Applications and Case Studies

Sunday, June 28, 2020 8:14 PM

TD-Gammon:

- This algorithm is a combination of a nonlinear form of TD(λ) and a neural net backpropagating TD errors
- The estimated state value $\hat{v}(s,\vec{w})$ was meant to estimate the probability of winning starting from state s
- Rewards were defined as 0 for all time steps except those on which the game was won
- The value function was represented as a neural net with inputs representing the backgammon position, one hidden layer with 40-80 units and an output layer with 3 neurons, the first representing the probability of winning $\left(\hat{v}(s,\vec{w})\right)$ in the normal manner, the second representing the probability of winning by a gammon, and the third representing the probability of winning by a backgammon
- The first version of the input layer had 198 neurons with each point on the board being represented by 4 neurons for each color, the first neuron corresponding to 1 piece of the given color being present at that point, the second neuron corresponding to 2 or more pieces of the given color being present at that point (this is important since having 2 or more pieces of a given color at a certain point blocks the opponent from moving onto that point), the third neuron corresponding to exactly 3 pieces of the given color being present at that point, and the fourth neuron set to a value proportionate the number of additional pieces beyond 3 of a given color being present at that point according to $\frac{n-3}{2}$
  - The above mentioned strategy corresponds to 192 of the 198 inputs, with 2 more representing the number of pieces of each color on the bar (out of play and have to reenter the game from the starting position) by the value $\frac{n}{2}$ where n is the number of pieces of that color on the bar, 2 more representing the number of pieces of each color removed from the board by the value $\frac{n}{15}$ where n is the number of pieces of that color removed from the board, and the final 2 representing if it was black's or white's turn to move
    - The important theme to note is that one unit was provided for every conceptually distinct possibility that seemed relevant and scaled the inputs to roughly the same range (between 0 and 1)
- The neural net used a sigmoid activation function, which makes sense since only one hidden layer is present and the output represents a probability
- Semi-Gradient TD(λ) Algorithm Used:
- $\vec{w}_{t+1} \equiv \vec{w}_t + \alpha\left[R_{t+1} + \gamma\hat{v}(S_{t+1},\vec{w}_t) - \hat{v}(S_t,\vec{w}_t)\right]\vec{z}_t$
- $\vec{z}_t \equiv \gamma\lambda\vec{z}_{t-1} + \nabla\hat{v}(S_t,\vec{w}_t)$
  - Where $\vec{z}_0 = 0$
  - Note that γ in this case is 1 (this task is undiscounted)
- The algorithm was trained through self-play
- The weights were updated after every move
- Later versions of TD-Gammon used two-ply rollout search in planning move choices (evaluating moves through their consequences up to 2 moves in the future assuming the opponent makes the best possible moves for various dice rolls)

Samuel's Checkers Player:

- Samuel wrote his first checkers-playing program in 1952 for the IBM 701, with the first learning version completed in 1955
- Worked by performing a heuristic lookahead search
- Used a state-value function using linear function approximation
- Action selection was based on Claude Shannon's minimax procedure

Working backward through a search tree starting at terminal positions each position was given the score of the position that would result in the best move, assuming that the machine always tries to maximize its score and the enemy always tries to minimize it

He also introduced discounting

Used two learning methods:

Rote Learning:

Saving a description of each state encountered during play and the value determined by the procedure given above

Learning by Generalization:

Updating the value of each state after each move of the game by the machine, the update was towards the minimax value of a search launched from the position attained after the move being updated

This has the effect of the update being a backing-up over one event of real play then a search over possible events

No explicit rewards were present, but the program tried to maximize its piece advantage (number of pieces it has relative to its opponents, with kings weighted more than normal pieces)

This program included nothing tying state values to the true values of states (usually rewards and discounting or giving a fixed value to terminals states has this effect)

This means that the program could get worse with experience

Watson's Daily Double Wagering:

Tesauro (the guy behind TD-Gammon, helped create Watson's daily double strategy)

Watson selected wager amounts by comparing action values that estimated the probability of a win from the current game state for each round-dollar legal bet

Action values were computed using two types of estimates, the first estimating the value of afterstates that would result from any legal bet and the second giving an in-category confidence representing the likelihood that Watson would answer the question correctly

The reinforcement learning approach is the same as TD-Gammon (nonlinear TD(λ) with a neural net)

Features in the feature vector included the current scores of players, how many daily doubles were left, and other information about how much play was left in the game

Watson's state values were learned over millions of simulated games against crafted models of humans

Watson's confidence estimates were conditioned on the number of correct responses Watson gave in previously-played clues in the current category

Betting Action Values:

$\hat{q}(s, bet) = p_{DD} \times \hat{v}(S_W + bet, \cdots) + (1 - p_{DD}) \times \hat{v}(S_W - bet, \cdots)$

Where p_{DD} is the in-category confidence, S_W is Watson's current score and $\hat{v}$ gives the estimated value for the game state after Watson's response to the clue

The authors didn't like the amount of risk this strategy took on so they adjusted this by subtracting a small fraction of the standard deviation over Watson's correct/incorrect afterstate evaluations and prohibiting bets that would cause the wrong-answer afterstate value to decrease below a certain limit

Watson's win rate using a baseline heuristic daily double strategy was 61%, with learned values and a default confidence values it was 64% and with live in-category confidence it was 67%

Optimizing Memory Control:

This was done by Ipek et al (2008) and Martinez and Ipek (2009)

DRAM memory control can be formulated as a reinforcement learning task where the states are the contents of the transactions queue, the actions are commands to the DRAM system (precharge, activate, read, write, or NoOp) and the reward is 1 whenever the action is read or write and 0 otherwise

State transitions are here considered stochastic since they depend on things outside of the controller's control such as the workloads of the processor cores using the DRAM system

This setup is heavily dependent on making on certain actions available in certain states to ensure the integrity of the system

DRAM Actions:

- Precharge:
  - Transfers the data in the row buffer into the addressed row of the cell array
- Activate:
  - "Opens a row" which moves the contents of the row (address indicated by command) into the row buffer
- Read:
  - Transfers a word (sequence of bits) in the row buffer into the external data bus
- Write:
  - Transfers a word (sequence of bits) in the external data bus into the row buffer
- NoOp:
  - Does nothing

This agent used Sarsa to learn action values with ε-greedy exploration and tile coding

States were represented as 6 integer valued features with features including the number of read requests in the queue, number of write requests in the queue, the number of write requests in the queue waiting for their row to be opened, and the number of read requests in the queue waiting for their row to be opened

Online learning increased DRAM throughput by 19% on average, including an 8% advantage over a previously trained fixed policy

Deep-Q-Network:

- DQN combined Q-learning with a deep convolutional network
- DQN used the semi-gradient form of Q-learning
- DQN has 3 convolutional layers (including subsampling), 1 fully connected layer, and the output layer
  - The features maps of the convolutional layers are in order, 32 20 x 20 maps, 64 9 x 9 maps, and 64 7 x 7 maps
  - The fully connected layer has 512 units
  - The output layer has 18 units corresponding to the maximum number of possible actions in an Atari game
- DQN uses ReLU for its activation function
- The activation of the output layer represents estimated optimal action values for the state represented by the input vector
- The reward signal was +1 whenever the agent's score in the game increased, -1 whenever the agent's score in the game decreased and 0 otherwise
- DQN used and ε-greedy policy where ε decreased linearly over the first million frames then remained at a low value for the rest of the learning session
- The input to DQN was an 4 x 84 x 84 vector composed of the 4 more recent frames sampled down to 84 x 84 (the frames were originally 210 x 160) with 128 colors at 60 Hz
- General Semi-Gradient Q-Learning (Not exactly what DQN used, see below):

$$\vec{w}_{t+1} = \vec{w}_t + \alpha \left[R_{t+1} + \gamma \max_{a} \hat{q}(S_{t+1}, a, \vec{w}_t) - \hat{q}(S_t, A_t, \vec{w}_t) \right] \nabla \hat{q}(S_t, A_t, \vec{w}_t)$$

  - Where the gradient was computed through mini-batch backpropagation where a batch was 32 images
  - They also used RMSProp, which adjusts the step-size parameter for each weight based on a running average of the magnitudes of recent gradients for that weight
- DQN Modified Q-Learning in 3 Ways:
  - Experience Replay:
    - Storing tuples of $(S_t, A_t, R_{t+1}, S_{t+1})$ encountered during gameplay
    - At each time step a mini-batch update was performed using samples sampled uniformly from this replay memory
      - This works since Q-Learning is an off-policy algorithm so it doesn't need to be applied along connected trajectories

This helped reduce variance since successive updates weren't correlated

Duplicate Network:

Update Rule:

$$\vec{w}_{t+1} = \vec{w}_t + \alpha \left[R_{t+1} + \gamma \max_{\mathrm{a}} \tilde{q}\left(S_{t+1}, a, \vec{w}_t\right) - \hat{q}\left(S_t, A_t, \vec{w}_t\right) \right] \nabla \hat{q}\left(S_t, A_t, \vec{w}_t\right)$$

Where $\tilde{q}$ is a duplicate network set to equal the main network after every C updates

This deals with the problem of the target for the Q-learning update depending on the same parameters that are being updated, which can lead to oscillations or divergence

Error Clipping:

The term $R_{t+1} + \gamma \max_{\mathrm{a}} \tilde{q}\left(S_{t+1}, a, \vec{w}_t\right) - \hat{q}\left(S_t, A_t, \vec{w}_t\right)$ in the update rule was clipped to remain in the interval $[-1,1]$

AlphaGo/AlphaGo Zero:

AlphaGo used supervised learning from a large database of expert human moves while AlphaGo Zero used only reinforcement learning (which corresponds to the Zero in its name)

Go is a difficult problem since defining a position evaluation function is hard

AlphaGo selected moves from a modified version of Monte Carlo Tree Search (MCTS) they called asynchronous policy and value MCTS (APV-MCTS)

MCTS expands the search tree by using stored action values to select an unexplored edge but APV-MCTS expands its search tree by choosing an edge according to probabilities supplied by a 13-layer CNN called the SL policy network that was trained on a database of 30 million expert human moves

During execution APV-MCTS keeps track of how many simulations passed through each edge of the search tree and takes the action corresponding to the most-visited edge from the root node when search execution completed

APV-MCTS also using a previously learned value function (another 13-layer CNN) in addition to normal MCTS rollout

To train the value network they used a reinforcement learning network called the RL policy network initialized to the SL policy network mentioned above (both networks had the same 13-layer CNN structure) then trained further using MC methods and self-play

The output of the value network was a soft-max layer corresponding to the 19 x 19 Go board

The input to the network was a 19 x 19 x 48 vector consisting of 48 features applied to each point on the board

Features included whether a stone was present at a location and whose stone it was, the number of adjacent points that were empty, the number of opponent stones that would be captured by placing a stone at that location, the number of turns since a stone as placed in that location, and a bunch of other stuff

APV-MCTS State Values:

$$v(s) = (1 - \eta) v_\theta(s) + \eta G$$

Where G is the return of the rollout, v_θ is the previously learned value function, and η is a parameter that controls the relative values of each

η was set to 0.5 in the final version

AlphaGo trained by playing against a randomly selected previous policy to prevent overfitting

The reward was 1 if AlphaGo won, -1 if it lost, and 0 otherwise

AlphaGo Zero used only one CNN, had only the positions of pieces as input and used a simpler version of MCTS

AlphaGo Zero's MCTS ended at a lead node of the current search tree instead of at the terminal position of a complete game simulation

AlphaGo Zero output a probability of winning and probabilities of every possible move, including resign

AlphaGo Zero used its output to direct each iteration of MCTS
The input to AlphaGo Zero was a 19 x 19 x 17 vector with 17 binary feature maps over the board with 8 feature maps for the last 8 board configurations of the current player and another 8 feature maps doing the same thing for the other player, then 1 feature maps that was a constant 1 if black was to move and 0 if white was to move
AlphaGo Zero was two-headed, which means that it split into two separate "heads" of additional layers before outputting its final move probabilities and winning probability
AlphaGo Zero consisted of 41 convolutional layers with batch normalization and skip connections (for residual learning) before the heads split, the move probabilities were computed by 43 layers total and the winning probability was computed by 44 layers total
The network was trained by SGD with momentum, regularization, and a decreasing step-size parameter using batches of examples sampled uniformly at random from the most recent 500,000 games of self-play with the current best policy

> After every 1,000 training steps the current policy was evaluated by simulating 400 complete games against the current best policy, with the new policy replacing the current best policy if it won by a set margin

During training, noise was added to the output move probabilities to encourage exploration
AlphaGo Zero's MCTS runs for 1,600 iterations