RL for Large State Spaces: Value Function Approximation

Alan Fern

* Based in part on slides by Daniel Weld

Large State Spaces

- When a problem has a large state space we can not longer represent the V or Q functions as explicit tables
- Even if we had enough memory
  - Never enough training data!
  - Learning takes too long
- What to do??

Function Approximation

- Never enough training data!
  - Must generalize what is learned from one situation to other “similar” new situations
- Idea:
  - Instead of using large table to represent V or Q, use a parameterized function
    - The number of parameters should be small compared to number of states (generally exponentially fewer parameters)
  - Learn parameters from experience
  - When we update the parameters based on observations in one state, then our V or Q estimate will also change for other similar states
    - I.e. the parameterization facilitates generalization of experience

Linear Function Approximation

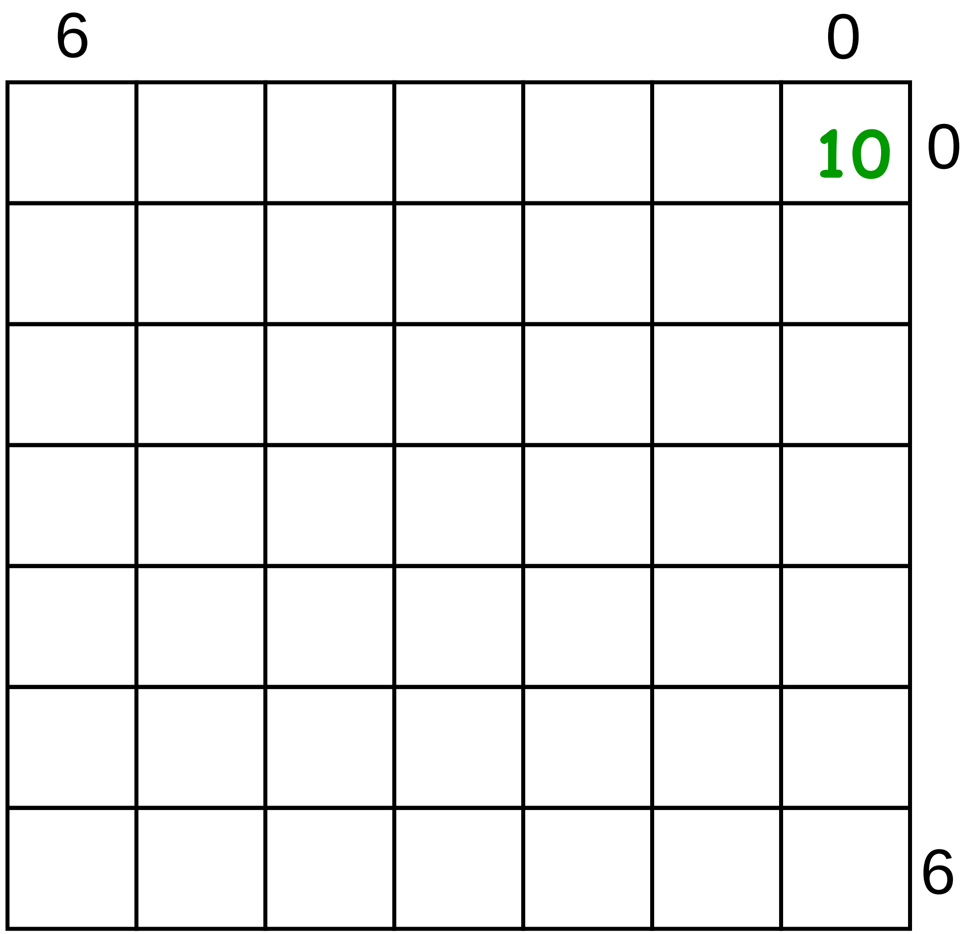
- Define a set of state features f1(s), …, fn(s)
  - The features are used as our representation of states
  - States with similar feature values will be considered to be similar
- A common approximation is to represent V*(s)* as a weighted sum of the features (i.e. a linear approximation)

$$\hat{V}_\theta(s) = \theta_0 + \theta_1 f_1(s) + \theta_2 f_2(s) + \ldots + \theta_n f_n(s)$$

- The approximation accuracy is fundamentally limited by the information provided by the features
- Can we always define features that allow for a perfect linear approximation?
  - Yes. Assign each state an indicator feature. (I.e. i'th feature is 1 iff i'th state is present and θ_i represents value of i'th state)
  - Of course this requires far to many features and gives no generalization.

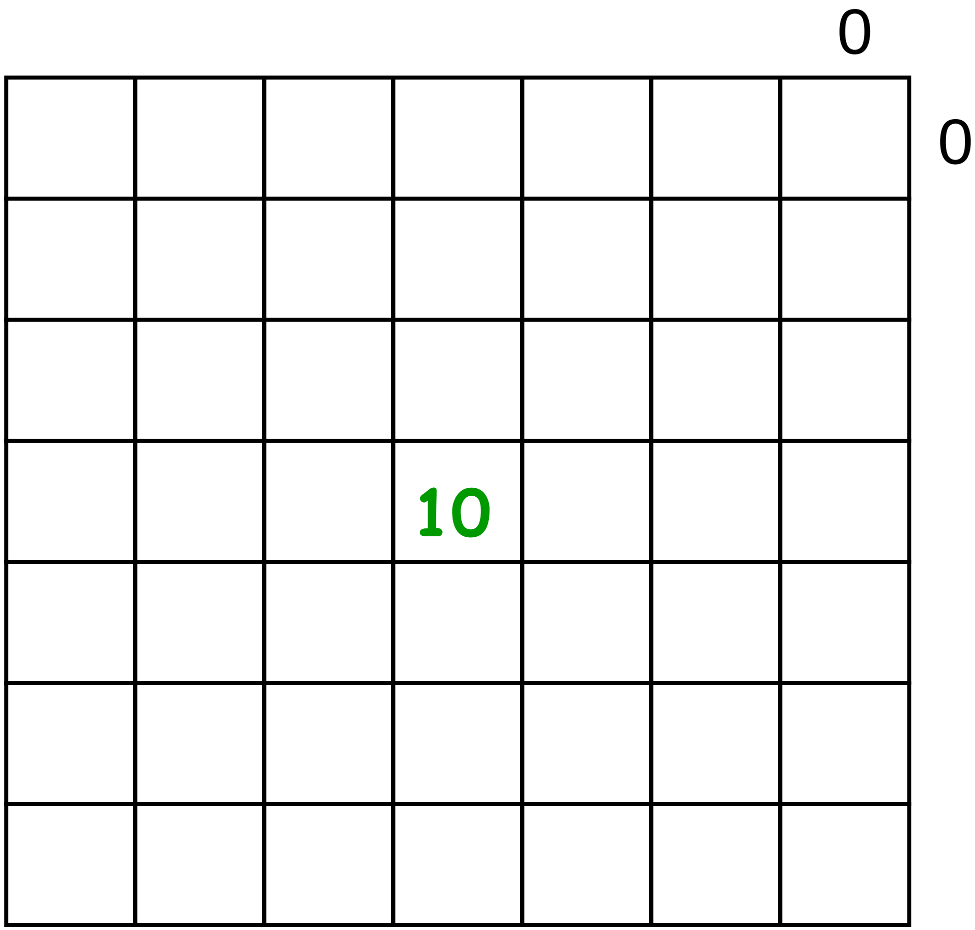
Example

- Grid with no obstacles, deterministic actions U/D/L/R, no discounting, -1 reward everywhere except +10 at goal
- Features for state s=(x,y): f1(s)=x, f2(s)=y (just 2 features)
- $V(s) = \theta_0 + \theta_1 x + \theta_2 y$
- Is there a good linear approximation?
  - Yes.
  - $\theta_0 = 10$, $\theta_1 = -1$, $\theta_2 = -1$
  - (note upper right is origin)
- $V(s) = 10 - x - y$ subtracts Manhattan dist. from goal reward



But What If We Change Reward …

- $V(s) = \theta_0 + \theta_1 x + \theta_2 y$
- Is there a good linear approximation?
  - No.

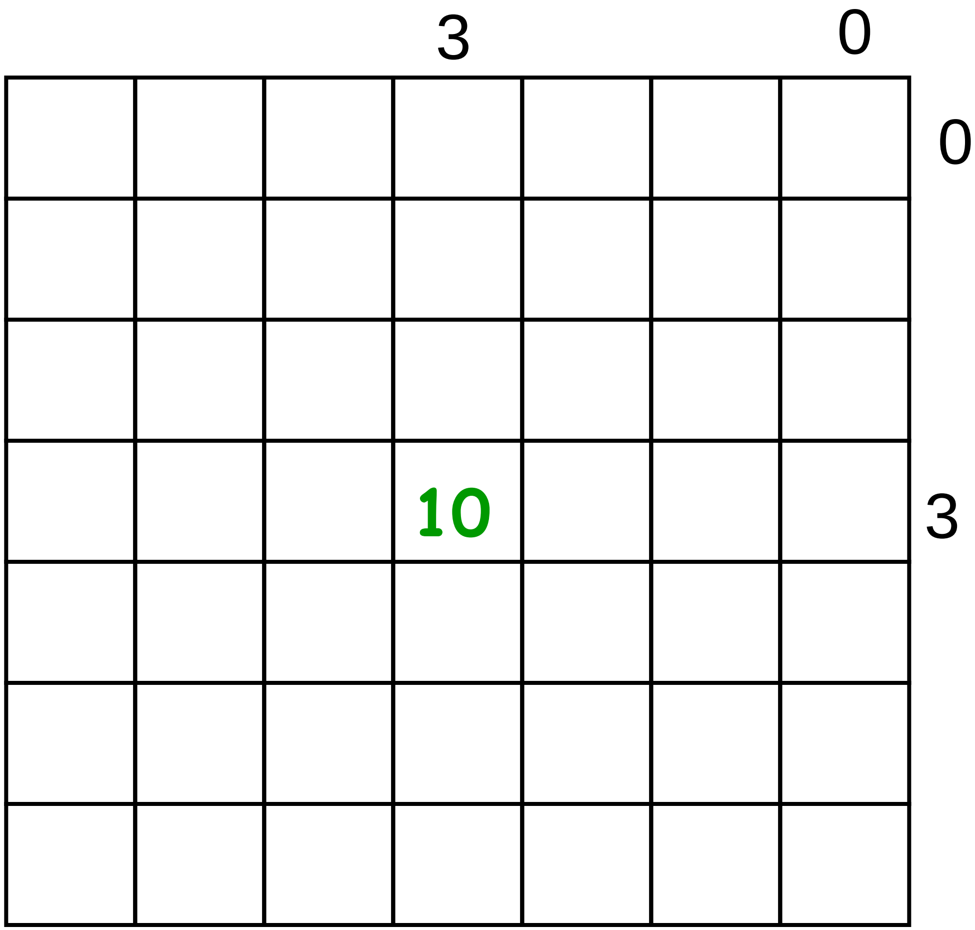


But What If…

- $V(s) = \theta_0 + \theta_1 x + \theta_2 y + \theta_3 z$

- Include new feature z
  - z= |3-x| + |3-y|
  - z is dist. to goal location

- Does this allow a good linear approx?
  - $\theta_0 = 10$, $\theta_1 = \theta_2 = 0$, $\theta_0 = -1$



Linear Function Approximation

- Define a set of features $f_1(s), \ldots, f_n(s)$
  - The features are used as our representation of states
  - States with similar feature values will be treated similarly
  - More complex functions require more complex features

$$\hat{V}_\theta(s) = \theta_0 + \theta_1 f_1(s) + \theta_2 f_2(s) + \ldots + \theta_n f_n(s)$$

- Our goal is to learn good parameter values (i.e. feature weights) that approximate the value function well
  - How can we do this?
  - Use TD-based RL and somehow update parameters based on each experience.

TD-based RL for Linear Approximators

1. Start with initial parameter values
2. Take action according to an explore/exploit policy (should converge to greedy policy, i.e. GLIE)
3. Update estimated model (if model is not available)
4. Perform TD update for each parameter

$$\theta_i \leftarrow ?$$

5. Goto 2

What is a "TD update" for a parameter?

Aside: Gradient Descent

- Given a function $f(\theta_1,\ldots,\theta_n)$ of n real values $\theta=(\theta_1,\ldots,\theta_n)$ suppose we want to minimize *f* with respect to θ
- A common approach to doing this is gradient descent
- The gradient of *f* at point θ, denoted by $\nabla_\theta f(\theta)$, is an n-dimensional vector that points in the direction where *f* increases most steeply at point θ
- Vector calculus tells us that $\nabla_\theta f(\theta)$ is just a vector of partial derivatives

$$\nabla_\theta f(\theta) = \left[\frac{\partial f(\theta)}{\partial \theta_1}, \ldots, \frac{\partial f(\theta)}{\partial \theta_n} \right]$$

where $$\frac{\partial f(\theta)}{\partial \theta_i} = \lim_{\varepsilon \to 0} \frac{f(\theta_1,\ldots\theta_{i-1},\theta_i+\varepsilon,\theta_{i+1},\ldots,\theta_n) - f(\theta)}{\varepsilon}$$

Can decrease f by moving in negative gradient direction

Aside: Gradient Descent for Squared Error

- Suppose that we have a sequence of states and target values for each state $\langle s_1, v(s_1)\rangle, \langle s_2, v(s_2)\rangle, \ldots$
  - E.g. produced by the TD-based RL loop
- Our goal is to minimize the sum of squared errors between our estimated function and each target value:

$$E_j = \frac{1}{2}\left(\hat{V}_\theta(s_j) - v(s_j)\right)^2$$

squared error of example j

our estimated value for j'th state

target value for j'th state

- After seeing j'th state the **gradient descent rule** tells us that we can decrease error by updating parameters by:

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial E_j}{\partial \theta_i}$$

learning rate

Aside: continued

$$\theta_i \leftarrow \theta_i - \alpha \frac{\partial E_j}{\partial \theta_i} = \theta_i - \alpha \underbrace{\left(\hat{V}_\theta(s_j) - v(s_j)\right)}_{\frac{\partial E_j}{\partial \hat{V}_\theta(s_j)}} \frac{\partial \hat{V}_\theta(s_j)}{\partial \theta_i}$$

$$E_j = \frac{1}{2}\left(\hat{V}_\theta(s_j) - v(s_j)\right)^2$$

depends on form of approximator

- For a linear approximation function:

$$\hat{V}_\theta(s) = \theta_1 + \theta_1 f_1(s) + \theta_2 f_2(s) + \ldots + \theta_n f_n(s)$$

$$\frac{\partial \hat{V}_\theta(s_j)}{\partial \theta_i} = f_i(s_j)$$

- Thus the update becomes: $\theta_i \leftarrow \theta_i + \alpha\left(v(s_j) - \hat{V}_\theta(s_j)\right) f_i(s_j)$
- For linear functions this update is guaranteed to converge to best approximation for suitable learning rate schedule

TD-based RL for Linear Approximators

1. Start with initial parameter values
2. Take action according to an explore/exploit policy (should converge to greedy policy, i.e. GLIE) Transition from s to s'
3. Update estimated model
4. Perform TD update for each parameter

$$\theta_i \leftarrow \theta_i + \alpha\left(v(s) - \hat{V}_\theta(s)\right)f_i(s)$$

5. Goto 2

What should we use for "target value" v(s)?

- Use the TD prediction based on the next state s'

$$v(s) = R(s) + \beta\hat{V}_\theta(s')$$

this is the same as previous TD method only with approximation

TD-based RL for Linear Approximators

1. Start with initial parameter values
2. Take action according to an explore/exploit policy (should converge to greedy policy, i.e. GLIE)
3. Update estimated model
4. Perform TD update for each parameter

$$\theta_i \leftarrow \theta_i + \alpha\left(R(s) + \beta\hat{V}_\theta(s') - \hat{V}_\theta(s)\right)f_i(s)$$

5. Goto 2

- Step 2 requires a model to select greedy action
- For some applications (e.g. Backgammon as we will see later) it is easy to get a model (but not easy to get a policy)
- For others it is difficult to get a good model

Q-function Approximation

- Define a set of features over state-action pairs: $f_1(s,a)$, …, $f_n(s,a)$
  - State-action pairs with similar feature values will be treated similarly
  - More complex functions require more complex features

$$\hat{Q}_\theta(s,a) = \theta_0 + \theta_1 f_1(s,a) + \theta_2 f_2(s,a) + \ldots + \theta_n f_n(s,a)$$

Features are a function of states and actions.

- Just as for TD, we can generalize Q-learning to update the parameters of the Q-function approximation

Q-learning with Linear Approximators

1. Start with initial parameter values
2. Take action a according to an explore/exploit policy (should converge to greedy policy, i.e. GLIE) transitioning from s to s'
3. Perform TD update for each parameter

$$\theta_i \leftarrow \theta_i + \alpha\Big(\underbrace{R(s) + \beta \max_{a'} \hat{Q}_\theta(s',a')}_{\text{estimate of Q(s,a) based on observed transition}} - \hat{Q}_\theta(s,a)\Big) f_i(s,a)$$

4. Goto 2

- TD converges close to minimum error solution
- Q-learning can diverge. Converges under some conditions.

Example: Tactical Battles in Wargus

- Wargus is real-time strategy (RTS) game
  - Tactical battles are a key aspect of the game

5 vs. 5

10 vs. 10

- **RL Task**: learn a policy to control *n* friendly agents in a battle against *m* enemy agents
  - Policy should be applicable to tasks with different sets and numbers of agents

Example: Tactical Battles in Wargus

- **States**: contain information about the locations, health, and current activity of all friendly and enemy agents
- **Actions**: Attack(F,E)
  - causes friendly agent F to attack enemy E
- **Policy**: represented via Q-function Q(s,Attack(F,E))
  - Each decision cycle loop through each friendly agent F and select enemy E to attack that maximizes Q(s,Attack(F,E))
- Q(s,Attack(F,E)) generalizes over any friendly and enemy agents F and E
  - We used a linear function approximator with Q-learning

Example: Tactical Battles in Wargus

$$\hat{Q}_\theta(s,a) = \theta_1 + \theta_1 f_1(s,a) + \theta_2 f_2(s,a) + \ldots + \theta_n f_n(s,a)$$

- Engineered a set of relational features
  $\{f_1(s,Attack(F,E)), \ldots., f_n(s,Attack(F,E))\}$

- **Example Features**:
  - # of other friendly agents that are currently attacking E
  - Health of friendly agent F
  - Health of enemy agent E
  - Difference in health values
  - Walking distance between F and E
  - Is E the enemy agent that F is currently attacking?
  - Is F the closest friendly agent to E?
  - Is E the closest enemy agent to E?
  - …

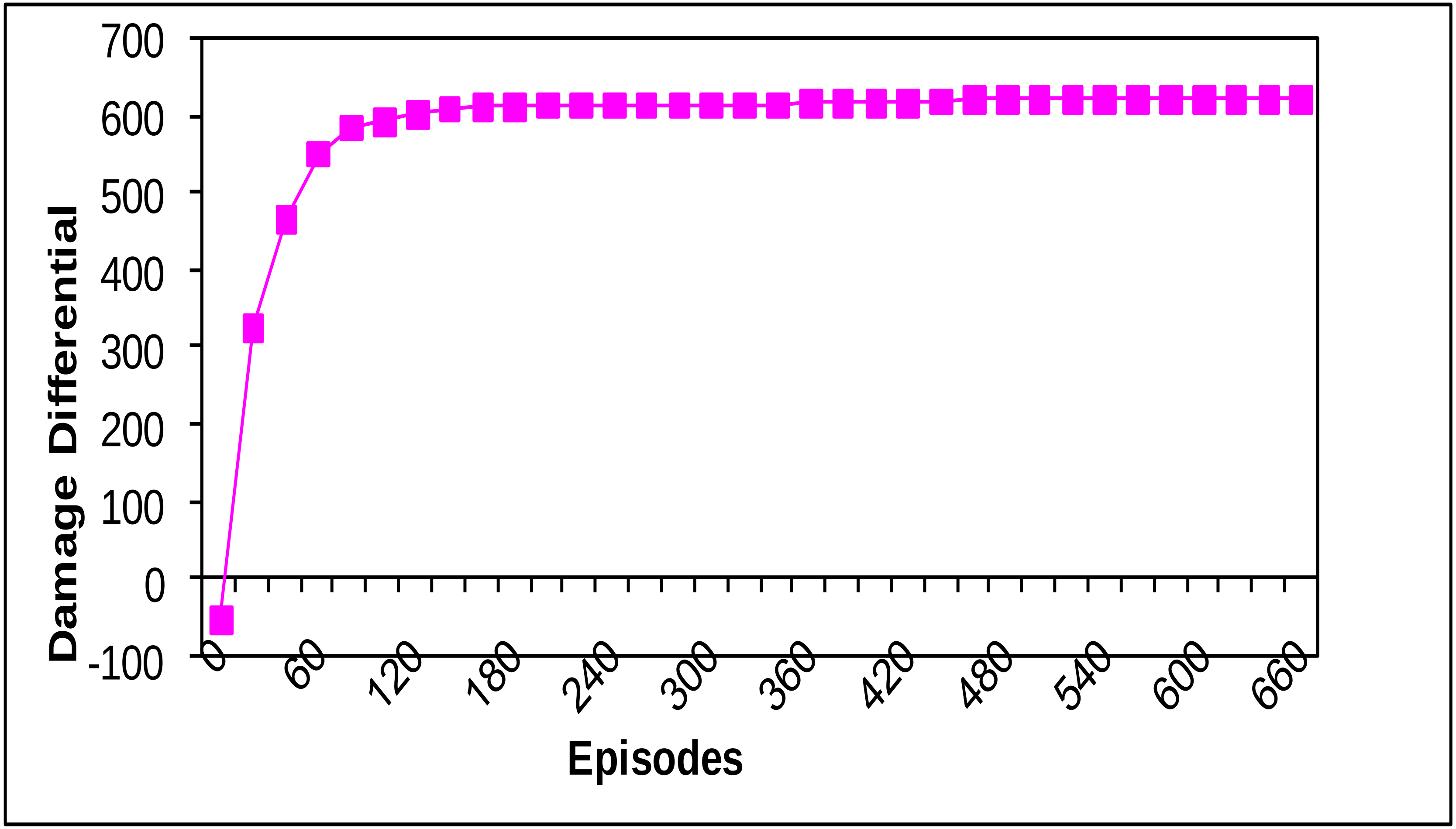
- Features are well defined for any number of agents

Example: Tactical Battles in Wargus

Initial random policy

Example: Tactical Battles in Wargus

- Linear Q-learning in 5 vs. 5 battle



Example: Tactical Battles in Wargus

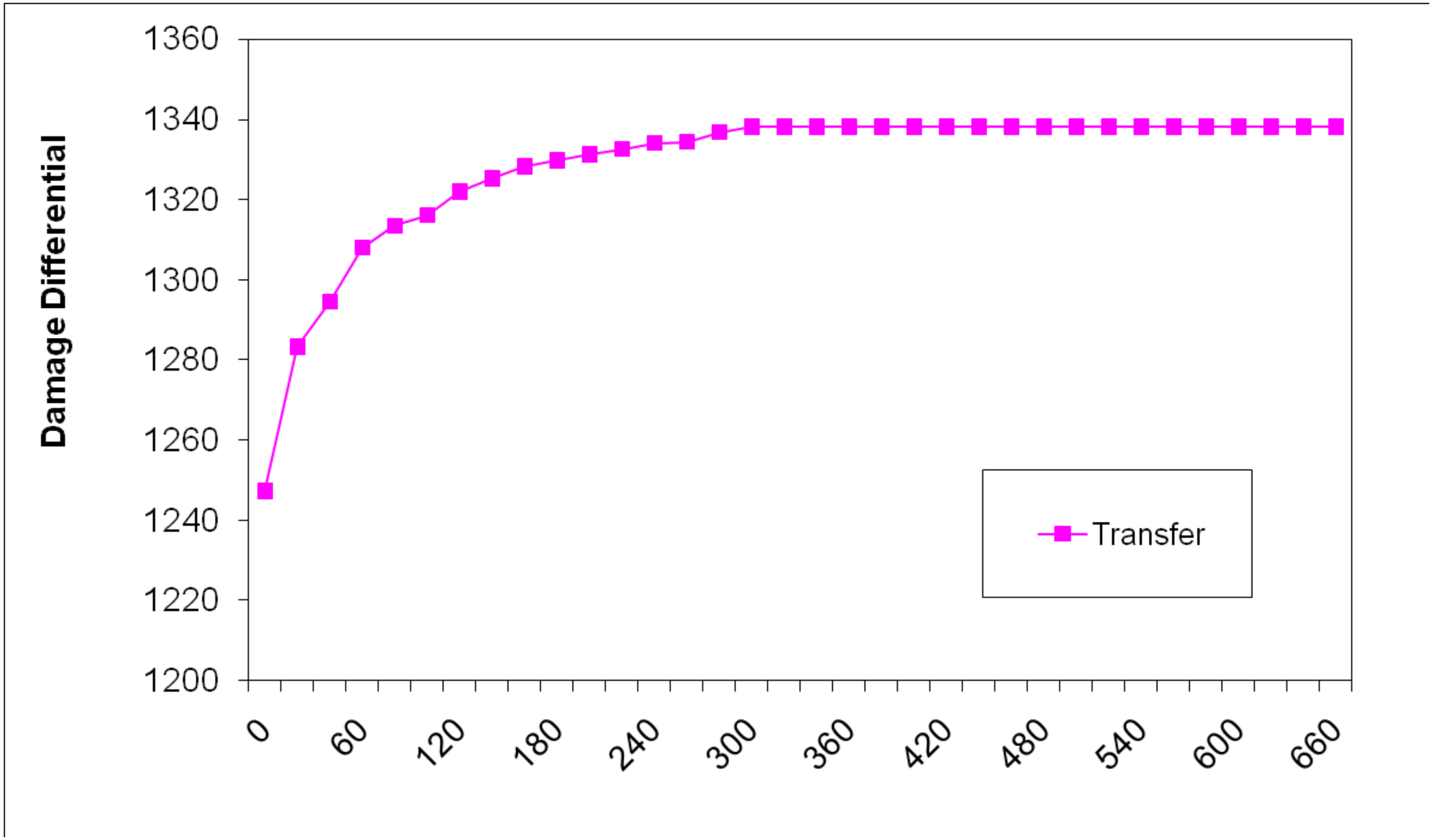
Learned Policy after 120 battles

Example: Tactical Battles in Wargus

10 vs. 10 using policy learned on 5 vs. 5

Example: Tactical Battles in Wargus

- Initialize Q-function for 10 vs. 10 to one learned for 5 vs. 5
  - Initial performance is very good which demonstrates generalization from 5 vs. 5 to 10 vs. 10



Q-learning w/ Non-linear Approximators

$\hat{Q}_\theta(s,a)$ is sometimes represented by a non-linear approximator such as a neural network

1. Start with initial parameter values
2. Take action according to an explore/exploit policy (should converge to greedy policy, i.e. GLIE)
3. Perform TD update for each parameter

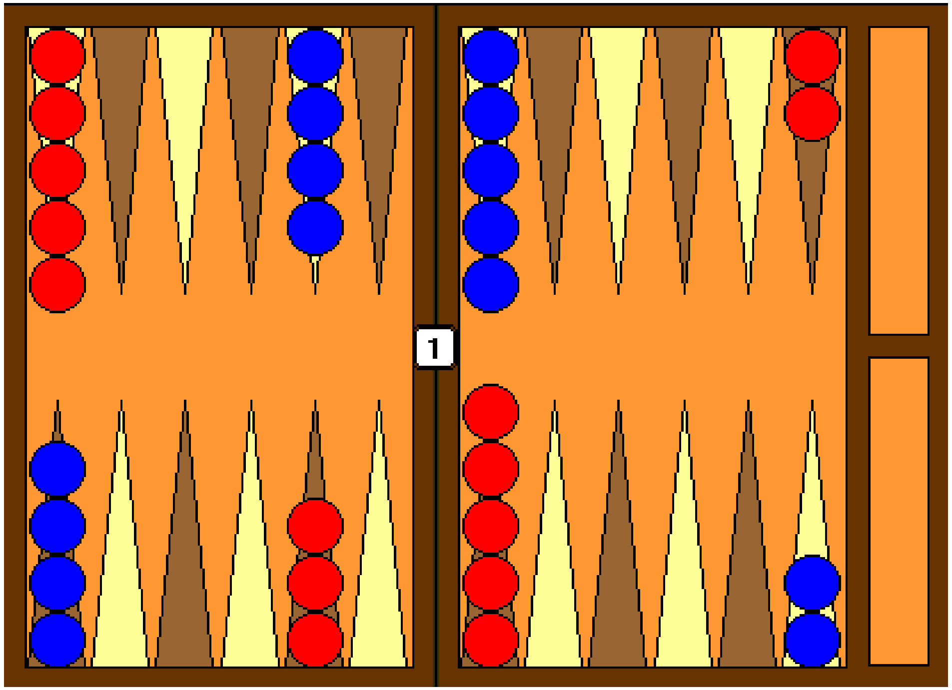
$$\theta_i \leftarrow \theta_i + \alpha\left(R(s) + \beta \max_{a'} \hat{Q}_\theta(s',a') - \hat{Q}_\theta(s,a)\right)\frac{\partial \hat{Q}_\theta(s,a)}{\partial \theta_i}$$

4. Goto 2

calculate closed-form

- Typically the space has many local minima and we no longer guarantee convergence
- Often works well in practice

~Worlds Best Backgammon Player



- Neural network with 80 hidden units
- Used TD-updates for 300,000 games against self
- Is one of the top (2 or 3) players in the world!