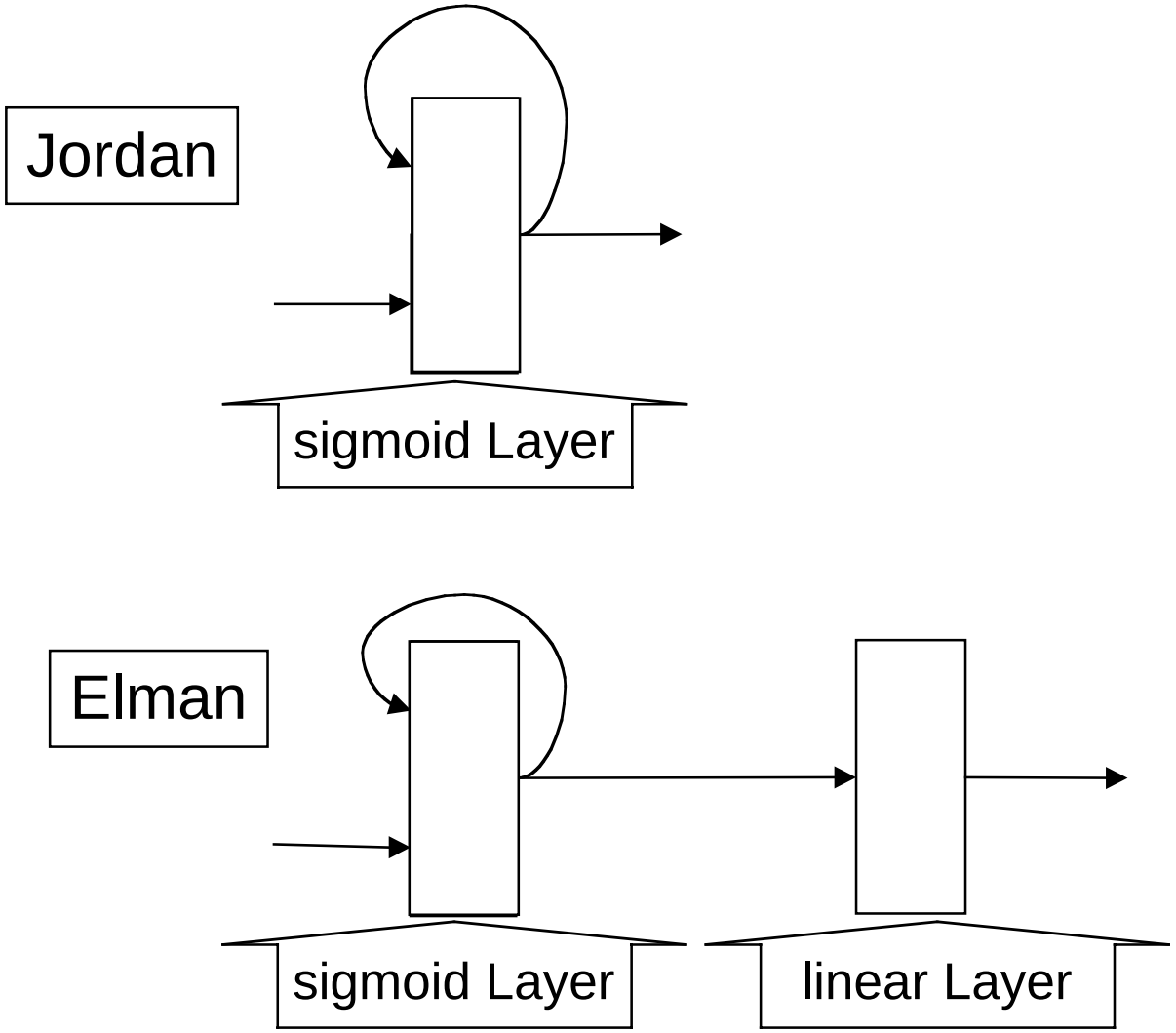
Truly Recurrent Networks

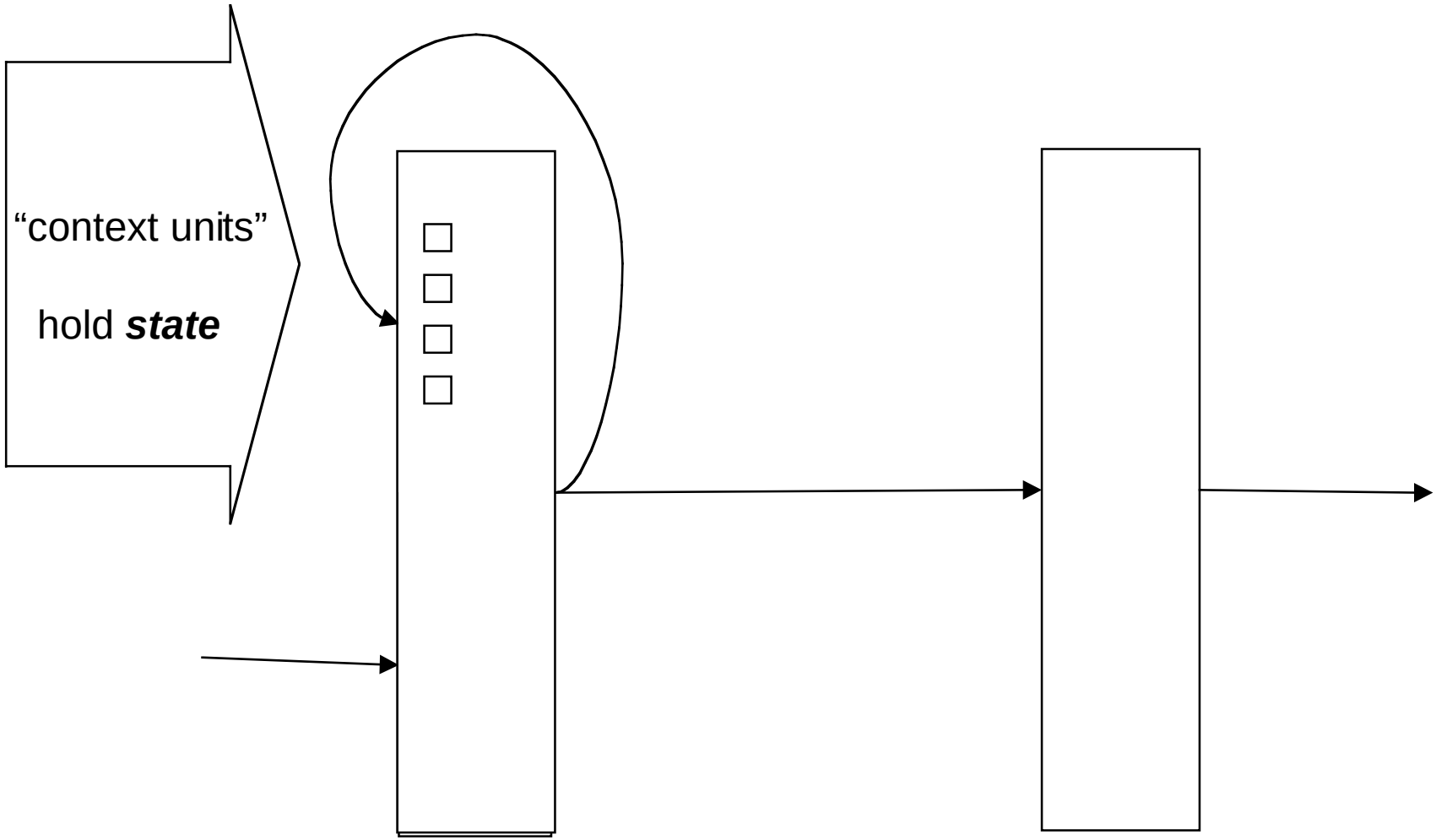
Truly Recurrent Network

- A truly recurrent network is one for which there is feedback from a neuron's output to its input.
- Various models exist:
  - Jordan Network (feedback from net output to input)
  - Elman Network (feedback from internal state output to input)
  - Hopfield Network (a special class, studied later)

Jordan vs. Eleman Networks



Eleman Networks



How to Train an Elman Network?

How to Train an Elman Network?

- One way:
  - Initialize the state values to *nominal*.
  - Repeat
    - Simulate one step of the network.
    - Compute the actual output.
    - Backpropagate the error.
    - Adjust the weights.
    - Compute the next state.
  - Until the error is sufficiently low.

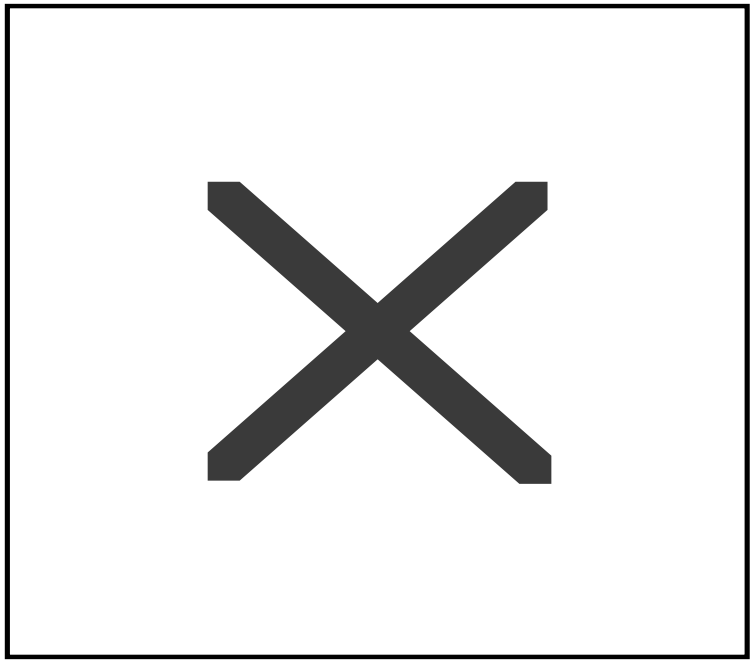
Could this Possibly Work?

- Two demos:
  - NAS (textbook) demo 11.2
  - Matlab appelm1

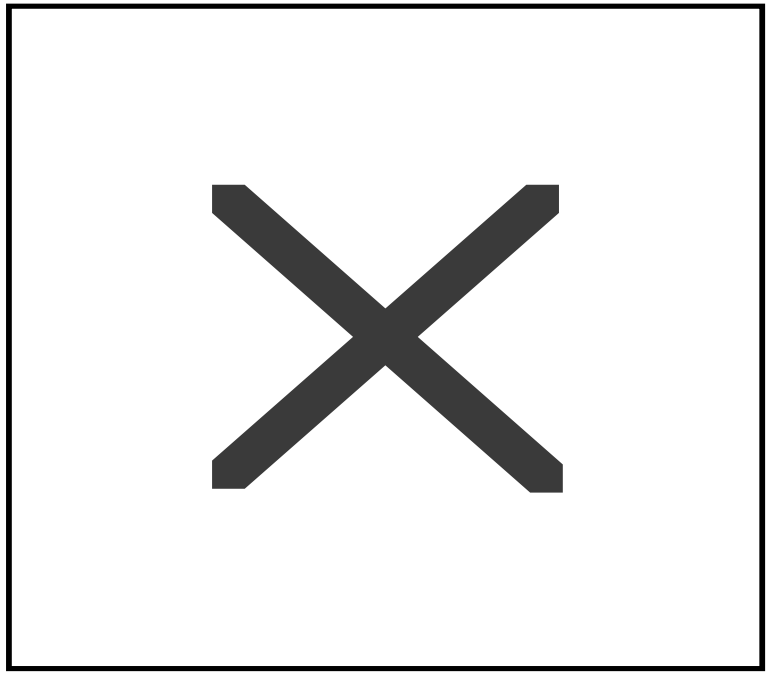
Elman Demo (matlab)

In this demo, an Elman network is trained to track the ***amplitude*** of a sine wave, by training on a signal of two different amplitudes.

Training

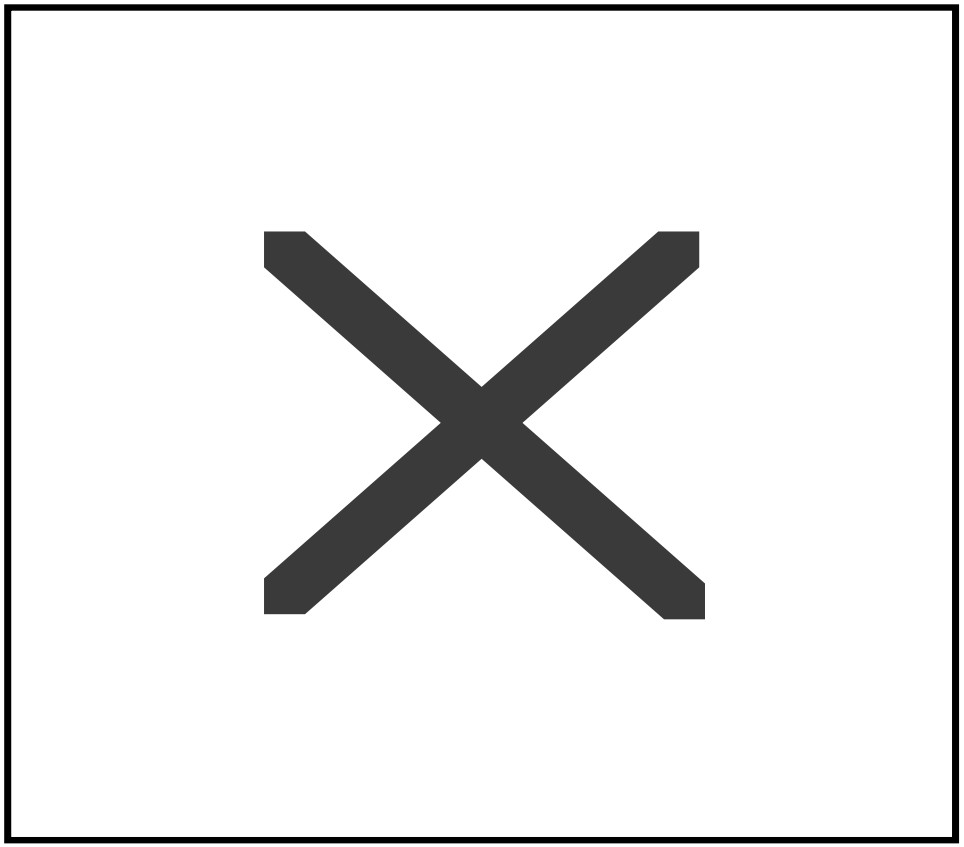


Validation



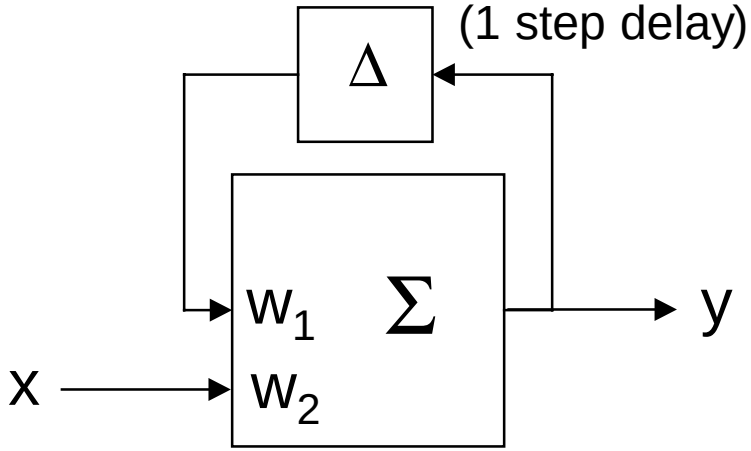
Elman Demo (matlab)

Training MSE plot



Training Feedback Weights

- Computing Sensitivities: See NAS section 11.3



$$y(k) = w_1\, y(k-1) + w_2\, x(k)$$

Training Feedback Weights

$y(k) = w_1\, y(k-1) + w_2\, x(k)$

$\partial/\partial w_2 \;\; y(k) = x(k)$

$\partial/\partial w_1 \;\; y(k)$

$= \partial/\partial w_1 \; w_1\, y(k-1)$

$= y(k-1) + w_1\, \partial/\partial w_1\, y(k-1)$ [derivative of a product]

$= y(k-1) + y(k-2) + w_1\, \partial/\partial w_1\, y(k-2)$ [a recurrence]

$= y(k-1) + y(k-2) + y(k-3) + \ldots$

Training Feedback Weights

$y(k) = w_1\, y(k-1) + w_2\, x(k)$

$\partial/\partial w_2 \;\; y(k) = x(k)$

$\partial/\partial w_1 \;\; y(k)$

$= \partial/\partial w_1 \; w_1\, y(k-1)$

$= y(k-1) + w_1\, \partial/\partial w_1\, y(k-1)$ [derivative of a product]

$= y(k-1) + y(k-2) + w_1\, \partial/\partial w_1\, y(k-2)$ [a recurrence]

$= y(k-1) + y(k-2) + y(k-3) + \ldots$

Conclusion: The sensitivities will depend on all previous values of the output.

Other Ways to Train an Elman Network

- BPPT (Backpropagation Through Time, seen last time) would be another way: unroll the network some large number of levels, backpropagate, average the weight changes over the unrolled stages to get a single set of weight changes.

Training Elman using BPPT

- See NAS demo 11.3.

Real-Time Recurrent Learning (RTRL)

- another approach to training recurrent networks, due to Williams and Zipser
- “dual” of BPPT? (NAS)
- Compute partial derivatives using Eq. 11.19, where $s^m_{ij} = \partial y_m / \partial w_{ij}$, with y_m being the m^{th} output variable

$$s^m_{ij} = f'(net)[\Delta_{im}\, y_j + \Sigma_k\, w_{mk}\, s^i_{ij}]$$

where Δ_{im} is the Kronecker delta ($\Delta_{im} = 1$ if $i = m$, 0 otherwise)

Game Applications

Game Applications

- BPPT seems potentially useable.
- Another approach is to use the "Temporal Difference" method.

Learning Types

- **Supervised learning**: Training with desired answer given for each action
- **Unsupervised learning**: No desired answer; learns *similarities* (clustering)
- **Reinforcement learning**: Reward given later, not necessarily tied to specific action

Example: Game Playing

- How to learn to play a game, say tic-tac-toe?

- Supervised learning approach:

  Listen to a teacher indicate good moves in various situations, or

  observe an expert player play games; learn to mimic the good player.

Problems with Supervised

- Need access to expert or many recorded game samples.
- There is generally no play-by-play target value; the value is only assigned to a sequence of plays, ending with +1 (win) or -1 (lose).

Reinforcement Learning

- Tries to address difficulties with supervised learning.
- Reward can be deferred until the end of the game.
- Can deal with stochastic environment (transition probabilities).

Reinforcement Learning
Temporal Differences
TD(λ)

Typical AI Model: MDP (Markov Decision Problem)

- Set of states
- Set of actions
- Transition probabilities between states:
  M^{a}_{ij} =

  prob[transition from state i to state j
  given action *a* is taken in i]

- Reward R(i) (positive, negative, or 0) associated with each state i

Utility Functions

- Desirability of moving to a given state s is expressed by a ***utility*** U(i).
- The utility is generally not given; it must be *learned*.
- Normally the ***expected*** utility is sought, since transitions can be probabilistic.
- Generally, given a choice of moves to several states, the state with ***highest expected utility*** will be chosen.

Additive Utility Function

- Assume that the sought utility function is *additive*:

$$U(i) = R(i) + \max_{a}\left(\sum_{j} M^{a}_{ij} \bullet U(j)\right)$$

Utility of state i

where R(i) is the reward of state i, a is an action, and M^{a}_{ij} is the probability of going from state i to state j with action a.

- This is the *dynamic programming* equation (Richard Bellman).

A way to compute U

- Initialize U to R, the reward function
- While(not converged)

  {

  foreach state i, compute

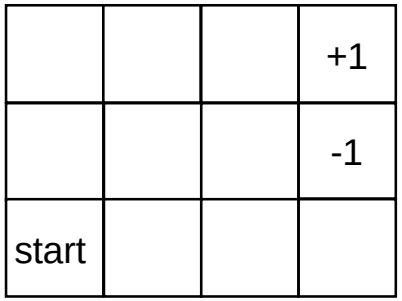
  $$U'(i) = R(i) + \max_{a}\left(\sum_{j} M^{a}_{ij} \bullet U(j)\right)$$

  replace U with U'

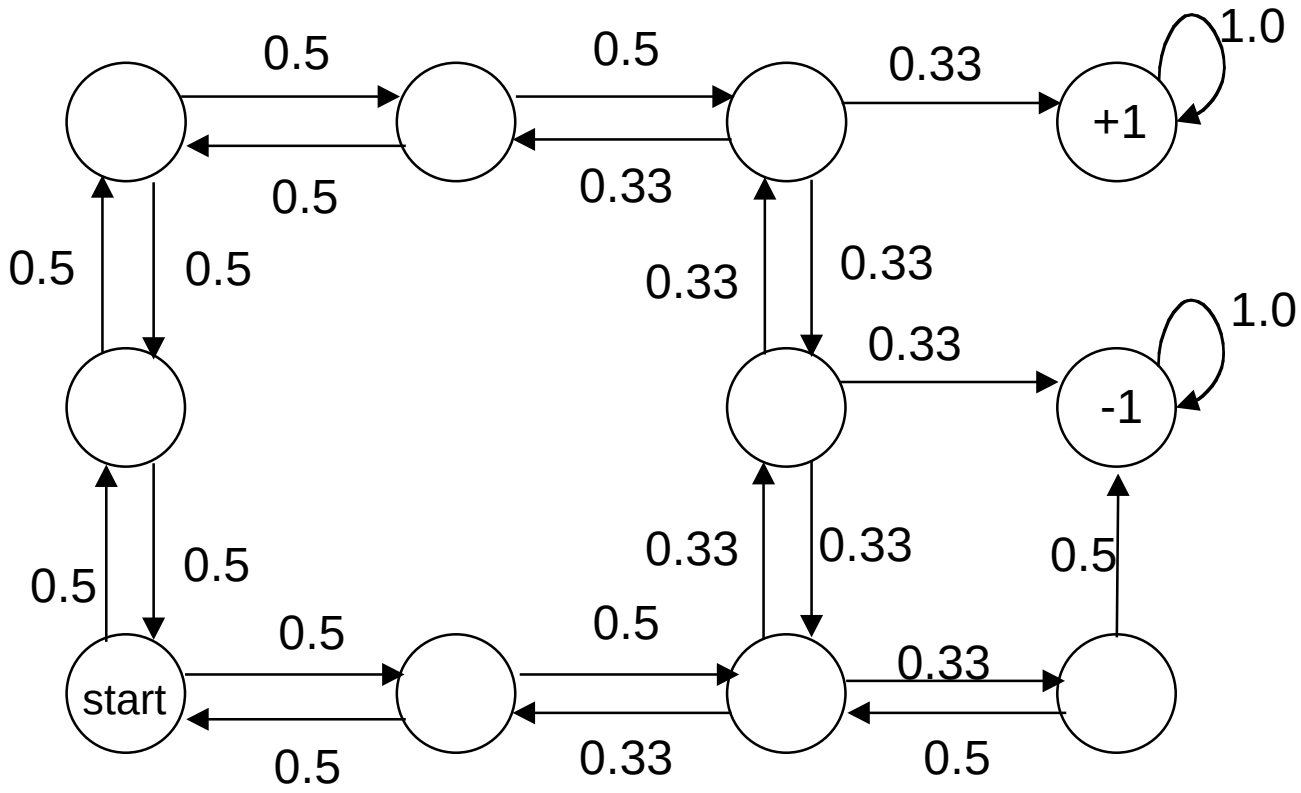
  }

Example (Norvig & Russell)

- Consider the following maze, with reward function 0 except as shown in the two boxes (with +1, -1).
- Assume a single action “move” with equal probabilities of moving any direction.



State-Transition Probs



Computed Utilities using Dynamic Programming Eqn

-0.03	0.09	0.22	1.0
-0.15		-0.43	-1.0
-0.28	-0.40	-0.53	-0.76

Problem with this Method

- There are generally too many states in a game to:
  - enumerate
  - compute their utilities

Temporal Difference Learning

- Through many trial runs, adjust the observed values of U so that they more closely agree with the dynamic programming equation.
- If there is a transition from i to j, adjust U(i) so that it better agrees with U(j).
- Updating rule (with η the learning rate):

$$\Delta U(i) = \eta \bullet (R(i) + U(j) - U(i))$$

Utilities Computed by TD vs. by Dynamic Programming

This was after 1 million cycles. The values were not yet stable.

TD →	-0.01	0.05	0.14	
DP →	-0.03	0.09	0.22	1.0
	-0.07		-0.5	-1.0
	-0.15		-0.43	
	-0.11	-0.16	-0.46	-0.76
	-0.28	-0.40	-0.53	-0.76

More on TD method

- TD tries to train a function to **predict** the utility of states.
- The earliest known use (not by the name TD) was in Samuel's checker-playing program (1959).
- Richard Sutton expressed the general framework (1988).

Single- vs. Multi-Step Prediction

- In single-step prediction, the outcome is revealed right after the prediction.
- In multi-step prediction, the outcome is delayed for several or many steps.
- TD is applied in multi-step cases (in single-step it is identical to supervised learning).

Sutton's Formulation (1)

- Observation-outcome sequence:

  $x_1, x_2, x_3, \ldots, x_m, z$

- x_t is the observation at step t
- z is the outcome of the sequence
- All values are real numbers
- Learner produces **predictions** that estimate z:

  $P_1, P_2, P_3, \ldots, P_m$

Sutton's Formulation (2)

- In general, a **prediction** P_t can be a function of all preceding observations, but for simplicity it can be assumed to just depend on x_t.
- (We could always include all previous observations as "part of" the current observation.)
- P_t can be regarded as the **utility** value in the previous slides.

Sutton's Formulation (3)

- If computed by a neural net, P_t will also depend on some weights w, and could be written explicitly as

$$P_t = P(x_t, w)$$

- The learning rule will indicate how to update w.
- Let Δw_t be the weight change as a result of prediction P_t .

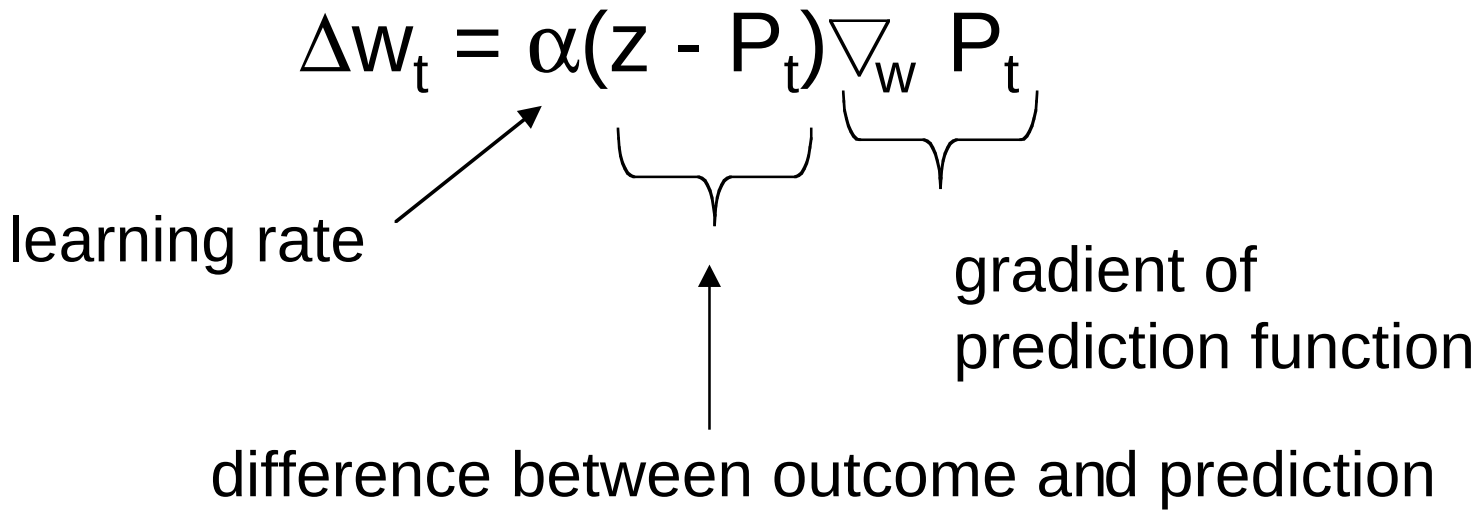
Sutton's Formulation (4)

- The net change in w over the entire observation sequence $x_1, x_2, x_3, \ldots, x_m$, is thus:

$$\Sigma_t \Delta w_t$$

Sutton's Formulation (5)

- Supervised learning would pair each observation with the final outcome and train thus:



Sutton's Formulation (6)

- Example: If the prediction function were linear: $P_t(w, x_t) = \Sigma\, w(i) \bullet x_t(i)$ then

  $\nabla_w P_t = x_t$

  and we have the Widrow-Hoff rule:

$$\Delta w_t = \alpha(z - w^T x_t) \bullet x_t$$

$(z - w^T x_t)$: difference between outcome and prediction

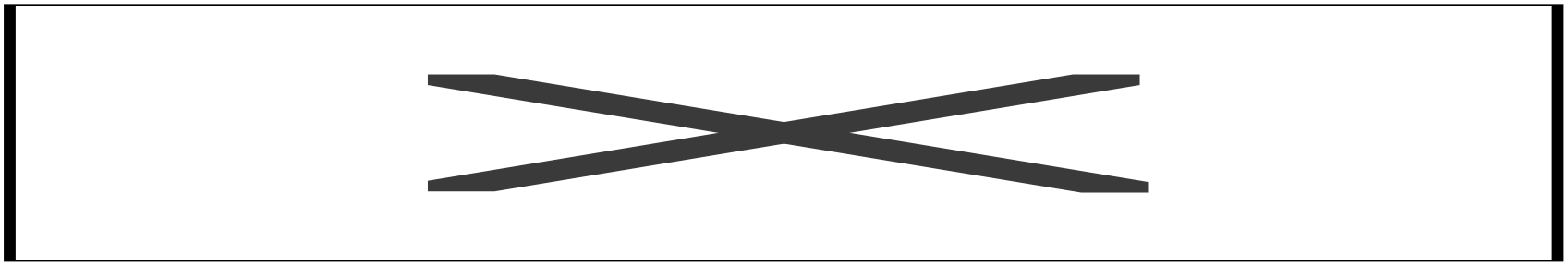
x_t: gradient of prediction function

Sutton's Formulation (7)

- For backpropagation network, rather than linear, the same update form can be used. The gradient is just more complicated.
- The problem with supervised technique is that it requires knowledge of the final outcome.
- Temporal differences remove this requirement.

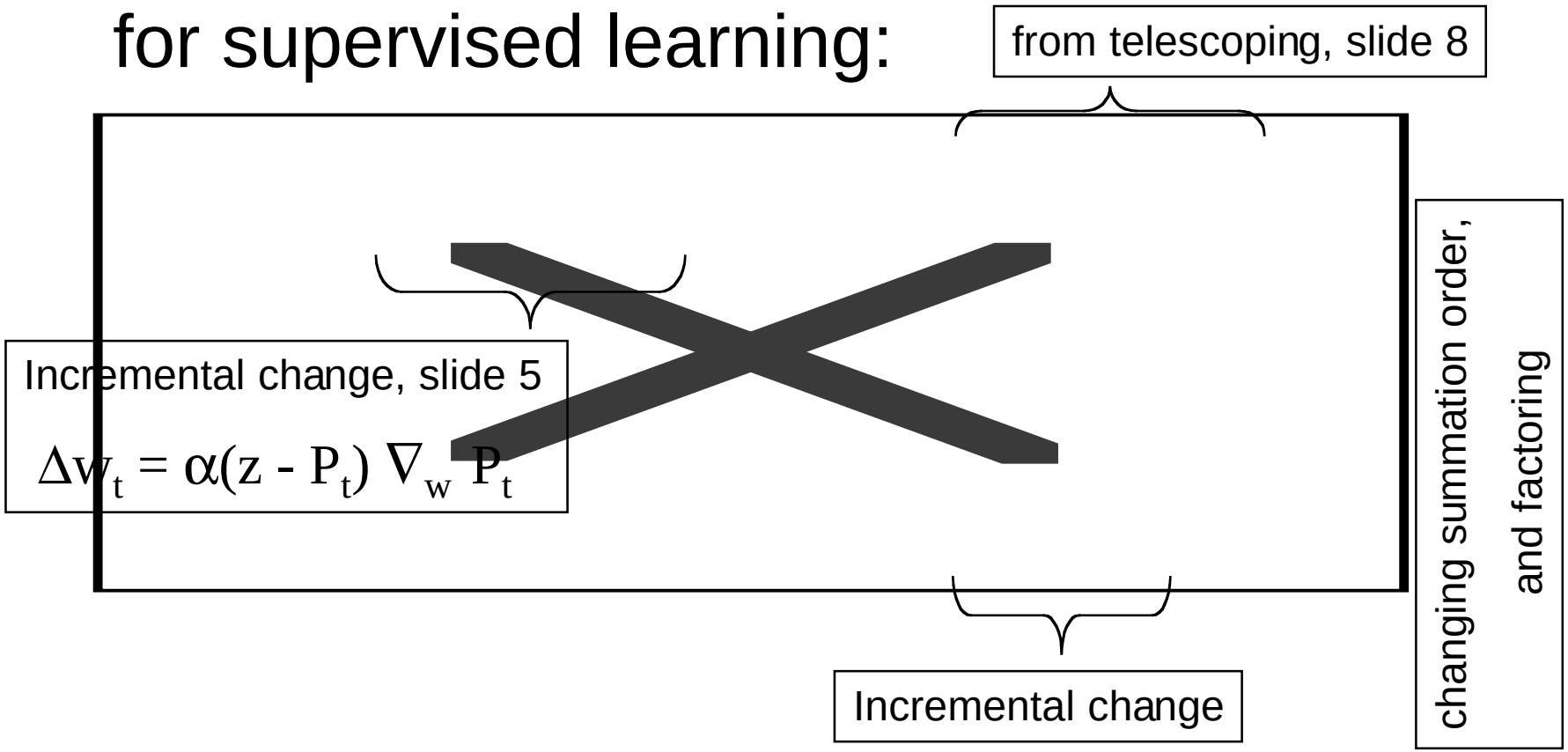
Sutton's Formulation (8)

- Represent the error in a prediction $z - P_t$ as a sum of **changes** in predictions, using "telescoping"



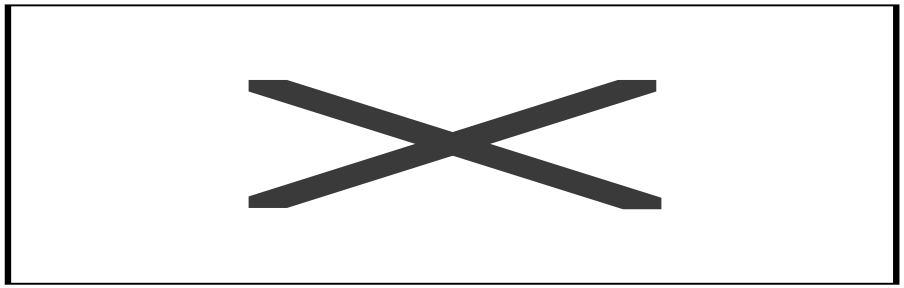
Sutton's Formulation (9)

- Now re-express the net weight-change for supervised learning:



Sutton's Formulation (10)

- From slide 9, the incremental weight change can be seen as



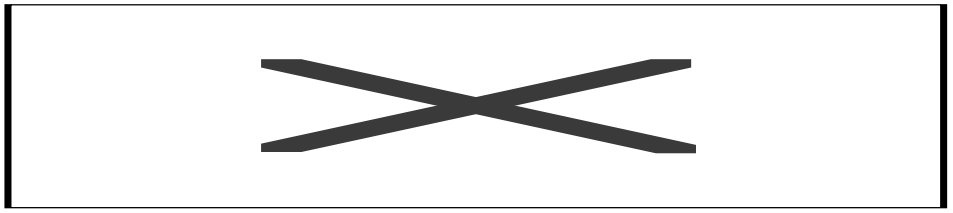
- In other words, weight change is based on the **difference** between current and previous prediction, and gradients computable at previous steps.

Sutton's Formulation (11)

- If using backprop, for example, one would need to maintain a sum of the gradient values (weight changes) from previous steps.
- The method on the previous slides is called TD(1).
- For the *linear* case, TD(1) gives the same weight changes as Widrow-Hoff would.

Sutton's Formulation (12)

- TD(1) can be **generalized** to TD(λ), where λ is any value between 0 and 1.
- The value of λ is a decay factor indicating what portion of previous weight changes are to be added in.



- Lower values of λ give more weight to recent predictions.

Sutton's Formulation (13)

- Of special interest is TD(0) (note $0^0 = 1$):

$$\Delta w_t = \alpha(P_{t+1} - P_t)\nabla_w P_t$$

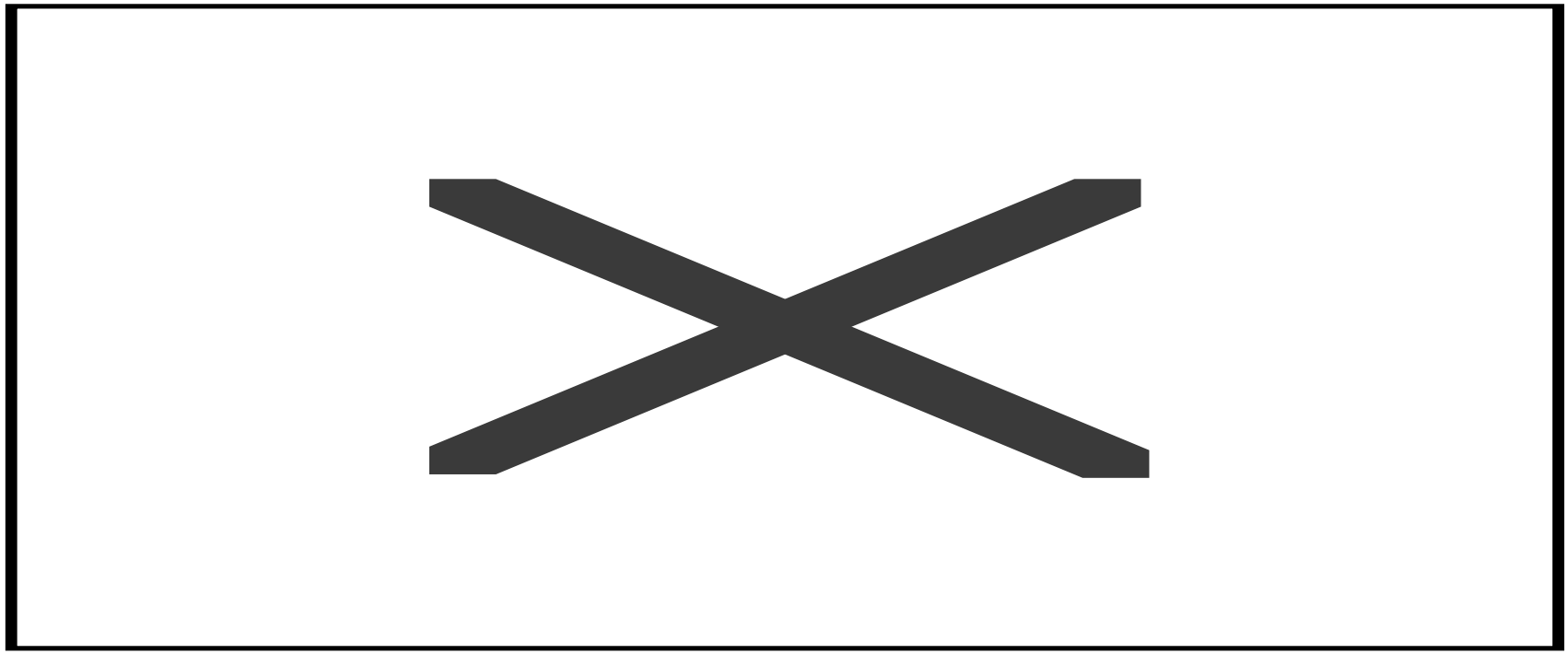
Change in value of prediction function

Gradient of P_t

Possible Use of TD in Game-Playing

- For a given state of the game, enumerate the possible moves.
- Evaluate P_t (prediction of a win) for each resulting state.
- Choose the move for which P_t is highest.

Case Study: Backgammon (Gerald Tesauro, 1995)



Backgammon Board

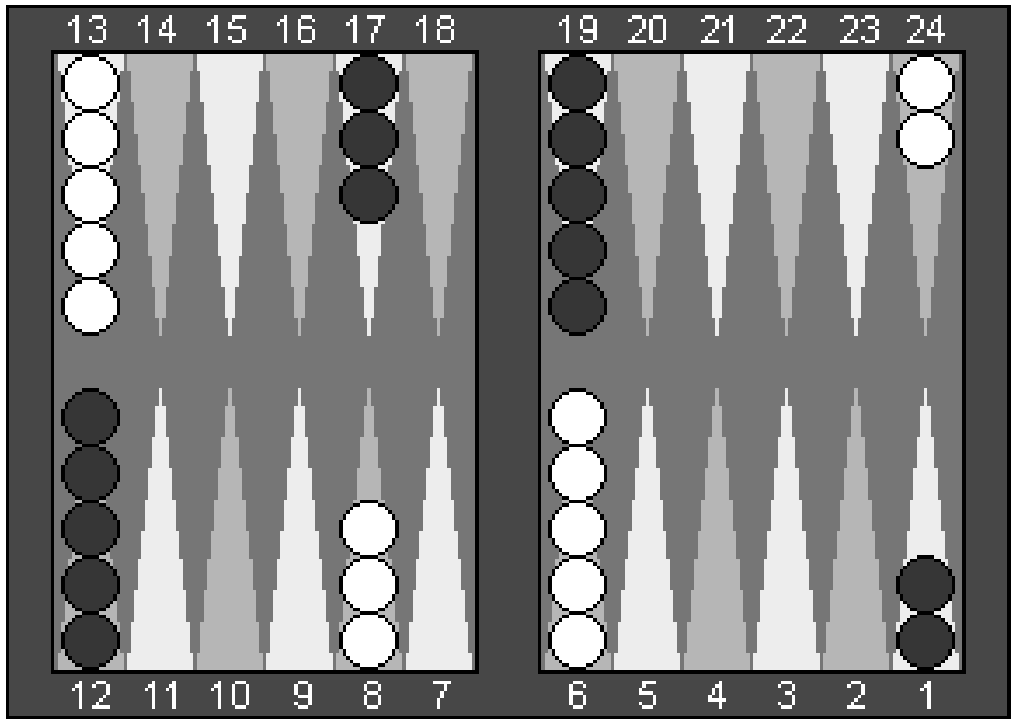


Figure 2. An illustration of the normal opening position in backgammon. TD-Gammon has sparked a near-universal conversion in the way experts play certain opening rolls. For example, with an opening roll of 4-1, most players have now switched from the traditional move of 13-9, 6-5, to TD-Gammon's preference, 13-9, 24-23. TD-Gammon's analysis is given in Table 2.

Summary of Backgammon

- Players roll dice and move their checkers from points according to the numbers shown on the dice.
- The sum of the number of points moved equals the number showing on the dice.
- Landing on another players checker captures it.

- (I’m hoping a player in class will fill in.)

Neurogammon

- Earlier program by the same author, 1989
- Trained using supervised learning (not TD):
  - 30,000 “expert opinions”
- Eventually augmented neural network with a traditional 2-ply AI search.

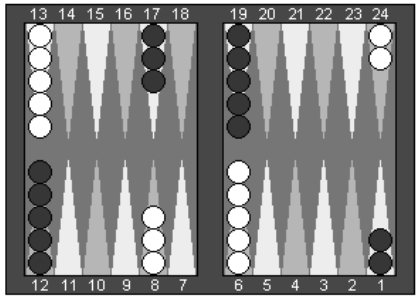
TD-gammon

- 2-layer network with:
  - 1 output (whether a proposed state is good or not)
  - 198 inputs
  - 40 or 80 hidden neurons
- Weight-update rule:

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

Board Encoding Important (1)

- 4 inputs encode the number of white pieces on each of 24 board points:
  - 0000: no pieces
  - 0001: one piece
  - 0011: two pieces
  - 0111: three pieces
  - x111: >3 pieces, x = (n-3)/2 for n pieces



- 4x24 = 96 inputs for white + 96 for red

Board Encoding Important (2)

- Two more inputs encode number of pieces on the bar (n/2) for n pieces.
- Two more inputs encode the number of pieces removed (n/15).
- Two units encode whose turn to move.
- All unit inputs were roughly in the 0 to 1 range.

TD-gammon results world-class play

Program	Training Games	Opponents	Results
TDG 1.0	300,000	Robertie, Davis, Magriel	-13 pts/51 games (-0.25 ppg)
TDG 2.0	800,000	Goulding, Woolsey, Snellings, Russell, Sylvester	-7 pts/38 games (-0.18 ppg)
TDG 2.1	1,500,000	Robertie	-1 pt/40 games (-0.02 ppg)

Results of testing TD-gammon in play against world-class human opponents. Version 1.0 used 1-ply search for move selection; versions 2.0 and 2.1 used 2-ply search. Version 2.0 had 40 hidden units; versions 1.0 and 2.1 had 80 hidden units.

TD-gammon results

- In 1994, TD-Gammon was at the level of the best human players in the world.
- Expert players learned new strategy from *it*.

Judgement superior to experts?

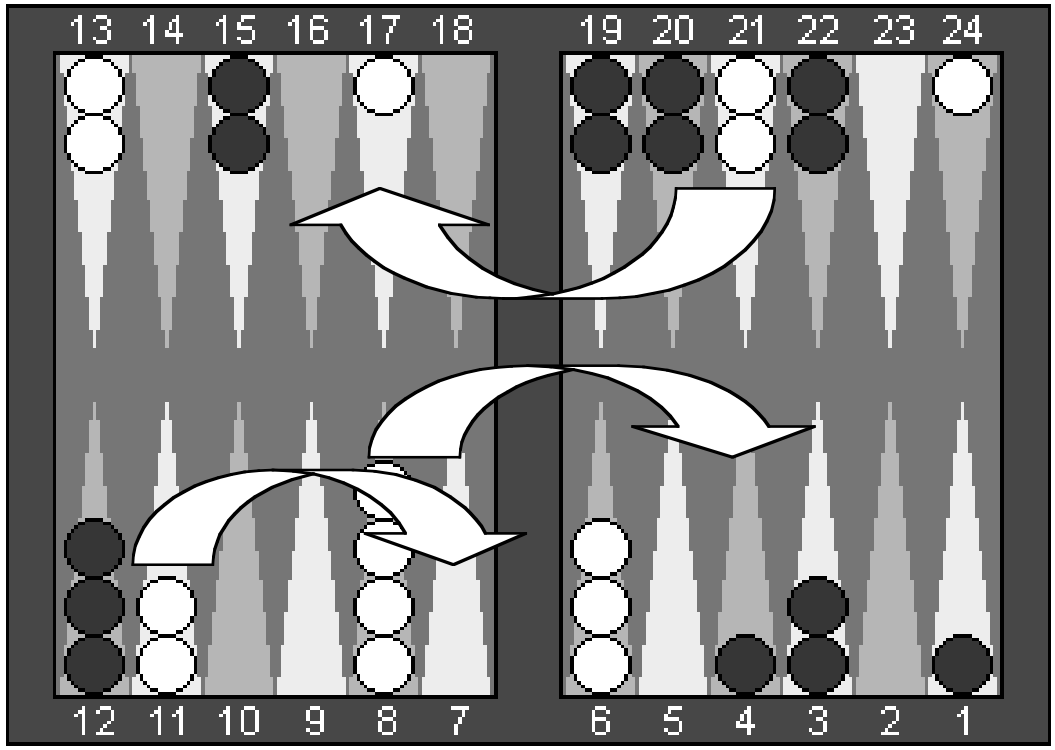


Figure 3. A complex situation where TD-Gammon's positional judgment is apparently superior to traditional expert thinking. White is to play 4-4. The obvious human play is 8-4*, 8-4, 11-7, 11-7. (The asterisk denotes that an opponent checker has been hit.) However, TD-Gammon's choice is the surprising 8-4*, 8-4, 21-17, 21-17! TD-Gammon's analysis of the two plays is given in Table 3.

Other TD uses

- Checkers (A. Samuel)
- Go
- Othello
- Chess
- AHC (Adaptive Heuristic Critic): pole-balancing, etc. (Barto, Sutton, and Anderson)