractions for the state-action vector. One version used indicator features to represent amounts of money for a total of 90 features. The other used a single aggregate feature to represent each amount of money for a total of 30 features. We trained our 500x8 network on both of these game representations:

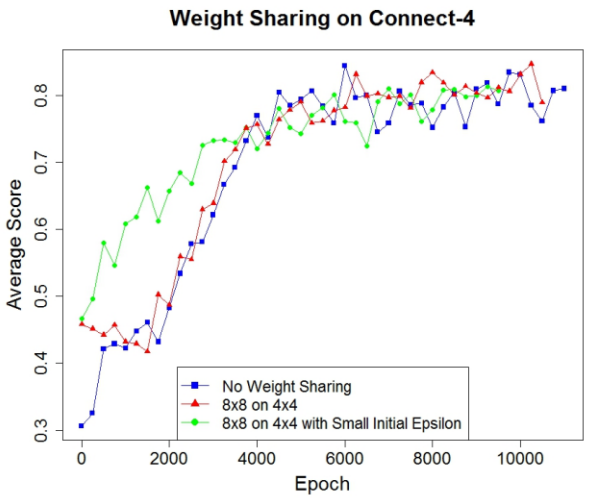



Despite having many fewer weights, the aggregate features performed almost as well as the indicator features, achieving a best score that was only 0.03 worse. This indicates that aggregate features can be an effective replacement for indicator features in at least some situations. It is possible, however, that if we were aggregating a larger number of indicator features, there would be a greater loss of performance.

## 5.4 Investigating Weight Sharing

Our final experiment involved weight sharing. We wanted to find out if the weights learned on one game could be transferred to another game. We have only taken one step in this direction, but if we could generalize weight sharing, then we could greatly speed up the training process.

We took our 500x8 network that had been trained on 8x8 Connect-4 and used the weights from the bottom left hand corner of the board to initialize the weights for 4x4 Connect-4. These transferred weights immediately produced a score of about 0.46, stunningly higher than the randomly initialized weights. However, after training began, the score got worse instead of better. This is because at the outset of training epsilon is 1.0, meaning that the weights were being trained on completely random memories, rather than memories generated by the network itself. When we changed the initial epsilon to be much lower, the results improved drastically:




Here we notice:

- The training that starts with the transferred weights and normal epsilon very quickly gets on the same track as the training from randomly initialized weights. The weight sharing is not being taken advantage of, since the memories are random.

- The training that starts with the transferred weights and lower epsilon shoots up very quickly but ultimately attains the same level of effectiveness as its peers.

These results indicate that although weight sharing does not create a higher effectiveness ceiling, it can be used to greatly speed up training.

## 6 Conclusion

We are very pleased with how well our model works. It seems that it scales well with the size of the neural network and that given a large enough memory size overfitting is impossible. In particular it is notable how effective it is to model the opponent as a constantly evolving MDP. This indicates that adversarial environments are not fundamentally different from any other Markov environment. The fact that the opponent is effectively modeled as a MDP means that our player is optimized to play against this particular estimated opponent, which is itself. This means no matter how well we can play any given game, the model will never be able to adapt to its opponents unless we allow it to generate memories against human opponents as well.

The tests we have run point us towards different ways to improve the model. We would like to gather more data on games and models of different complexities in order to more precisely learn the relationship between game complexity, model complexity, and performance. The fact that we were able to train well with non-indicator features means that it might be possible to train on more complex games where it is infeasible to represent states with only indicator features. In the future, we would also like to investigate whether weight sharing can transfer learning from a small game to a large game or between two games with a less direct relationship. For games with a lot of symmetry, it might be helpful to implement a convolutional neural network as well. We also would like to test games with different numbers of players, asymmetrical game elements etc. in order to find the capabilities and limitations of our system.

## Acknowledgment

We would like to thank Chris Piech for providing computational resources.

## References

[1] Tesauro, Gerald. “Temporal difference learning and TD-Gammon.” Communications of the ACM 38.3 (1995): 58-68.

[2] Mnih, Volodymyr, et al. “Playing atari with deep reinforcement learning.” arXiv preprint arXiv: 1312.5602 (2013).