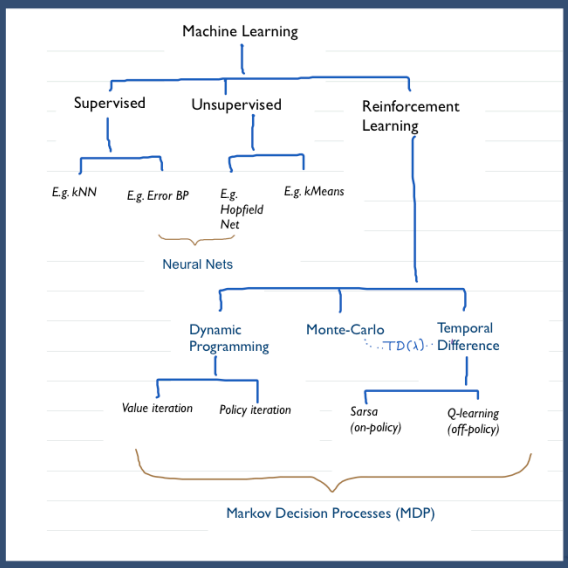

Reinforcement Learning
October 2011
EECE 592 -Reinforcement Learning

Taxonomy of Machine Learning



Introduction

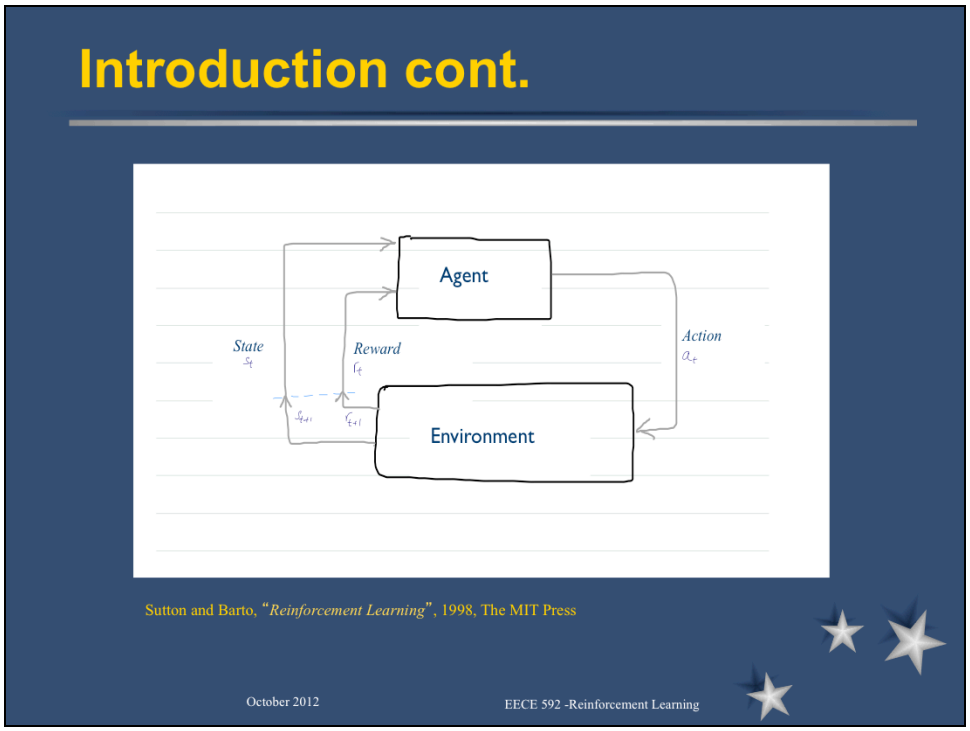
There are traditionally two different learning paradigms; *unsupervised learning* and *supervised learning*. This chapter, based upon the book by Sutton & Barto (1998), introduces a third; *reinforcement learning* (RL). RL is characterized as learning that takes place via interaction of a learning agent with its environment. Like supervised learning a feedback signal is required, however for RL, this feedback signal represents a *reward*, not a desired value. RL is therefore not supervised learning. The purpose of RL is to learn which actions yield the greatest reward.

Examples of RL

The following are examples of RL:

> Amongst other things, a good chess player uses intuitive judgment to decide the next move. Over time, the player refines this intuition based upon the outcome of each game.
>
> To control temperature and stirring rates of a bioreactor.

Both of these examples illustrate reinforcement learning. There is no teacher or supervisor. However there is a reward, which in the case of a game of chess is the result (a win, a loss or a draw) and in the case of the bioreactor is a



The environment is modelled using discrete time steps, $t = \{0,1,2,3.....T\}$

The agent is able to sample the environment at each/any time step to determine the state of the system $s_t \in S$

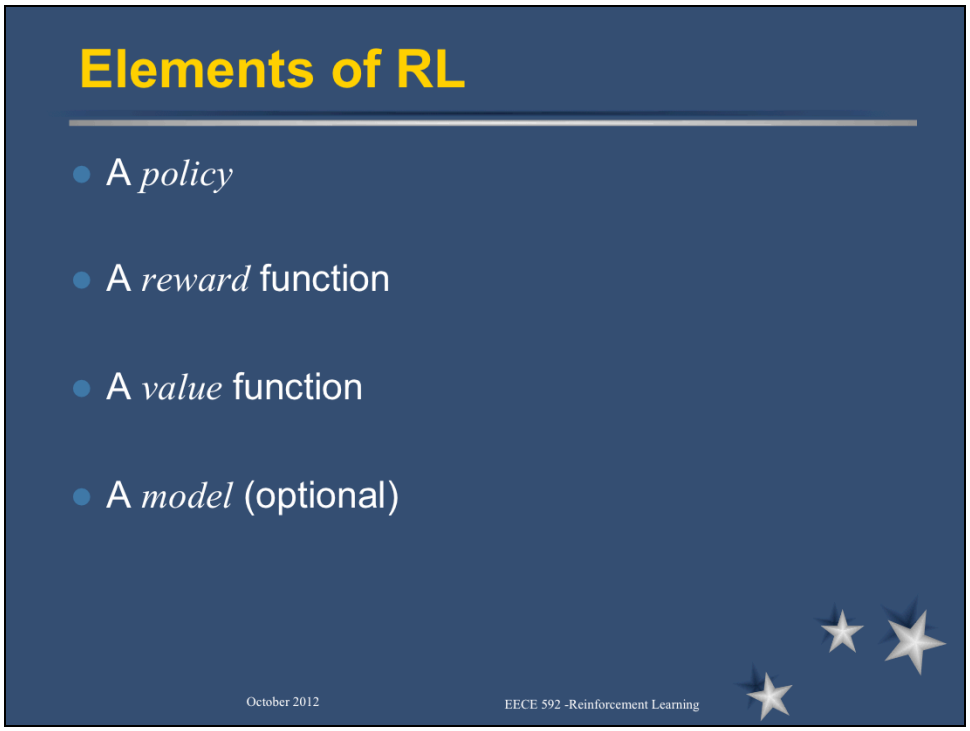
where S is the set of all possible states.

The agent selects an action $a_t \in A(s_t)$

where $A(s_t)$ is the set of actions that are possible in state s_t.

Then one time step later, the agent finds itself in a new state, s_{t+1} and receives a reward r_{t+1}

where $r_{t+1} \in \Re$



The ***policy*** represents the learning agents behaviour and can be thought of as a mapping from perceived environmental states to actions.

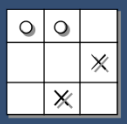
A ***reward*** function defines the goal. Given any state of the environment, the reward function identifies how good or bad this state is.

The difference between a ***value*** function and a reward function is subtle. Whereas the reward function measures the reward associated with the immediate state of the system, a value function represents a prediction of reward at some future time, based on the current state. It takes into account all future rewards that may ensue as a result of reaching the current state.

In RL, a ***model*** is typically the set of probability functions that determine how an agent would behave in any given state of the system. While it is a requirement for some types of RL (e.g. Dynamic Programming), it is not necessary for the type of learning that will be addressed in this course.

Example: Tic-Tac-Toe

- The *policy* defines how the next move is selected
- The result, a win a loss or a draw, is the *reward*
- A function that returns the probability of winning for a given state is the *value* function
- No model is used.



"Tic-Tac-Toe"or "Noughts-and-Crosses" is a simple game played on a 3x3 grid. The objective is to be the first to make a line of crosses or noughts across the grid.

The environment in this case, is the board. The reward function is simple, but can only be defined for the final state to indicate a win, a loss or a draw. Given that the environment of all possible states is small and reasonably manageable, the value function can be implemented as a table indexed by all possible states. (One may even wish to take advantage of the symmetrical nature of the board and reduce the number of actual states referenced in this table). The policy takes into account the rules of the game and its choices are dictated by all possible states that it can select given its current state.

The effect of RL is to learn a value function that can reliably predict if a state (the next move) will result in a win or not. If a loss and draw are considered to be equal, then the terminal states in the table that represents the value function can be set to 1 or 0. All others would be initialized to 0.5 (i.e. probability of a win is known to be no better than chance).

The Value Function

- A value function can be a table of estimates.

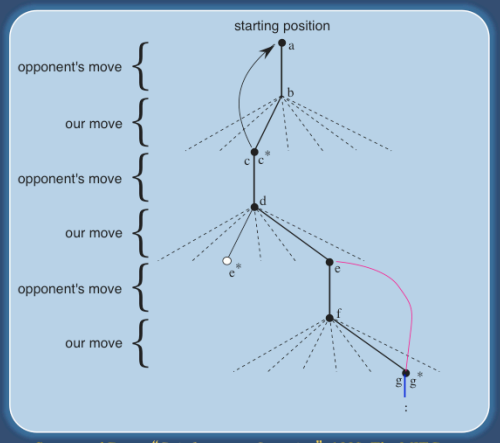
State	Probability of a win (Computer plays "o")
	0.5
	0.5
	1.0
	0.0
	0.5
etc	

Tic-tac-toe has a small finite number of states and the value function can be implemented as a look-up table. In the above figure, each separate board position has an associated probability.

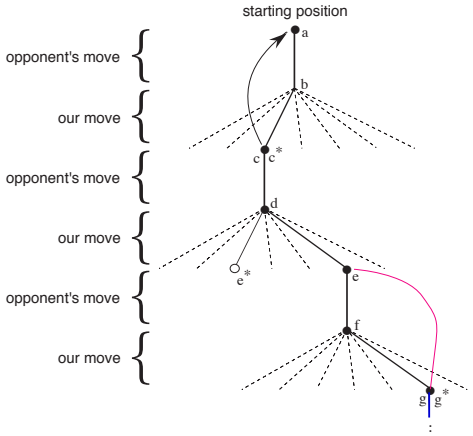
As play proceeds, the computer will perform a search of all possible moves and select that move for which the value function indicates the greatest likelihood of winning. Given that the value function will return 0.5 for all initial states, the earliest games in the learning process will be unlikely to win against an intelligent player.

Tic-Tac-Toe : Moving

- A move is selected based on the value function.
- Moves may be *greedy* or *exploratory*



Sutton and Barto, "*Reinforcement Learning*", 1998, The MIT Press

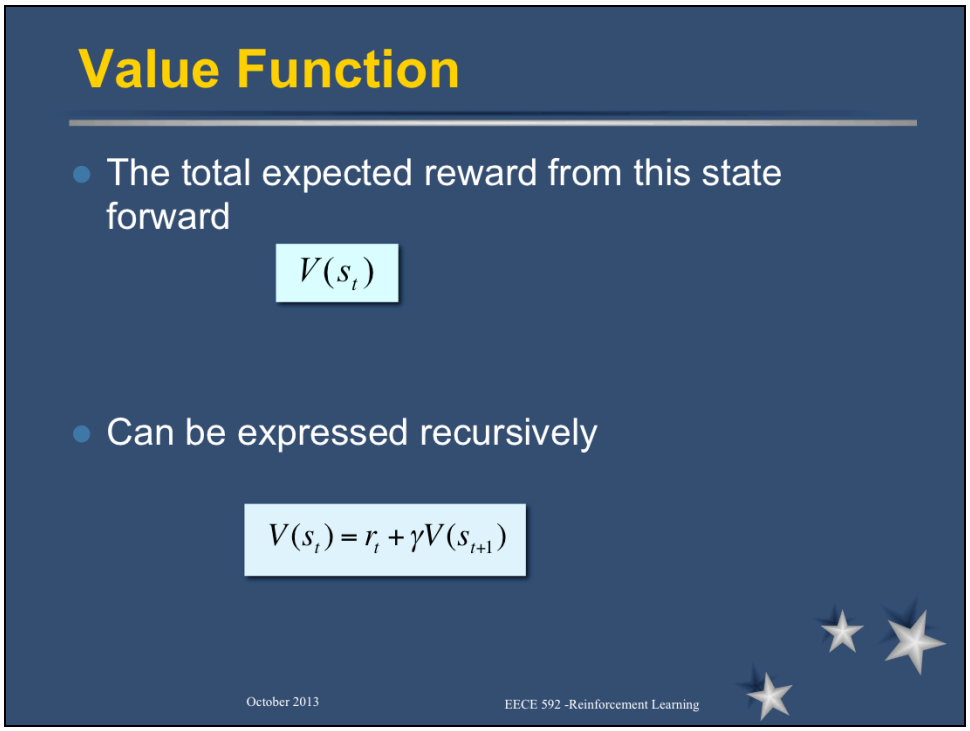


The figure shows a sequence of moves. Most moves will be *greedy* moves, i.e. selection of a move that has the highest value. However it is useful at times to select moves randomly. These moves, although not necessarily the best, are termed *exploratory* moves as they help to experience states that would otherwise not be encountered.

During play, the learning process updates the entries in the table, attempting to improve their accuracy. Over time and after playing many games, the value function will return values that represent the true probabilities of winning from each state.

Tic-Tac-Toe

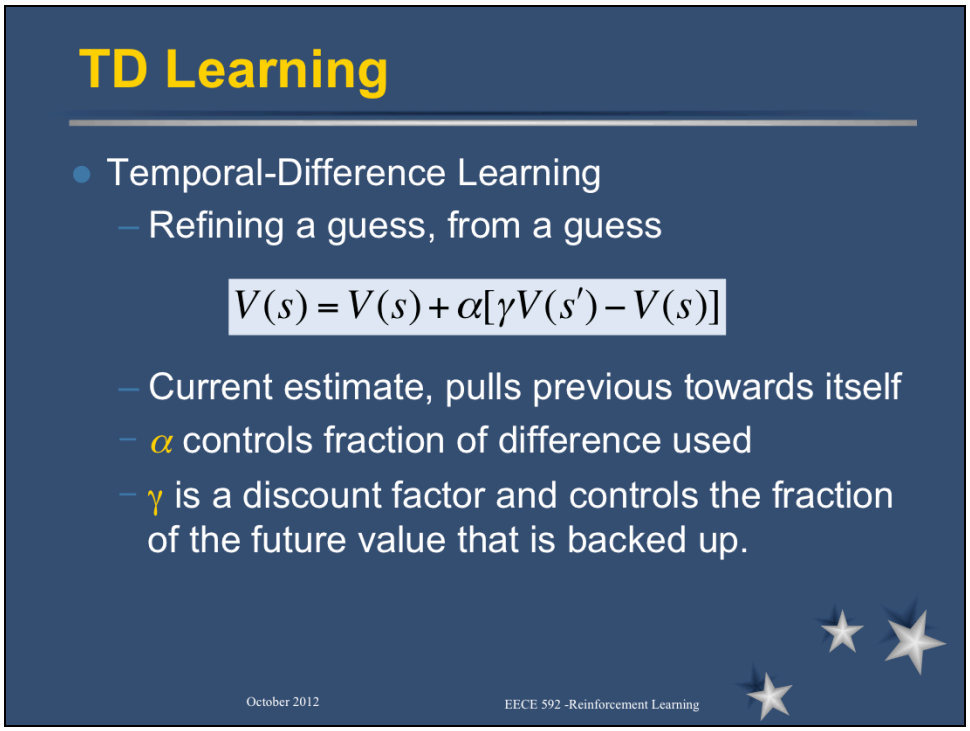
- Notice that for an exploratory move…
  - ..the diagram shows no learning. So what are the options?:
    - Perform no learning *or*
    - Learn from e to c* *or* "*on-policy*"
    - Learn from e* to c* *or* "*off-policy*"



$$V(s_t) = r_t + \gamma V(s_{t+1})$$

This basically means add up all future rewards until the terminal state:

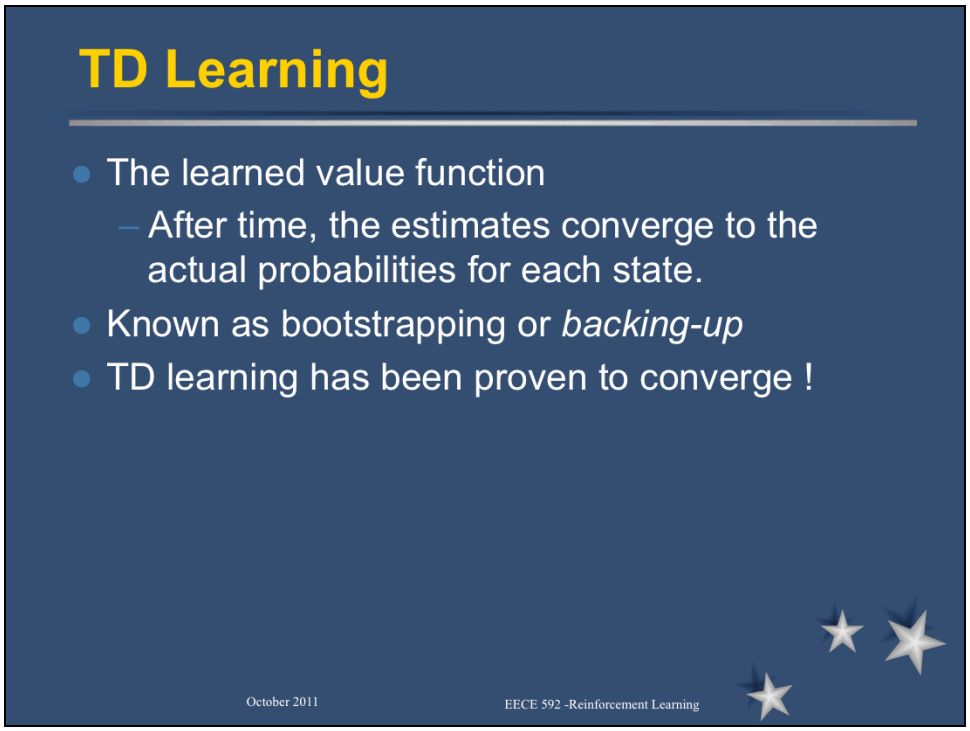
$$V(s_t) = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \gamma^3 r_{t+3} \ldots\ldots \gamma^{t+n} r_T$$



Learning involves updating the table of probabilities representing the value function. This actually happens during play as mentioned previously and can be best described as updating a guess, using a guess! The approach is known as temporal difference learning. The idea is to move the estimate of the state *s* before the move, a fraction of the way to the state *s'* after the move. An example of an update rule based on temporal differences is given below.

$$V(s) = V(s) + \alpha[\gamma V(s') - V(s)]$$

Where $V(s)$ is the value of the state before the move and $V(s')$ is the value of the state after the move. Alpha represents the learning rate parameter and is typically decayed gradually to zero. If this step-size parameter is not reduced over time, then the system is able to adjust to different styles of opponent play. Gamma is a discount factor and is used to specify how much influence future values have. In this Tic-Tac-Toe example, it has a value of 1.

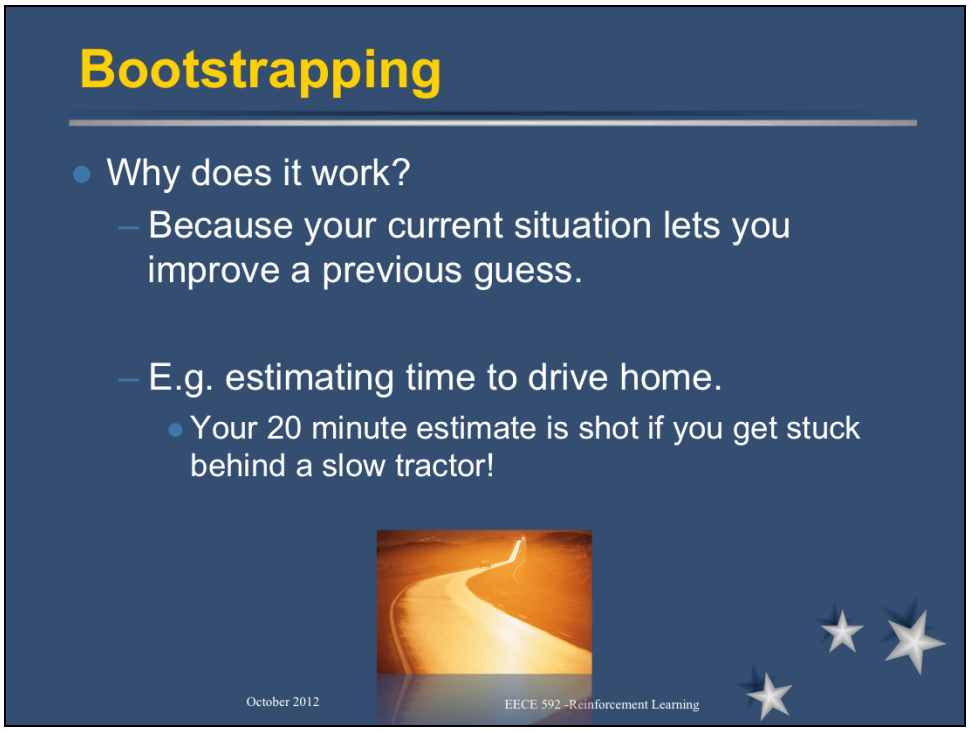


Learning a Guess, from a Guess

So for example, in the previous figure, *e* and *g* show the state of the game before and after a move by the computer. The value of the state function for *e*, $V(e)$ will be modified so that it is closer to the value of the state function at *g*, $V(g)$. Again, note that $V(g)$ is in fact only an estimate of the actual probability for that state; it is a guess. The learning step is called *temporal-difference* because the changes are based on the difference between two estimates at two different times. The reward signal is not available until the final state of the game is reached. Only at this time is the value for $V(s)$ an actual known value, a 0 for a losing and a 1 for a winning state.

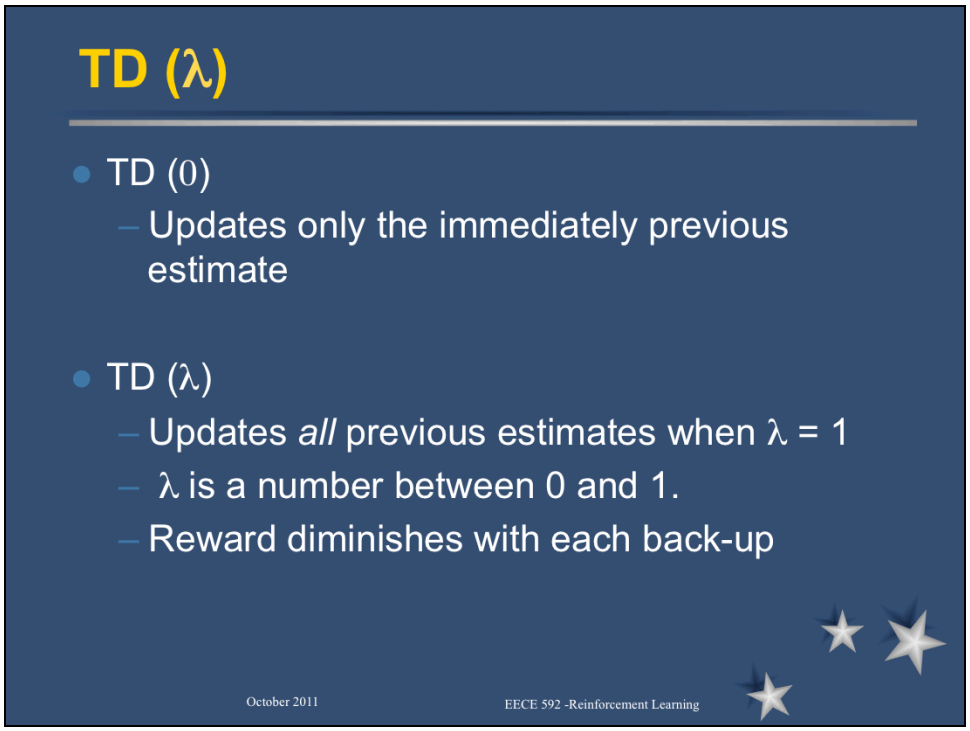
Bootstrapping

The simple example illustrates some key features of reinforcement learning. First of all, the interesting thing is that it is not necessary to wait for a known reward before an update can be made. If this were the case then learning could not take place until each game, or *episode*, had been completed. Instead, by *bootstrapping*, or backing up, it is possible to refine estimates on the fly. This enables learning to take place during play and perhaps evolve as play takes place. Secondly, game play involves a clear goal, which requires planning and foresight. It is incredible that reinforcement learning exhibits the effects of planning and look ahead without actually doing so. It appears intelligent!



Why does bootstrapping or learning a guess from a guess work? This is an interesting question and it is not really obvious why such an approach works. An example will help to illustrate this. Each day when you drive home, you are able to estimate how long it will take to get home. However, it might be a rainy day so you revise your estimate and add 5 minutes to the time; after all, experience tells you that traffic tends to be slower in wet weather. Later on you find that actually you're making good time on the highway so you revise your estimate and knock off a few minutes. Later however, you get stuck behind a slow truck and you again update your estimate. This is all an example of temporal difference learning at work. Your current situation lets you improve a previous guess. You thought the rain would slow you down, but actually you made good time, so actually the rain didn't slow you down that much say. Remember, the actual reward signal, the actual time it took you to get home is not available, until you get home. The ability to update estimates before availability of a reward signal can be very beneficial. For example, some episodes (e.g. a complete game) can be quite long, delaying learning until the episode is complete.

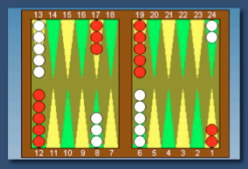
A mathematical proof of TD learning would be more convincing and although this is beyond the scope of this text, TD has been shown to always converge to a solution. I.e. for any fixed policy π, the TD learning algorithm has been



The basic formula for learning using temporal-differences states that the estimate of value function for the immediately previous state, $V(s)$, be moved closer to the estimate of the value function for the next state $V(s')$. I.e., only the immediately previous estimate is updated, but what about previous estimates? A simple variation of TD learning, usually referred to in the literature as TD(λ) learning does exactly this, where λ represents a number between 0 and 1. When the value is 0, TD(λ) degenerates to the basic learning step where only the previous estimate is updated. When λ is 1, *all* previous estimates are updated. However, each time the reward signal is *backed-up*, it is discounted so that it's influence slowly diminishes. TD(λ) is the learning algorithm used in the very successful TD-Gammon project discussed next.

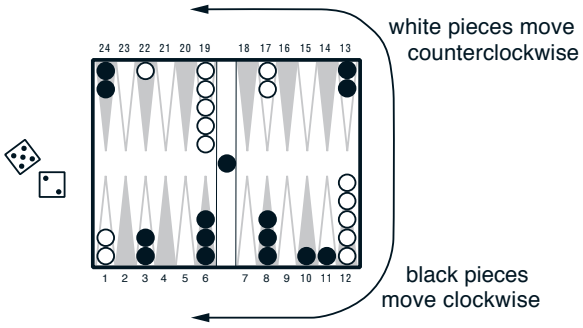
TD-Gammon

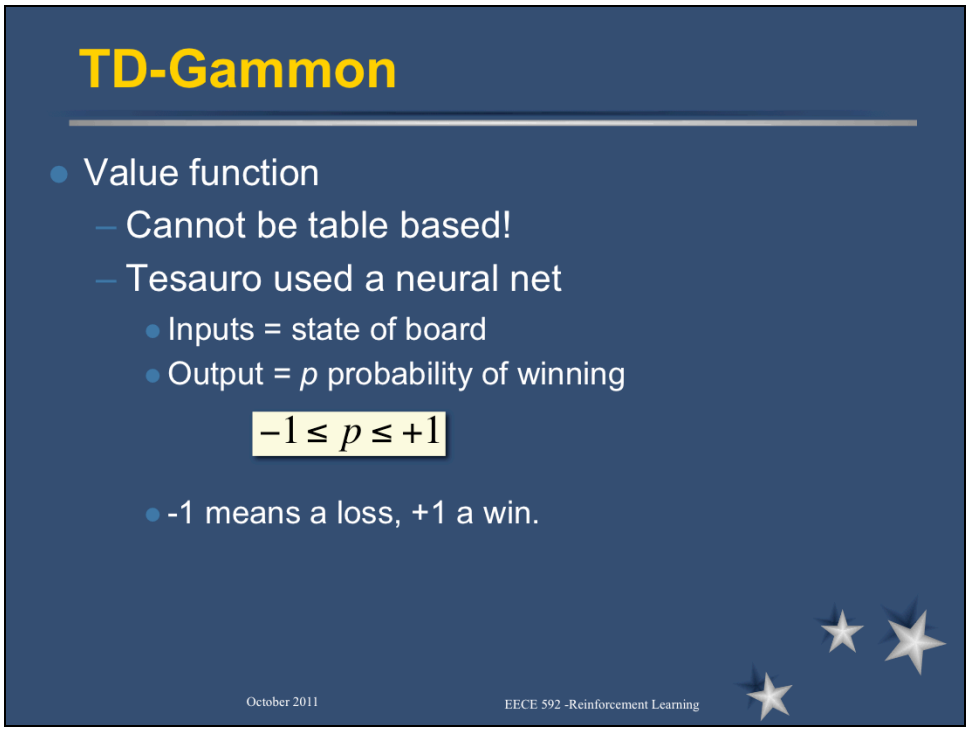
- Backgammon
  - 1000 years older than chess
  - Over 10^{20} states!
  - Many hundred ply branching factor. (Chess is only 30-40)
  - Depth based search not effective - difficult to automate



The Game of Backgammon

First, it is useful to emphasis some of the main characteristics the game. Backgammon is an ancient two-player game (at least a thousand years older than chess, according to some estimates) that is played on an effectively one-dimensional track. The players take turns rolling dice and moving their checkers in opposite directions along the track as allowed by the dice roll. The first player to move all his checkers forward and off his end of the board is the winner. In addition, the player wins double the normal stake if the opponent has not taken any checkers off; this is called winning a "gammon."



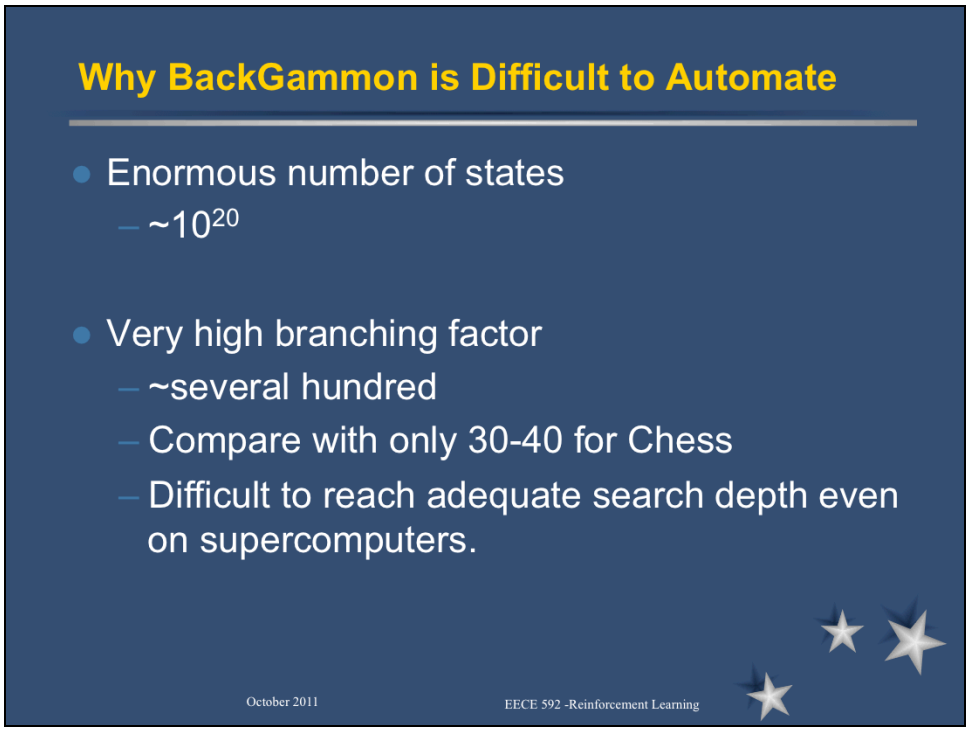


Learning to Play Games

Tic-Tac-Toe illustrates how RL can be applied to simple games. In this case, the value function can be implemented as a simple look-up table. Furthermore, given the symmetry of many of the board positions, the size of this table can be somewhat compressed. However, it's interesting to apply RL to more complex games too. For example, the ability to learn to play complex games such as chess has been an open-ended problem in artificial intelligence. Clearly a look-up table is not practical in this case; the number of possible states is simply too great. This is where neural nets come in. The ability of neural network approaches to learn from example has thus attracted much attention in this respect, with varying results. Two prominent attempts, one in chess and another in backgammon will be discussed. Parts of the following are based on*:* *Temporal Difference Learning and TD-Gammon* By Gerald Tesauro, Communications of the ACM, March 1995 / Vol. 38, No. 3

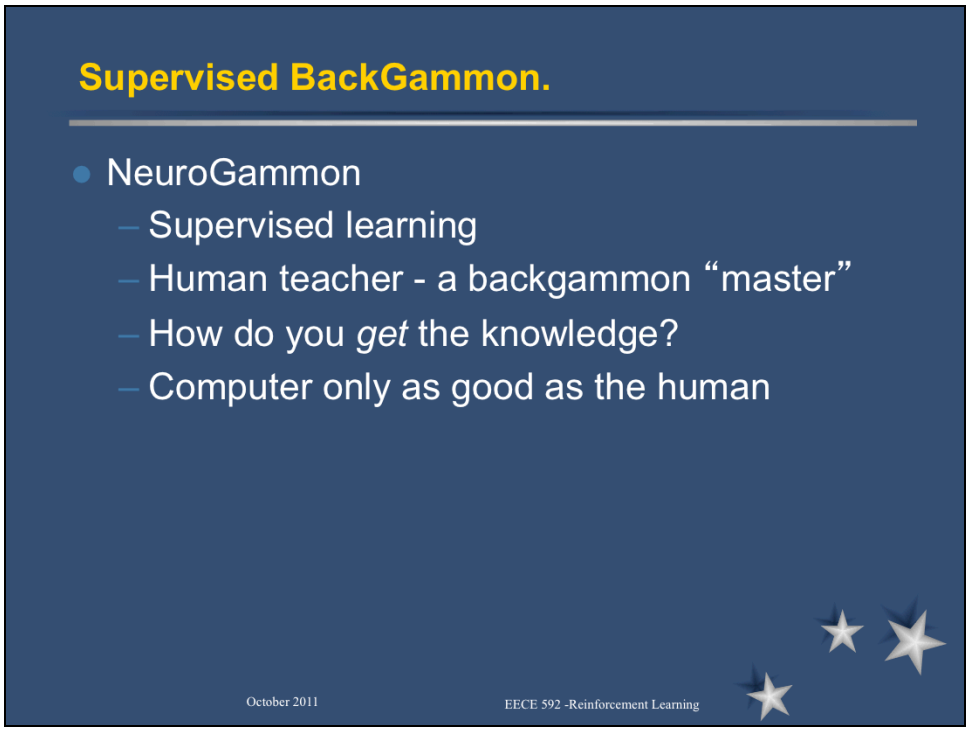
Gerald Tesauro's TD-Gammon

TD-Gammon is truly a milestone in the field of AI and machine learning. Until recently, the ability of a machine to learn to become an expert by simply playing against itself has only been a dream. Yet Tesauro and TD-Gammon show how a neural net and TD learning can attain grand master levels of play! So much so that the 4000 or so weights that were arrived at are now kept as a



Why is Learning to Play Difficult to Automate?

Programming a computer to play high-level backgammon has been found to be a rather difficult undertaking. In certain simplified endgame situations, it is possible to design a program that plays perfectly via table look-up. However, such an approach is not feasible for the full game, due to the enormous number of possible states (estimated at over 10 to the power of 20). Furthermore, the brute-force methodology of deep searches, which has worked so well in games such as chess, is not feasible due to the high branching ratio resulting from the probabilistic dice rolls. At each ply there are 21 dice combinations possible, with an average of about 20 legal moves per dice combination, resulting in a branching ratio of several hundred per ply. This is much larger than in checkers and chess (typical branching ratios quoted for these games are 8-10 for checkers and 30-40 for chess), and too large to reach significant depth even on the fastest available supercomputers.

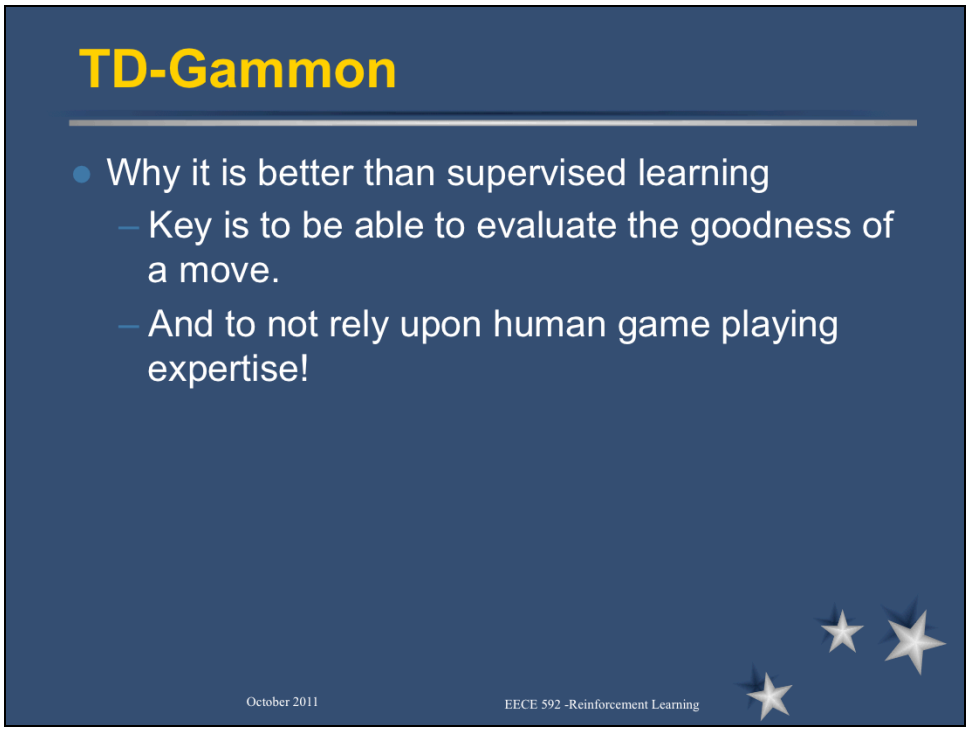


The Problem with Experts Teaching a Computer

However, such a supervised approach relies on human expertise and building human expertise into an evaluation function, whether by knowledge engineering or by supervised training, has been found to be an extraordinarily difficult undertaking, fraught with many potential pitfalls. A further problem is that the human expertise that is being emulated is not infallible. As human knowledge and understanding of a game increase, the concepts employed by experts and the weightings associated with those concepts undergo continual change. This has been especially true in Othello (aka Reversi) and in backgammon, where over the last 20 years, there has been a substantial revision in the way experts evaluate positions. Many strongly held beliefs of the past, which were held with near unanimity among experts, are now believed equally strongly to be erroneous. In view of this, programmers are not exactly on firm ground in accepting current expert opinions at face value.

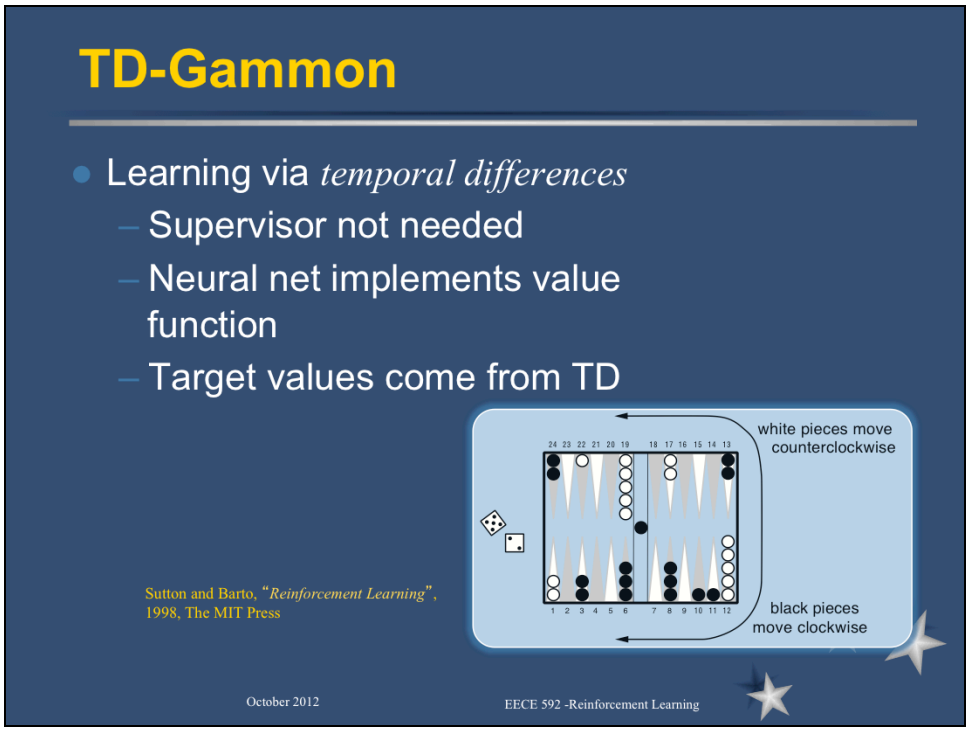
NeuroGammon

NeuroGammon was the predecessor of TD-Gammon and although it was a strong player, did not reach master level. It was subject to the problems described above and as such could only be as good as its' teacher. Supervised learning of multi-layer perceptrons certainly provides a convincing capacity to learn to play the game. The problem was that the supervisor, being human brings with it all the problems associated with knowledge engineering.



Evaluating a Move

The ability to evaluate the goodness of a move is key to becoming a strong contender. In computing terms this typically means an evaluation function which can return a value that indicates the probability of a win given a specific move. In neural net terminology this problem can be formulated as a supervised learning task where the inputs represent the board positions for the next proposed move and the output is a single value between, say -1 and +1 indicating the probability of winning from this move.

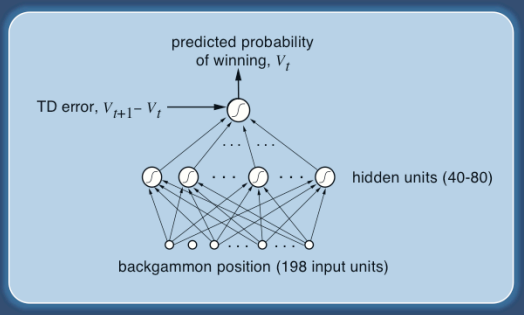


Temporal Difference Learning.

The "TD" in TD-Gammon stands for *Temporal Difference*. These are a class of methods for solving the temporal credit assignment problem. The basic idea of TD methods is that the learning is based on the difference between temporally successive predictions. In other words, the goal of learning is to make the learner's current prediction for the current input pattern more closely match the next prediction at the next time step. The most recent of these TD methods is an algorithm proposed by [Sutton, R. S. *Learning to predict by the methods of temporal differences.* Mach. Learning **3**, (1988), 9-44.] for training multi-layer neural networks called TD(lambda). No longer is a supervisor needed. Instead, the target values for our network are generated automatically via TD.

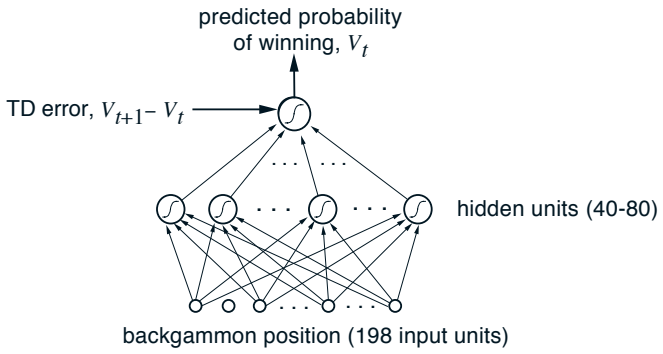
TD-Gammon

- TD Gammon



Sutton and Barto, "*Reinforcement Learning*", 1998, The MIT Press

At the heart of TD-Gammon is a multi-layer perceptron trained via the backpropagation learning algorithm. This neural net is used to implement the value function.



TD-Gammon

- NN input representation
  - 4 inputs for each player per rung
  - 2 players + 24 rungs
  - ~200 inputs

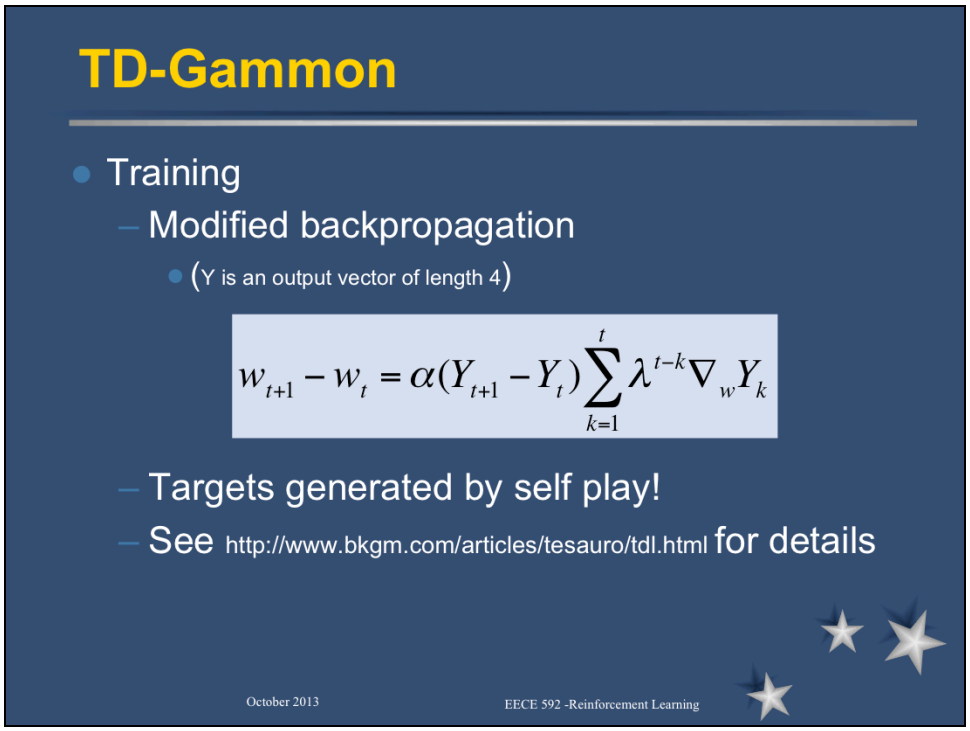
Number of pieces on rung	Encoding
0	0 0 0 0
1	1 0 0 0
2	1 1 0 0
3	1 1 1 0
n (n>3)	1 1 1 (n-3)/2

Tesauro used 4 inputs to represent the number of each players pieces on each of the 24 rungs, plus 2 units to indicate how many are on the bar and how many off. There were additional units to encode the number of pieces already removed from the board and finally two more to indicate if it was white or blacks turn to move.

The four units indicated the number of pieces on the rung. The table shows the encoding. There were four inputs to encode each player's pieces, so there were 8 inputs for each rung. These were fully connected to between 40 and 80 hidden units. A single output neuron was used to indicate the probability of winning. In total there were around 4000 trainable weights.

TD-Gammon

- Hidden layer
  - 40 to 80 neurons
- Output layer
  - Single output (probability)



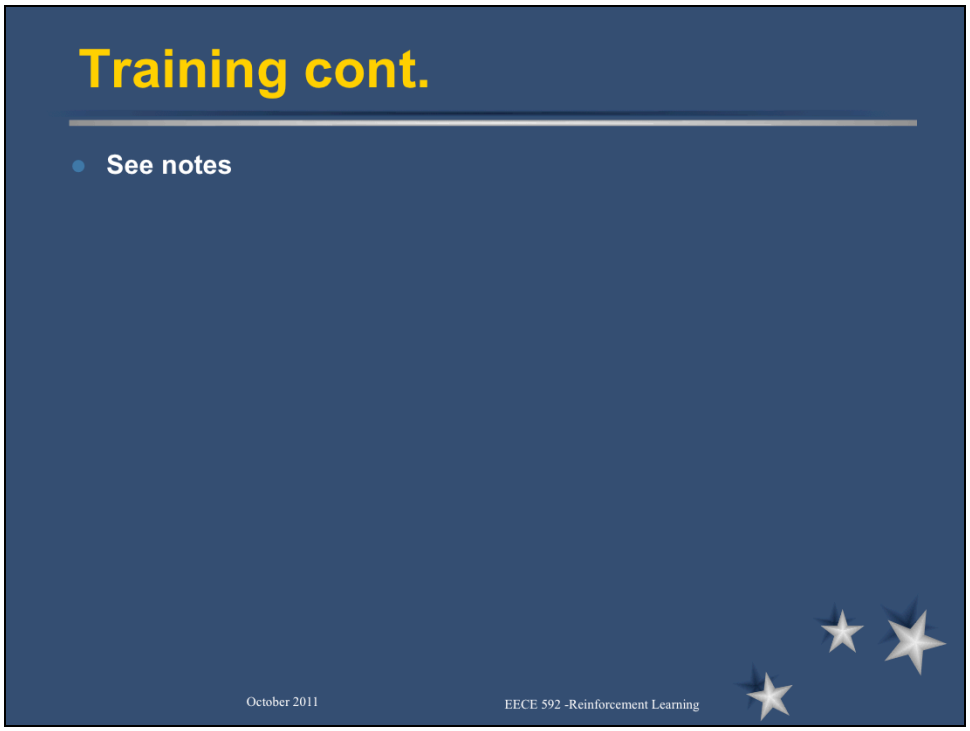
The basic methodology is simple. A neural network value function is used to rate the *goodness* of a move, i.e. how likely that move is to result in a win. Selection of a move is made by applying a single or 2-ply search. The evaluation function can then be applied to all candidate moves and the best move identified.

Modified Backpropagation

$$w_{t+1} - w_t = \alpha(Y_{t+1} - Y_t)\sum_{k=1}^{t} \lambda^{t-k} \nabla_w Y_k$$

The training of the net is a variation of the backpropagation algorithm modified for temporal differences. α is the learning rate. The target Y_t is a vector of length 4 corresponding to the 4 possible outcomes of either white winning with a normal or gammon or black winning, normal or gammon. $\nabla_w Y_k$ is the gradient to be followed. λ^{t-k} is a coefficient that controls how far back the current error is propagated. When $\lambda=0$, only one back-step is computed. When $\lambda=1$, the updates back-step all the way to the beginning.

Consider the computer's very last move. In this case it is likely trivial to indicate whether the computer is going to win lose or draw. This then becomes the target value, However, what is the target value for all subsequent moves? This comes from the TD approach and can be thought of as a way of propagating a prediction back through time.



The Algorithm

The training procedure for TD-Gammon is as follows: the network observes a sequence of board positions starting at the opening position and ending in a terminal position characterized by one side having removed all its checkers. The board positions are encoded and fed as input vectors x[1], x[2], . . . , x[f] to the neural network. (Each time step in the sequence corresponds to a move made by one side, i.e., a "ply" or a "half-move" in game-playing terminology.) For each input pattern x[t] there is a neural network output vector Y[t] indicating the neural network's estimate of expected outcome for pattern x[t]. At each time step, the TD (lambda) algorithm is applied to change the network's weights as on the previous slide.

The Teacher

TD-Gammon was taught by board positions taken from self-play. While this would seem counter-intuitive, it turns out that the results were rather surprising! Initially of course, the performance was very poor and some of the early games would take a long while to complete. After all, the moves were being selected arbitrarily. After a few dozen games however, performance rapidly improved.

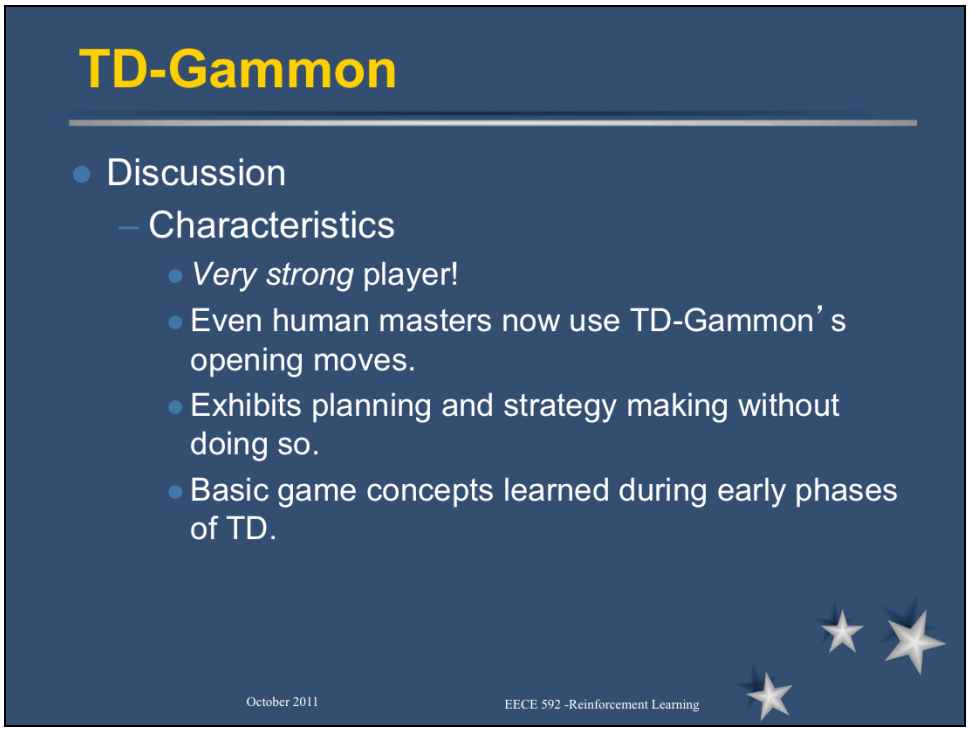
Performance

Training involved vast amounts of board positions. In fact, some of the best results were obtained by training a network with 80 neurons in the hidden layer with over 200,000 games!

TD-Gammon can beat Grand Masters

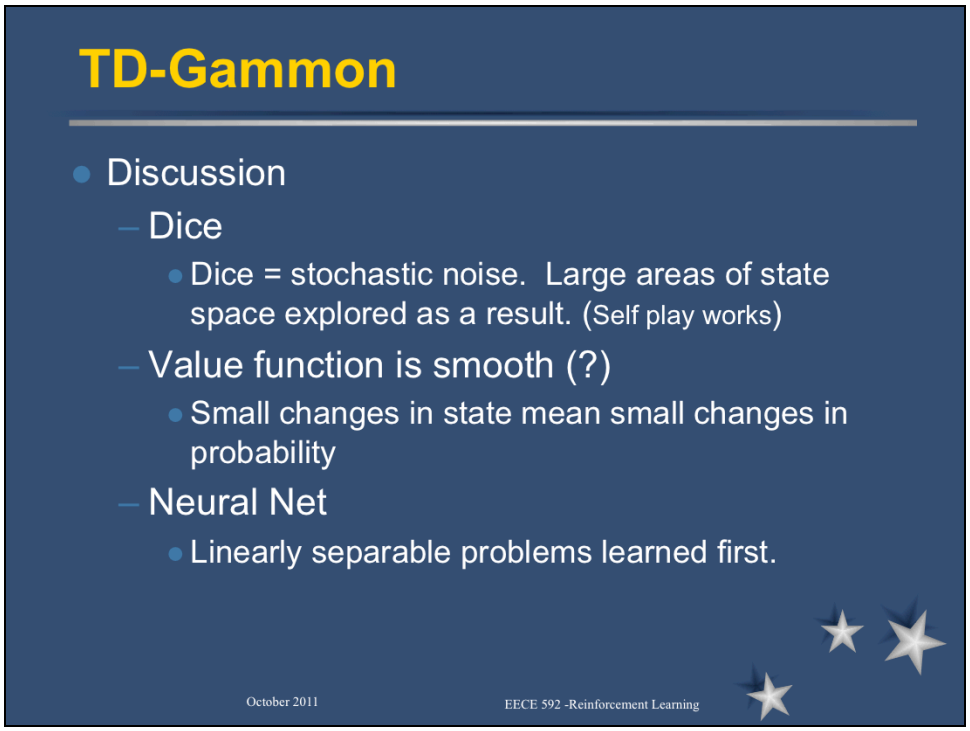
The following table shows TD-Gammon against a number of top world class players. For example, Kit Woolsey, is one of the game's most respected analysts and is perennially rated as one of the ten best players in the world. (He was rated fifth in the world in 1992).

Version 1.0 used 1-ply search whereas versions 2.0 and 2.1 used 2-ply. Version 1.0 and 2.0 hand 40 units and versions 1.0 and 2.1 had 80. According to an article by Bill Robertie published in Inside Backgammon Magazine, TD-Gammon's level of play is significantly better than any previous computer program. Robertie's overall assessment is that TD-Gammon 2.1 now plays at a strong master level that is extremely close (within a few hundredths of a point) to equaling the world's best human players. He thinks that in at least a few cases, the program has come up with some genuinely novel strategies that actually improve on the way top humans usually play. TD-Gammon's style of play frequently differs from traditional human strategies, and in at least some

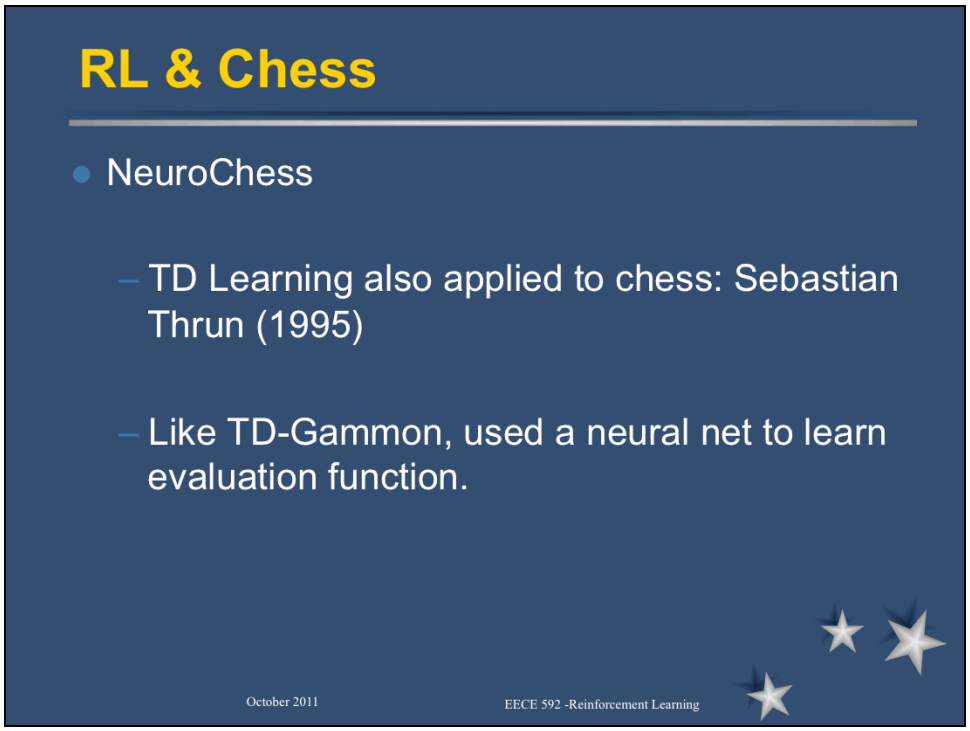


By combining the TD approach to temporal credit assignment with the MLP architecture for nonlinear function approximation, rather surprising results have been obtained, to say the least. It's incredible to believe that the system exhibits traits characteristic of planning and strategy making, yet all moves are based on a number generated on the instantaneous state of the board, no *thinking ahead* takes place. The TD self-play approach has greatly surpassed the alternative approach of supervised training on expert examples, and has achieved a level of play well beyond what one could have expected, based on prior theoretical and empirical work in reinforcement learning. Hence there is now considerable interest within the machine learning community in trying to extract the principles underlying the success of TD-Gammon's self-teaching process.

It's interesting to note that a complete understanding of the learning process is still far away, however some important insights have been obtained.

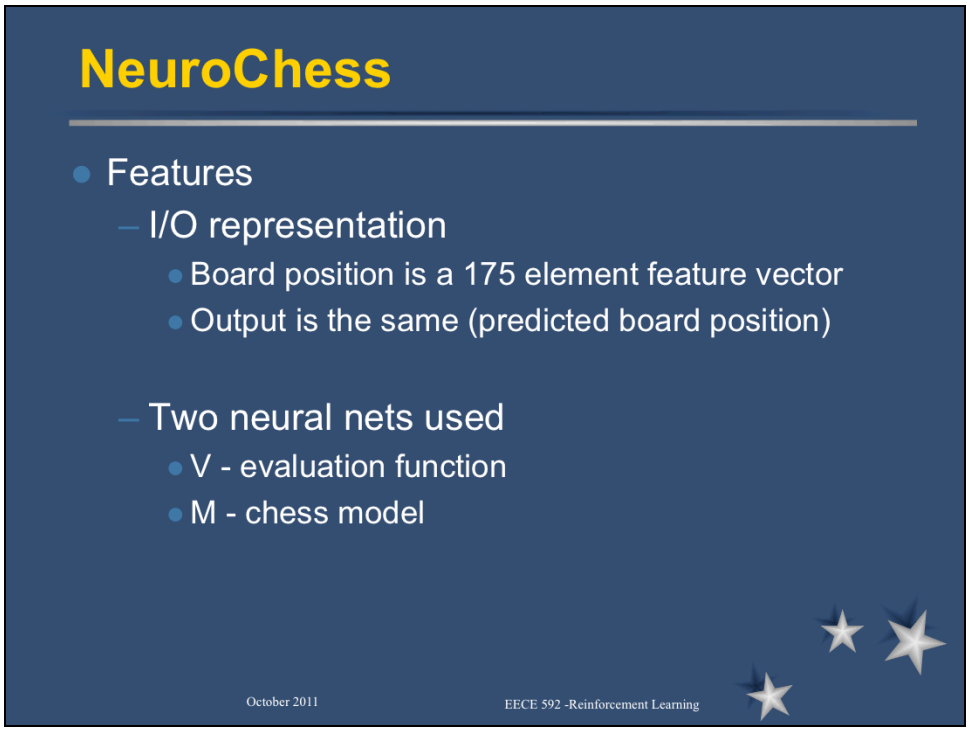


One feature of backgammon, believed to be significant in the success of TD-Gammon is dice. An important effect of the stochastic dice rolls is that they produce a high degree of variability in the positions seen during training. As a result, the learner explores more of the state space than it would in the absence of such a stochastic noise source, and a possible consequence could be the discovery of new strategies and improved evaluations. In contrast, in deterministic games such as chess, a system trained by self-play would restrict exploration to very narrow portions of the state space. Another conjecture is that for non-deterministic games, the evaluation function is real-valued exhibiting a great deal of smoothness and continuity, that is, small changes in the position produce small changes in the probability of winning. In contrast, for deterministic games like chess this function is discrete (win, lose, draw) and presumably more discontinuous and harder to learn.Its interesting to note that a close examination of the early phases of the learning process indicated that during the first few thousand training games, the network learns a number of elementary concepts, such as bearing off as many checkers as possible, hitting the opponent, playing safe. It turns out that these early elementary concepts can all be expressed by an evaluation function that is linear in the raw input variables. Thus what appears to be happening in the TD learning process is that the neural network first extracts the linear component of the evaluation function, while nonlinear concepts emerge later in learning. (This is also frequently seen in backpropagation: in many applications, when training a multi-layer net on a complex task, the network first extracts the linearly separable part of the problem.)



Given the tremendous success of TD-Gammon it was not surprising that similar approaches were applied to chess.

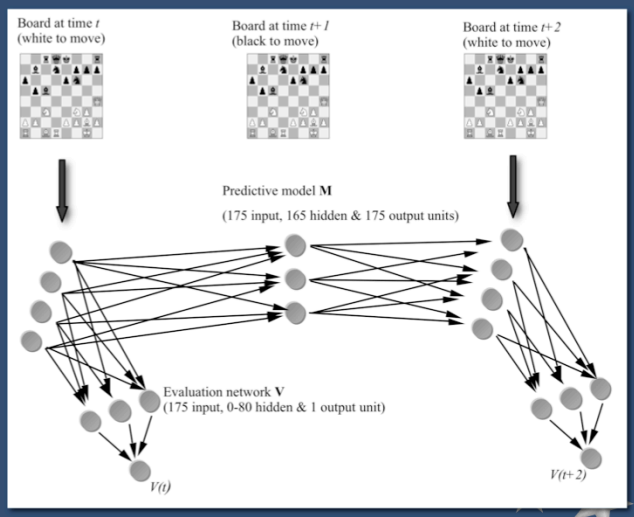
Sebastian Thrun (1995) was one of the first to attempt this. As with TD-Gammon, a search algorithm was used to select candidate moves and a neural net used to learn the value function.



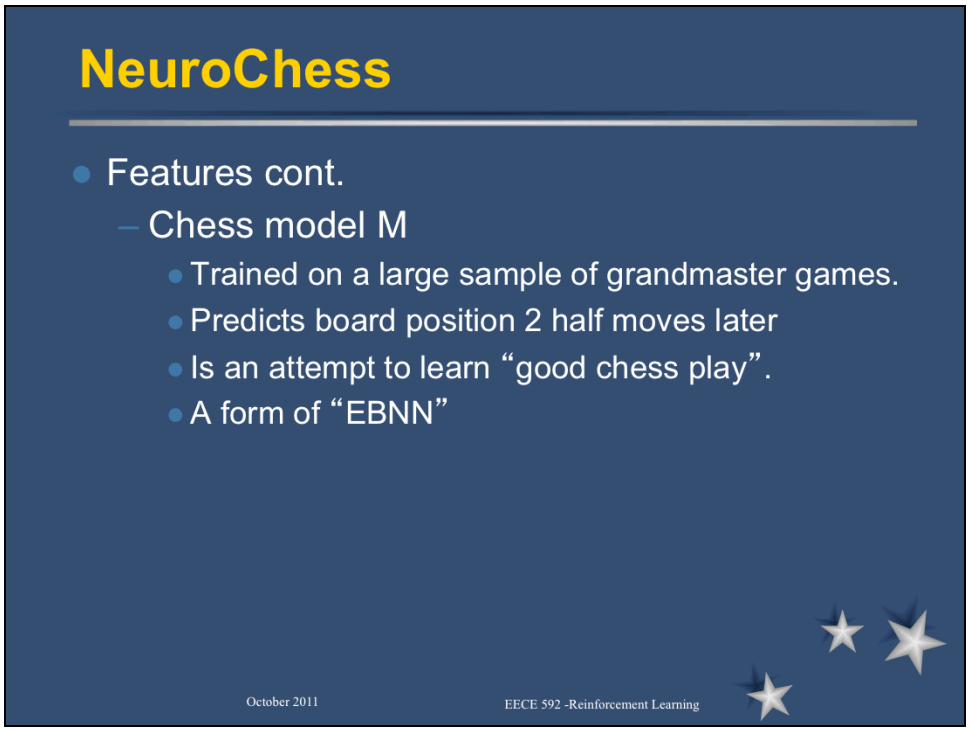
NeuroChess represents a chess position as a vector of 175 hand-written features. It has two neural networks, *V* and *M*. *V* is the value function, which takes the feature vector as input and produces an evaluation used to play chess. *M* is a chess model, which takes the feature vector as input and predicts its value two half-moves later. *M's* output is used in training *V*.

NeuroChess

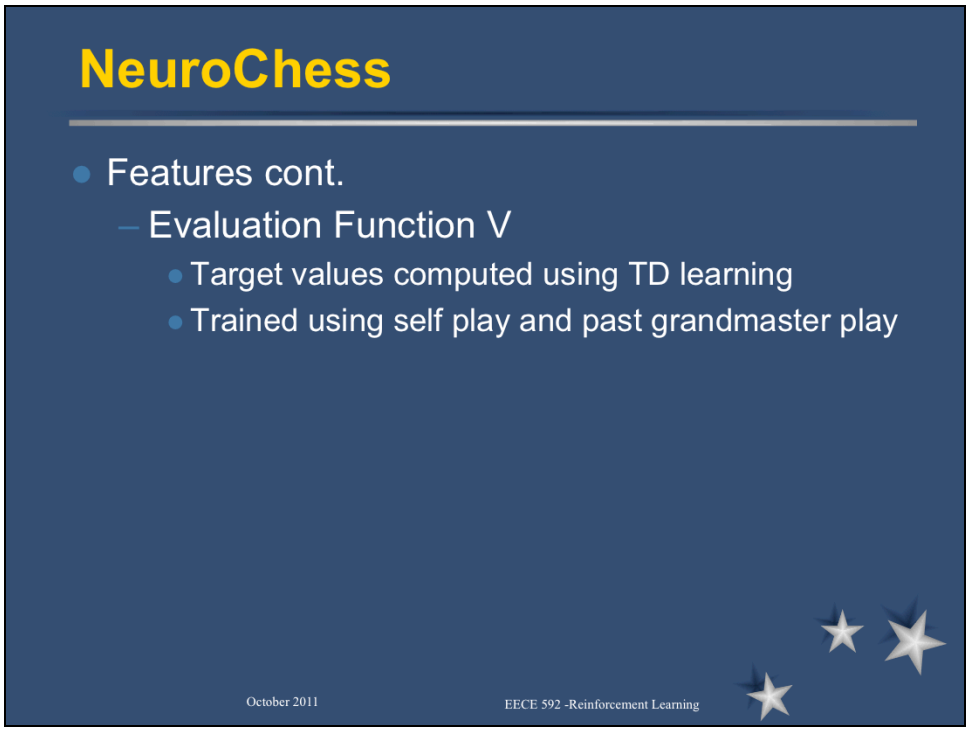
From: Thrun, S., 1995, *"Learning To Play the Game of Chess"*, in Advances in Neural Information Processing Systems 7



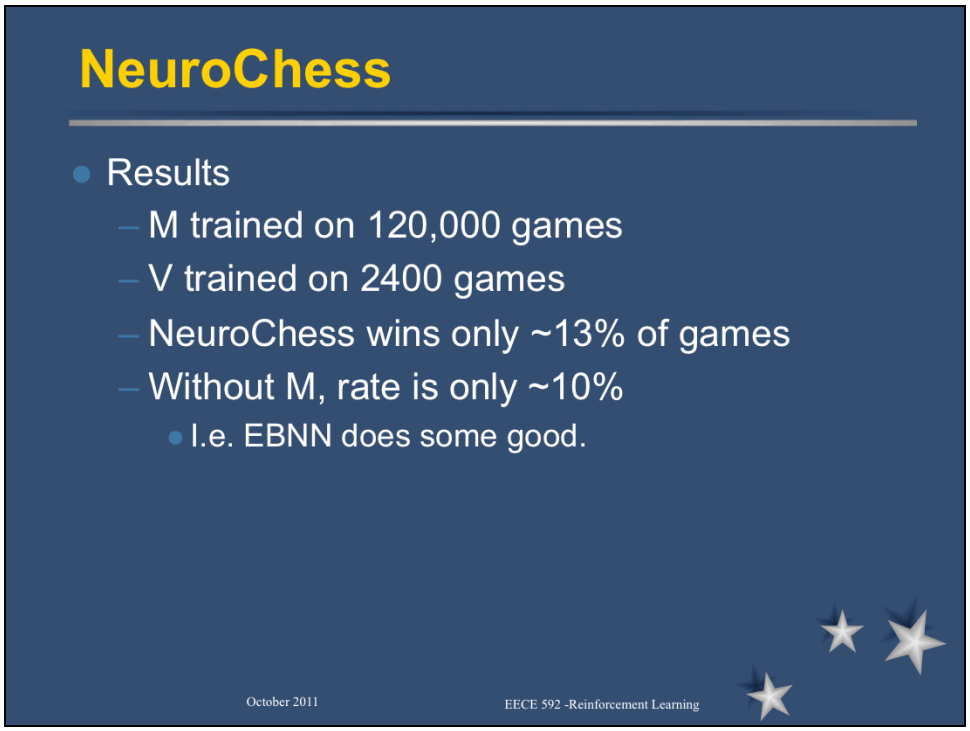
Shown is a illustration of the NeuroChess system.



The chess model *M* is a neural network "theory" of good chess play. It is trained before *V*, on a large sample of grandmaster games. The purpose of *M* is to map arbitrary board positions to the corresponding board position two half moves later. The intention is that *M* will capture important knowledge relevant to high quality chess play. This is a form of explanation based learning and *M* is an explanation based neural net (EBNN). Explanation based learning typically generalizes better from less training data, relying instead on the availability of domain data rather than vast amounts of statistical input.



The evaluation function *V* is trained on a mixture of grandmaster games and self-play. It learns from each position in each game using an algorithm that takes into account a target evaluation function value and a target slope of the evaluation function for the training vector of 175 features. The target values are computed by a temporal difference method. The target slopes are computed using the chess model *M* applied to the feature vector of the chess position. In effect, the chess model estimates the importance of each feature to the evaluation of the position, and the training of *V* takes that information into account.



After training *M* with 120,000 grandmaster games and training *V* with a further 2400 games, NeuroChess defeats gnuchess about 13% of the time. With the same training, a version of NeuroChess, which does not use the chess model *M*, wins about 10% of the time against gnuchess. EBNN is doing some good.

Learning from self-play alone failed. However, learning primarily from expert play introduced artifacts into the program's evaluations. The paper gives an example: the program observes that grandmasters are more likely to win when they move their queens to the center of the board. It doesn't realize that grandmasters only do such a thing when the queen is safe from harassment, and ends up moving its queen to the center at the earliest opportunity, safe or not. The best results came from a mixture of self-play and expert play. Training against other opponents was not tried (gnuchess was used only as a test opponent).

The previous diagram illustrates the structure of Neurochess. Boards are mapped into a high-dimensional feature vector, which forms the input for both the evaluation network *V*, and the chess model *M*. The evaluation network is trained by Backpropagation and the TD (0) learning procedure. Both networks are employed for analyzing training examples in order to derive target slopes for *V*.

NeuroChess

- Results

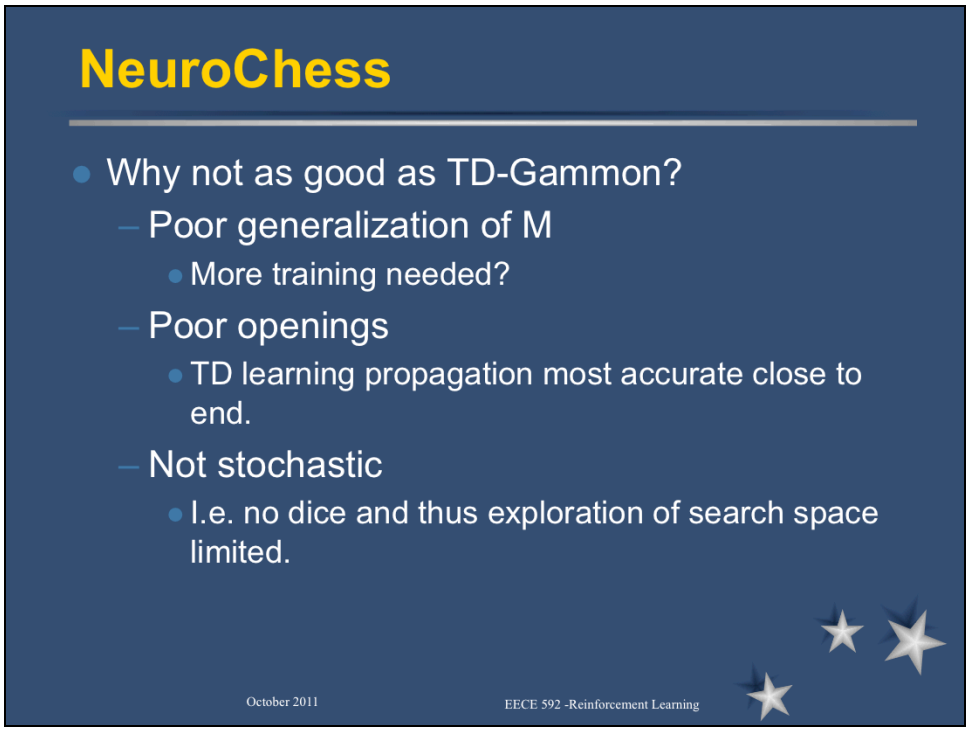
# of games	Backpropagation	EBNN
100	1	0
200	6	2
500	35	13
1000	73	85
1500	130	135
2000	190	215
2400	239	316

The results are summarised in the table shown.

Unlike TD-Gammon, Neurochess did not reach an expert level of play. In fact the level of play was still below that of gnuchess or of human play. The natural question that follow is why did TD learning not produce similar levels of success in each case?

Thrun himself believes that training times are one reason. Without excessive learning, it is difficult to reach a high level of skill. For example, Neurochess is seen to perform, at times, very well and has learned to trade and protect its king. However, at other times the moves it makes are absurd. This seems to suggest that the training has not generalized to all board position.

Neurochess also exhibits particularly poor openings. This is perhaps not surprising since TD propagates values from the end of a game to the beginning. The strength of the reinforcement signal is far stronger for later moves than for earlier ones.



The stochastic nature of backgammon may be a significant factor in its success, something that the game of chess lacks. During self-play, TD-Gammon is searched to explore a large area of the search space simply because of the dice. Training using self-play alone leads results in a very narrow level of experience. Without any randomness, the number of board positions explored is limited. To some degree online play and the use of sampled grand master games to enhance learning address this. The chess model M, which is an example of an *explanation-based-neural-net*, also attempted to address this problem.