COMS W4701: Artificial Intelligence

Lecture 13: RL Generalization and Applications

Tony Dear, Ph.D.
Department of Computer Science
School of Engineering and Applied Sciences

Today

- Model-based reinforcement learning
- Function approximation
- Stochastic gradient descent
- Reinforcement learning applications

Models and Planning

- All search and decision problems that we discussed prior to RL assumed knowledge of the environment *model*
- With knowledge of a complete model, an agent can *plan offline*

- If the model is stochastic, an agent can also *simulate experiences*
- These experiences can then be used to estimate values and policies

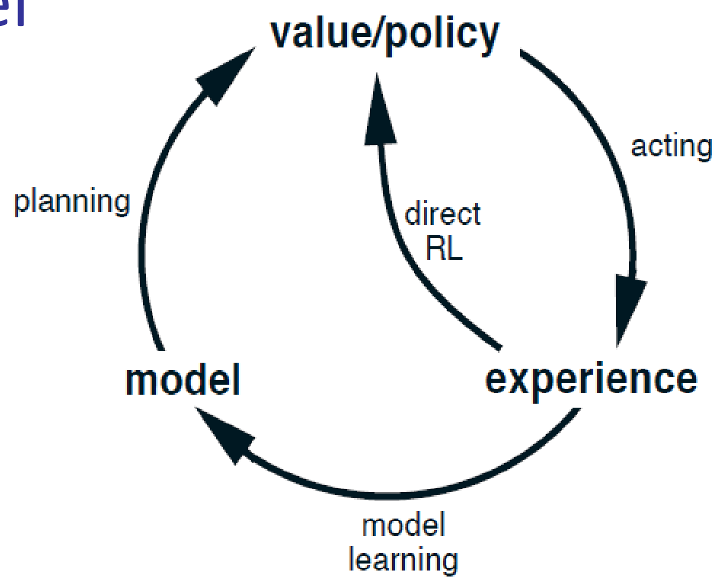
- Learning methods use "real" experiences in an online setting, but they should also work with *simulated* experience as well!

Planning, Acting, and Learning

- Suppose our agent is able to plan given some initial environment model
- The agent acts according to its plan (or policy) and makes observations

- These can then be used to improve the existing value function and policy
- These can also be used to improve the agent’s model

- *Model learning* can then indirectly also change the agent’s value function and policy
- May be better in cases of limited experiences

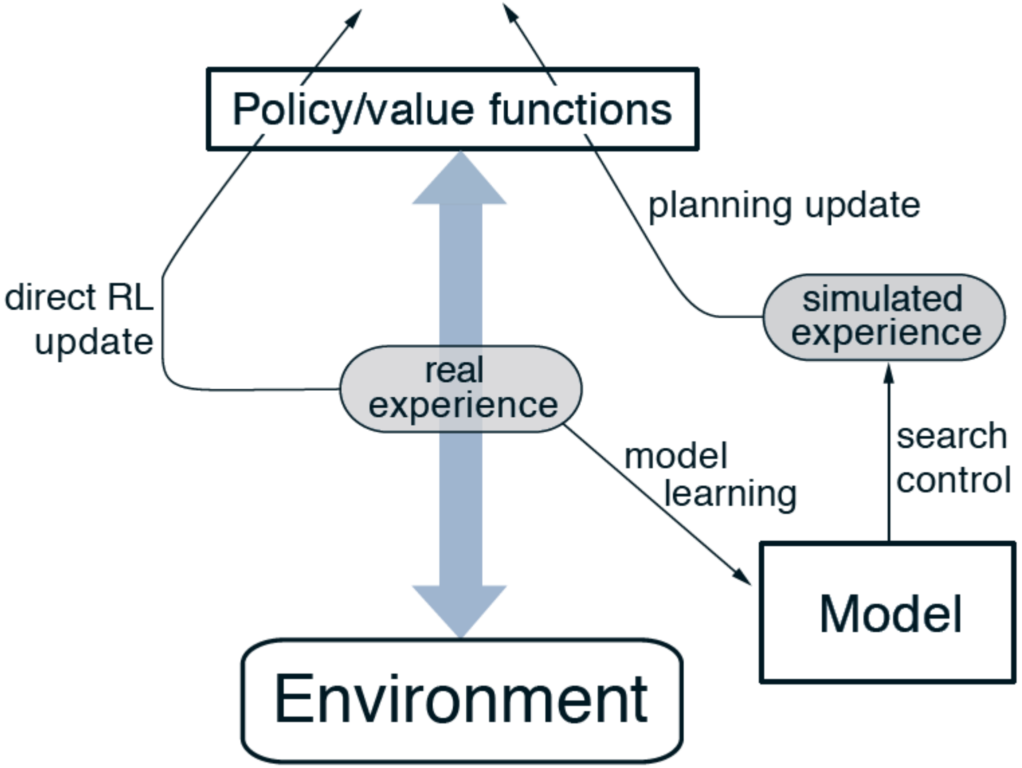


Dyna-Q

- Dyna-Q integrates both learning and planning in the same algorithm
- Steps (a-d) of each iteration are just regular RL (e.g., Q-learning)

Tabular Dyna-Q

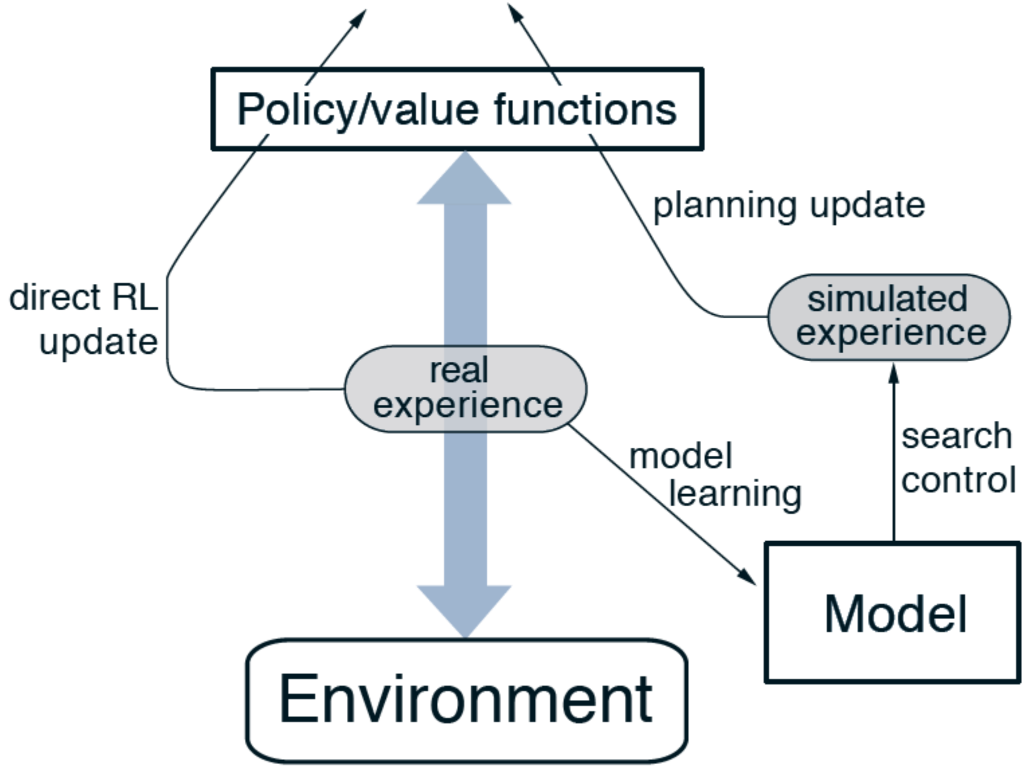
Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Loop forever:
  (a) $S \leftarrow$ current (nonterminal) state
  (b) $A \leftarrow \varepsilon\text{-greedy}(S, Q)$
  (c) Take action A; observe resultant reward, R, and state, S'
  (d) $Q(S,A) \leftarrow Q(S,A) + \alpha\big[R + \gamma \max_a Q(S',a) - Q(S,A)\big]$



Dyna-Q

- After updating Q-value, we can perform *model learning* by updating transition probabilities and rewards
- E.g., we can compute and update average rewards, frequencies of all possible state-action-state transitions

(e) $Model(S, A) \leftarrow R, S'$
(f) Loop repeat n times:
    $S \leftarrow$ random previously observed state
    $A \leftarrow$ random action previously taken in S
    $R, S' \leftarrow Model(S, A)$
    $Q(S, A) \leftarrow Q(S, A) + \alpha\left[R + \gamma \max_a Q(S', a) - Q(S, A)\right]$

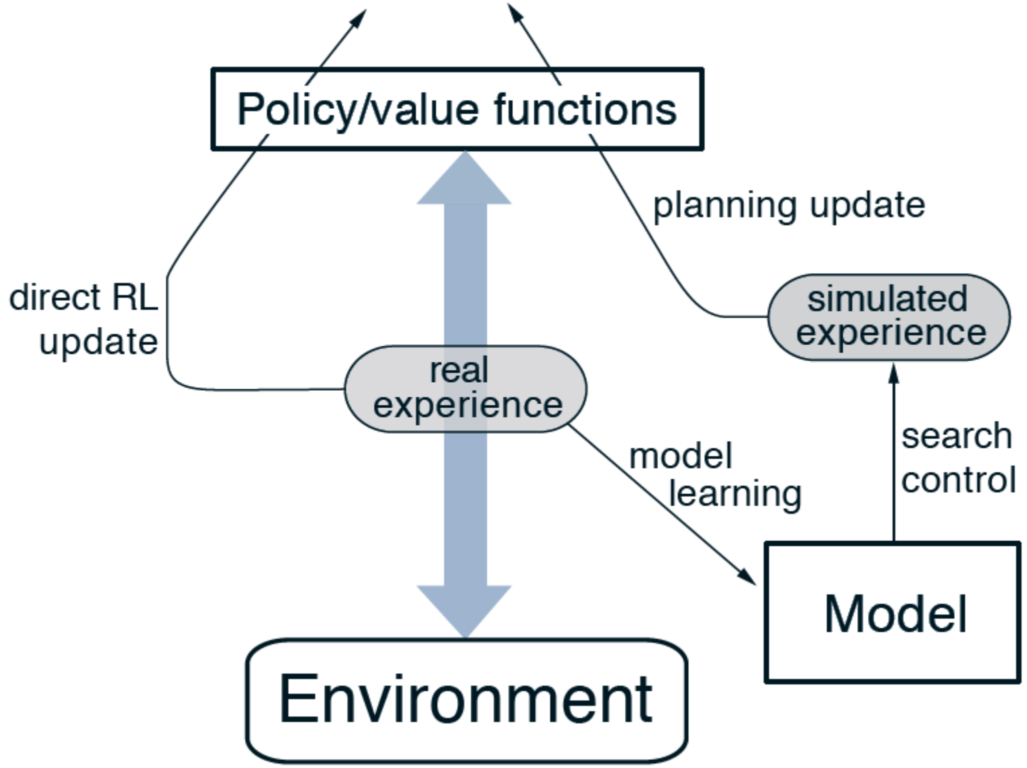


Dyna-Q

- Finally, we can *continue learning Q-values* by using the current model to *simulate* rewards and transitions (planning step)

- Query the model for past experiences
- Subsequent Q-value updates can then lead to updates in the policy!

(e) $Model(S, A) \leftarrow R, S'$
(f) Loop repeat n times:
  $S \leftarrow$ random previously observed state
  $A \leftarrow$ random action previously taken in S
  $R, S' \leftarrow Model(S, A)$
  $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$

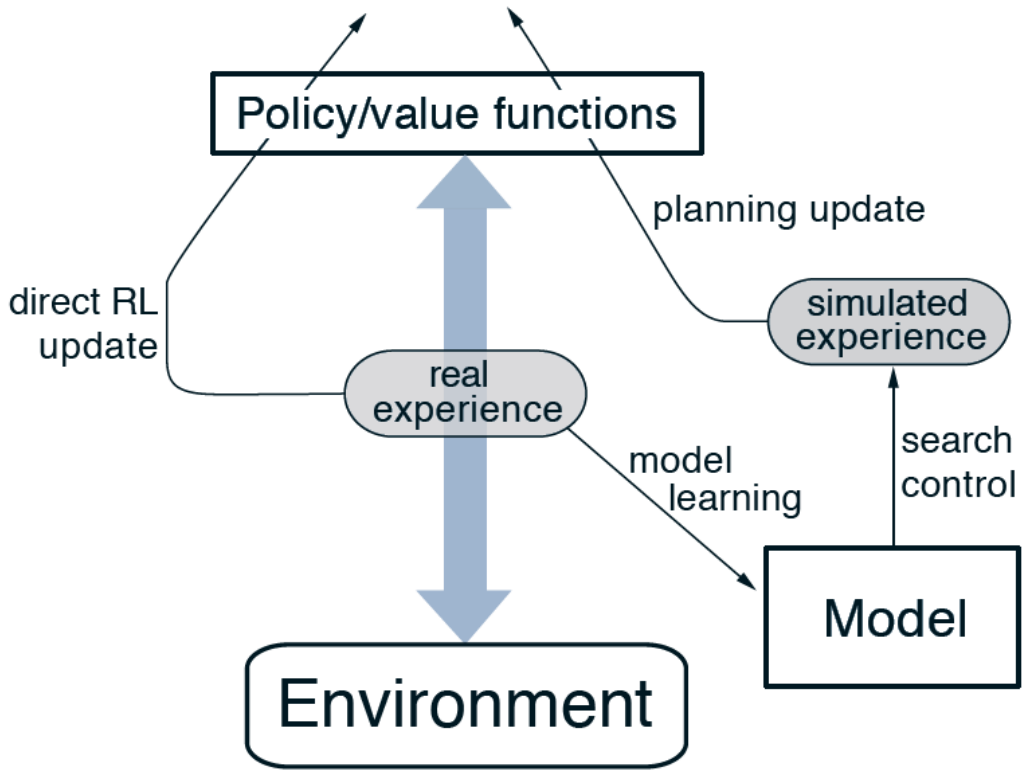


Dyna-Q

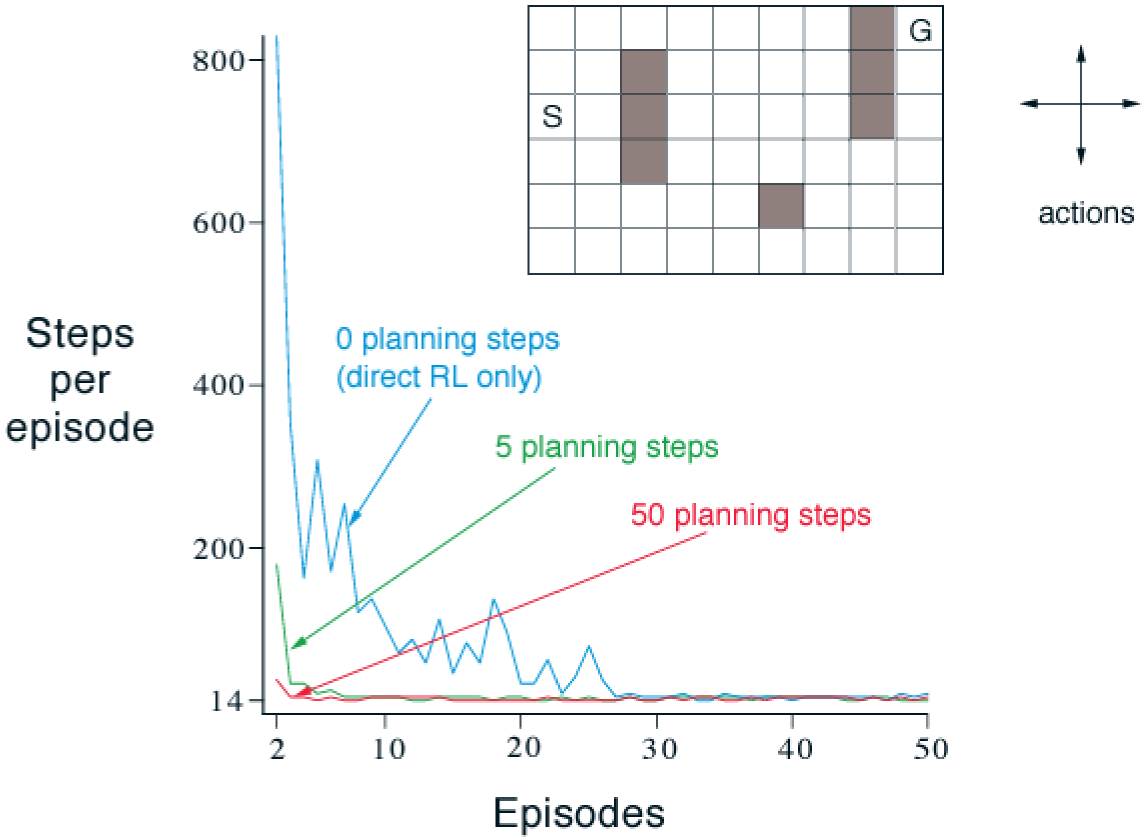
- Dyna-Q integrates both learning and planning in the same algorithm
- In practice, we can interleave both stages in an arbitrary way

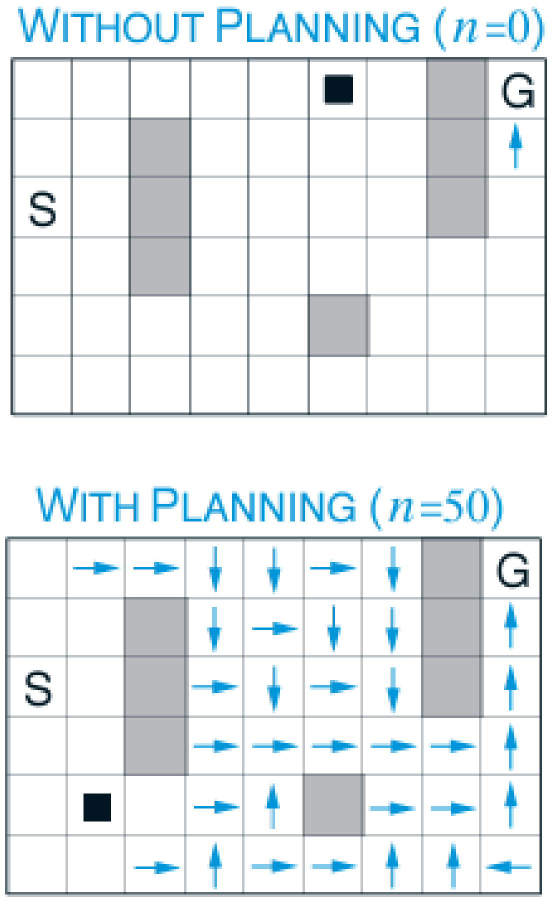
Tabular Dyna-Q

Initialize $Q(s,a)$ and $Model(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}(s)$
Loop forever:
  (a) $S \leftarrow$ current (nonterminal) state
  (b) $A \leftarrow \varepsilon$-greedy(S,Q)
  (c) Take action A; observe resultant reward, R, and state, S'
  (d) $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$
  (e) $Model(S,A) \leftarrow R, S'$
  (f) Loop repeat n times:
    $S \leftarrow$ random previously observed state
    $A \leftarrow$ random action previously taken in S
    $R, S' \leftarrow Model(S,A)$
    $Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',a) - Q(S,A)\right]$



Maze Example





Function Approximation

- In RL, we are trying to learn *functions* over states or state-actions: V, Q, π
- So far these have specified values for *every* individual state!
- Not possible in problems with too many states or continuous states

- We need to do *function approximation*—instead of learning $V^{\pi}(s)$, we learn a (smaller) set of weights $\mathbf{w}$ describing a function $\hat{V}(s, \mathbf{w})$

- Like the evaluation functions used in game trees, $\hat{V}$ may be a linear feature combination, neural network, decision tree, etc.

Objective Functions

- Suppose we've chosen a function parameterization for $\hat{V}$ with weights $\mathbf{w}$
- How do we *update* $\mathbf{w}$ given a sample transition sequence?

- Specify an underlying *objective*, e.g., minimizing *squared error* in $\hat{V}$:

$$L(\mathbf{w}) = \frac{1}{2}\left(V^{\pi}(s) - \hat{V}(s, \mathbf{w})\right)^2$$

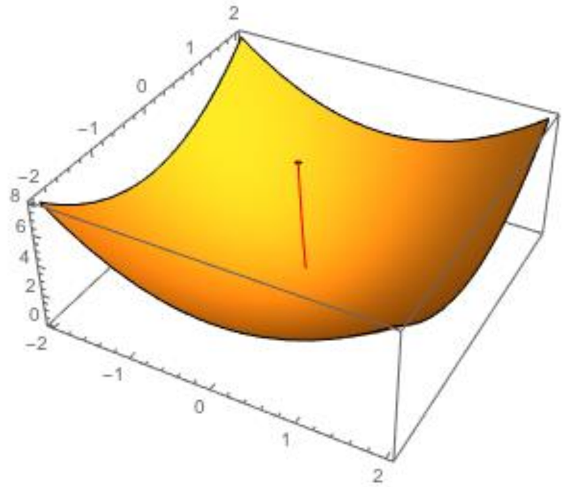
- Ideally we can find a global optimum $\mathbf{w}^*$ that minimizes L, but many problems will have many local optima as well

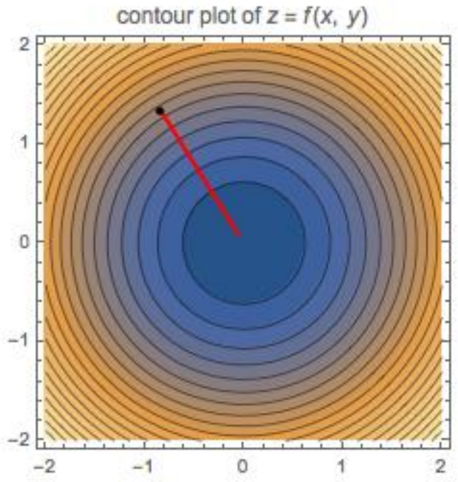
Gradient Descent

- Suppose that L is a differentiable function of the weight vector $\mathbf{w}$
- The **gradient** of a multivariate function $L(w_1, \ldots, w_n)$ is a *vector* of partial derivatives wrt all w_i

$$\frac{\partial L}{\partial \mathbf{w}} = \left[\frac{\partial L}{\partial w_1} \quad \cdots \quad \frac{\partial L}{\partial w_n}\right]$$

- Indicates magnitude and *direction* of largest change in L
- The value of L *decreases* fastest along direction of $-\frac{\partial L}{\partial \mathbf{w}}$
- **Gradient descent:** Initialize the configuration $\mathbf{w}$, and repeatedly update it as $\mathbf{w} \leftarrow \mathbf{w} - \alpha \frac{\partial L}{\partial \mathbf{w}}$ until convergence





Updating the Weights

- The gradient descent update for the mean squared error of $\hat{V}$ is thus

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \frac{\partial}{\partial \mathbf{w}_t} \frac{1}{2} \left(V^\pi(s_t) - \hat{V}(s_t, \mathbf{w}_t) \right)^2$$
$$= \mathbf{w}_t + \alpha \left(V^\pi(s_t) - \hat{V}(s_t, \mathbf{w}_t) \right) \frac{\partial \hat{V}(s_t, \mathbf{w}_t)}{\partial \mathbf{w}_t}$$

- Ex: Linear combination of features

$$\hat{V}(s, \mathbf{w}) = w_1 x_1(s) + w_2 x_2(s) + \cdots = \sum_i w_i x_i(s) = \mathbf{w}^\top \mathbf{x}(s)$$

- Gradient is simply $\dfrac{\partial \hat{V}(s_t, \mathbf{w}_t)}{\partial \mathbf{w}_t} = (x_1, x_2, \ldots, x_n) = \mathbf{x}(s)$

Stochastic Gradient Descent

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha\left(V^\pi(s_t) - \hat{V}(s_t, \mathbf{w}_t)\right)\frac{\partial \hat{V}(s_t, \mathbf{w}_t)}{\partial \mathbf{w}_t}$$

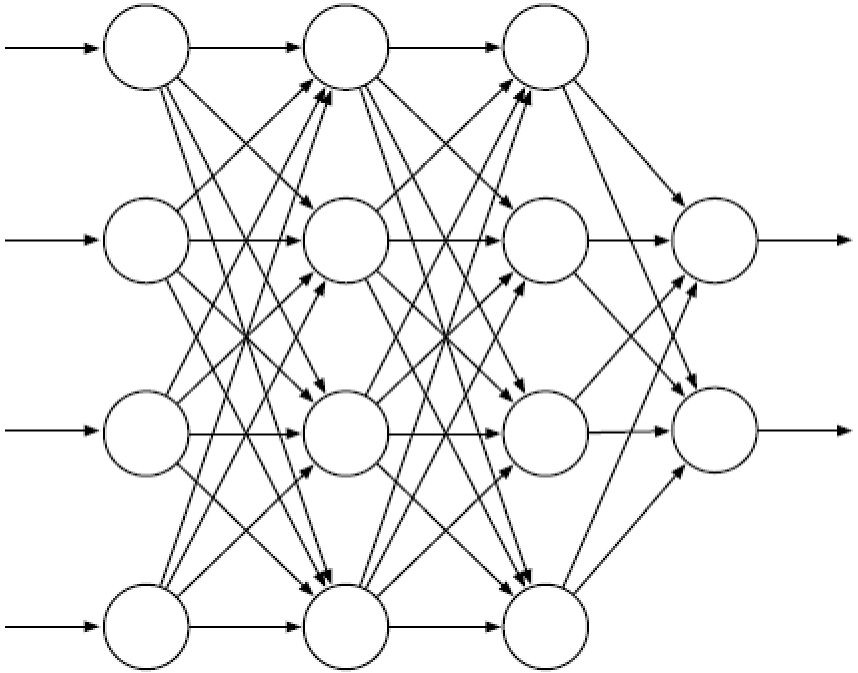
- Last issue: We don’t know the “true” or *target* value $V^\pi(s_t)$
- **Stochastic gradient descent**: *Approximate* the gradient using samples, just like in classical RL

- Gradient MC prediction: $\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha\left(G_t - \hat{V}(s_t, \mathbf{w}_t)\right)\frac{\partial \hat{V}(s_t, \mathbf{w}_t)}{\partial \mathbf{w}_t}$
- Semi-gradient TD(0): $\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha\left(r + \gamma\hat{V}(s_{t+1}, \mathbf{w}_t) - \hat{V}(s_t, \mathbf{w}_t)\right)\frac{\partial \hat{V}(s_t, \mathbf{w}_t)}{\partial \mathbf{w}_t}$

Neural Networks

- **Neural networks** are function approximators composed of *layers* of *neurons*
- Each neuron applies a nonlinear *activation function* to a weighted sum of inputs

- Hidden (internal) layers are “feature” transformations of raw inputs
- More neurons yield greater representative power, but at the expense of training efficiency

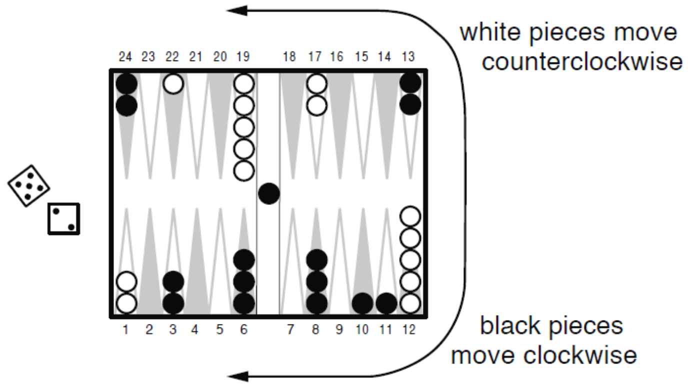


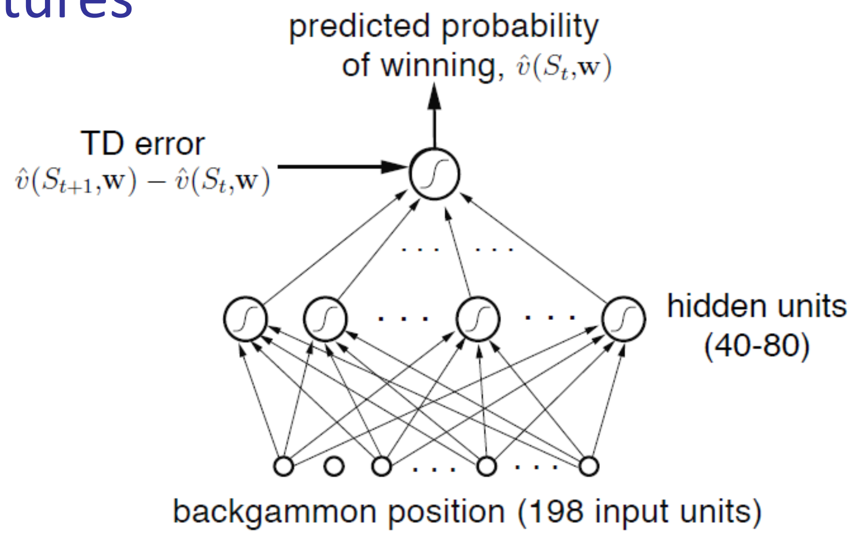
- Goal: Learn the weights of the neuron connections
- Most methods like *backpropagation* run some kind of stochastic gradient descent

TD-Gammon

- Backgammon: High branching factor (~400), highly stochastic game
- Can approach using depth-limited tree search with evaluation function

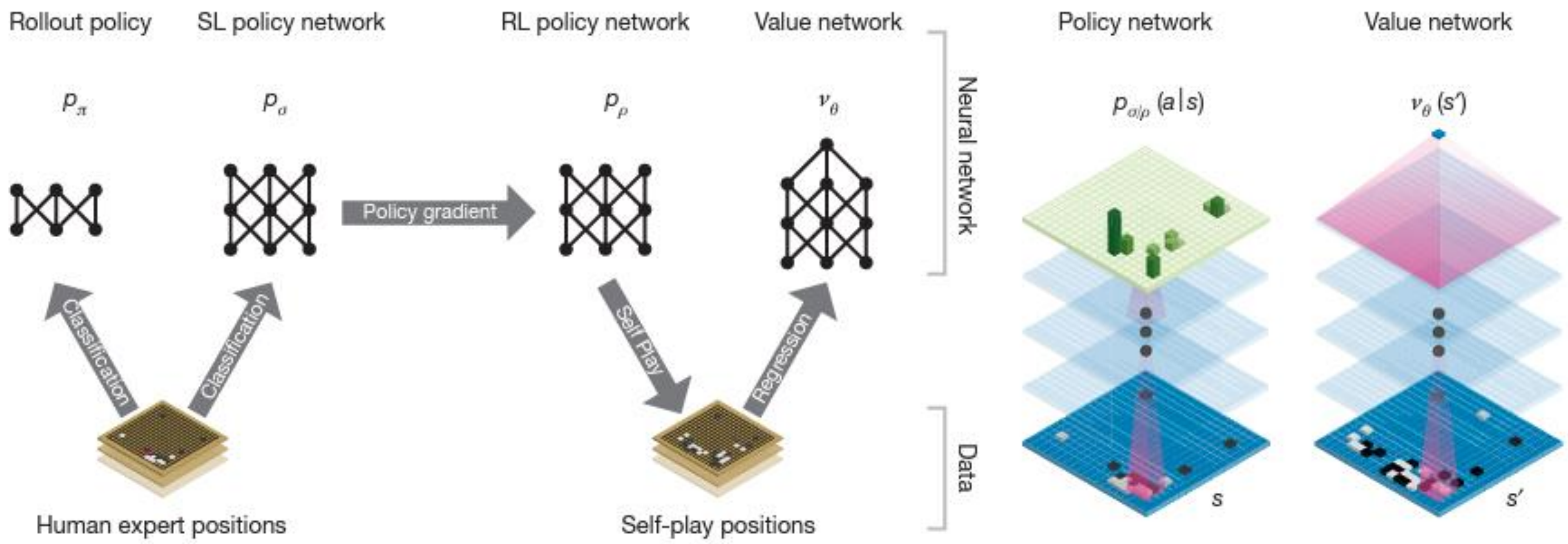
- TD-Gammon (Tesauro, 1992) *approximated* the eval function using a neural net
- Weights were learned via semi-gradient TD learning on self-play
- Incorporated human expert data for hand-designed features





AlphaGo

- Go has much larger game tree than chess, no high-performance eval function
- AlphaGo used MCTS with rollout and selection policies trained on human expert data
- SL network improved using RL, then evaluated using self play to obtain value network
- MCTS values updated using combination of rollout result and value network output

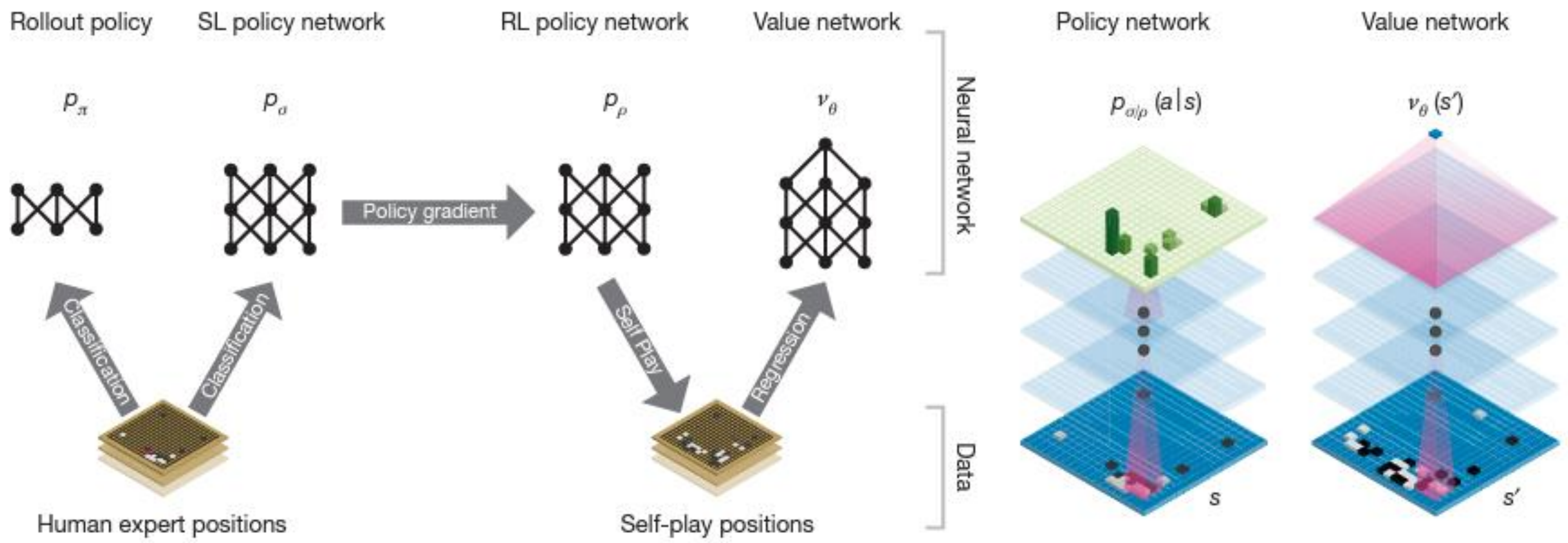


https://deepmind.google/technologies/alphago/

AlphaGo

- AlphaGo used a combination of reinforcement learning with MCTS to master Go
- Human expert data was first used to train a *policy network* (board state -> action) using supervised learning (classification)
- Policy network was significantly improved via RL and self-play

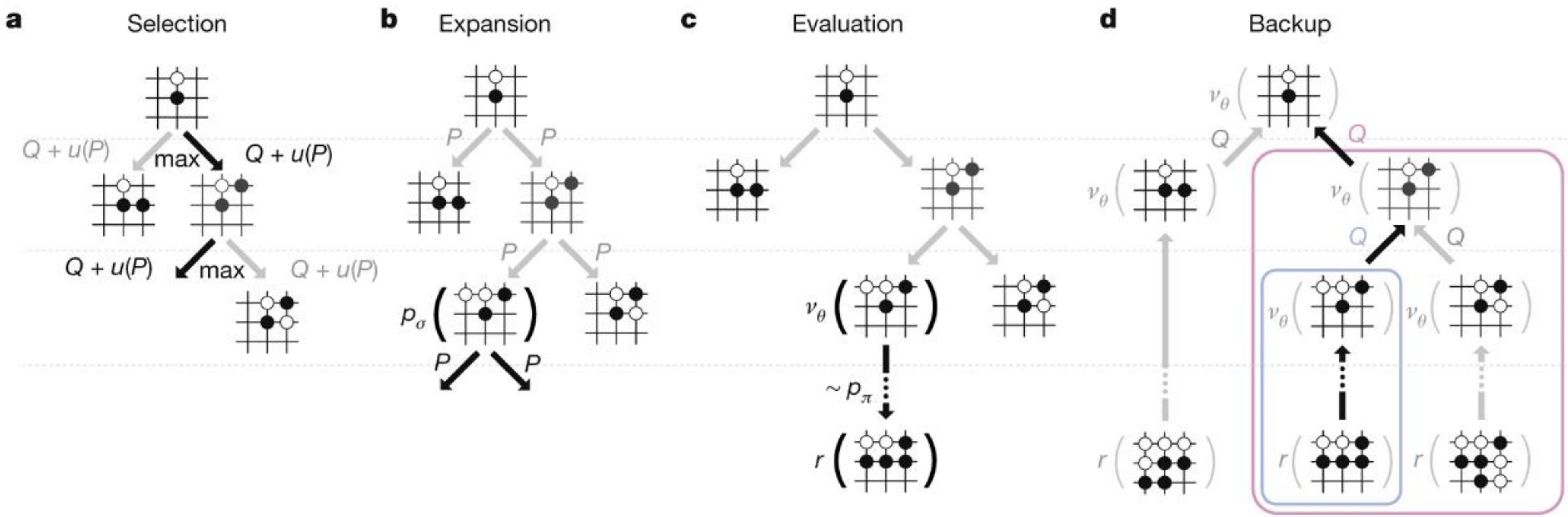
- Policy network was then used to generate *simulated* data from self-play
- Finally, regression was used to train a *value network* (board -> value)



https://deepmind.google/technologies/alphago/

AlphaGo

- During actual gameplay, AlphaGo ran MCTS using its hard-trained network functions
- Node values are weighted averages of value network outputs and simulated values
- Policy network is used for encouraging exploration during the selection step as well as the rollout steps during simulation (evaluation)



Deep Q-Network

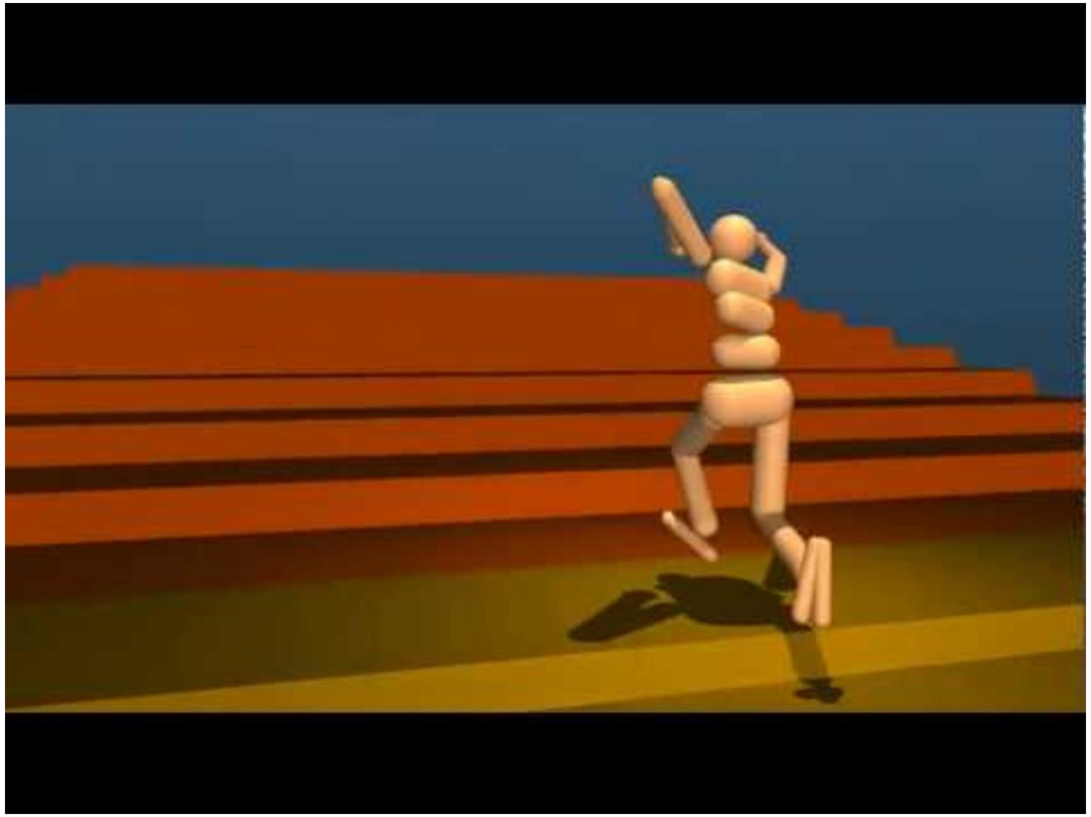
- Combination of Q-learning with deep convolutional neural networks (CNNs)
- Raw inputs in the form of video game streams: Automated feature design

- DQN achieved human level play on a few dozen different arcade video games

- Learned different policies to cope with different dynamics and reward structures
- No game-specific modifications!

https://deepmind.com/blog/article/deep-reinforcement-learning

RL for Locomotion



https://www.youtube.com/watch?v=hx_bgoTF7bs

Multi-Agent RL

https://openai.com/blog/emergent-tool-use/

Summary

- Model-based RL methods use raw data to learn a model
- Models can be used to generate simulated data and supplement real data

- Function approximations are important for generalization when tabular representations are not feasible
- Learning via different forms of gradient descent

- Many modern applications of RL using deep neural networks