Watson’s Daily Double Wagering

A. Introduction to Watson

Watson is a natural language question-answering system developed by IBM to compete on the TV quiz show *Jeopardy!*.

Watson achieved fame in 2011 by defeating two human champions, showcasing advanced capabilities in natural language processing, decision-making, and strategy.

While Watson’s primary strength was its ability to answer natural language questions accurately and quickly, its *Jeopardy!* success also relied on sophisticated strategies, including **Daily-Double (DD) wagering**, derived from reinforcement learning.

B. Overview of Jeopardy!

Game Mechanics:

- Played by three contestants across three rounds, each featuring a total of 60 clues.
- Clues are arranged on a board with six categories, each containing five clues of increasing monetary value.
- Contestants buzz in to answer:
    - Correct answers increase the contestant’s score by the clue’s dollar value.
    - Incorrect answers reduce the score by the same amount.

Daily Double (DD):

- One or two randomly selected clues per game are designated as DDs.
- The contestant selecting a DD clue has exclusive rights to answer but must wager an amount before seeing the clue.
- Correct answers increase the score by the wagered amount; incorrect answers decrease it by the same amount.

Final Jeopardy (FJ):

- Each contestant secretly wagers an amount and answers a clue, with the highest score after all rounds winning the game.

C. Significance of Daily-Double Wagering

- Winning or losing often hinges on a contestant's DD wagering strategy.
- Watson’s advanced DD wagering system leveraged reinforcement learning and opponent modeling, outperforming human players in this critical aspect of the game.

D. Watson's Approach to DD Wagering

1. **Action Value Calculation**:
    - Watson calculated **action values** $(\hat{q}(s, \text{bet}))$ to estimate the probability of winning from a given game state (s) for each possible wager.
    - The optimal bet was selected as the one maximizing $(\hat{q}(s, \text{bet}))$ subject to risk management constraints.
2. **Components of** $(\hat{q}(s, \text{bet}))$:
    - **State-Value Function** $(\hat{v}(\cdot, w))$:
        - Estimated the probability of winning from any game state.
        - Learned using reinforcement learning with **nonlinear TD(λ)** and a multilayer ANN.
    - **In-Category Confidence** :
        - Estimated Watson's likelihood of answering the DD clue correctly.
        - Based on Watson's accuracy in the current category, calculated using statistics of right (r) and wrong (w) answers from previously played clues.
3. **Action Value Formula**:

$$\hat{q}(s, \text{bet}) = p_{\text{DD}} \cdot \hat{v}(S_W + \text{bet}, \ldots) + (1 - p_{\text{DD}}) \cdot \hat{v}(S_W - \text{bet}, \ldots)$$

    - SW: Watson's current score.
    - Combines expected values of winning after a correct or incorrect response to the DD clue.
4. **Risk Mitigation**:
    - Adjusted $(\hat{q}(s, \text{bet}))$ by subtracting a fraction of the standard deviation of Watson's correct/incorrect afterstate evaluations.
    - Prohibited bets that could reduce the worst-case scenario value below a predefined threshold.
    - These measures minimized downside risk while maintaining high overall performance.

E. Learning the Value Function

- **Reinforcement Learning**:
    - Watson learned the state-value function $(\hat{v}(\cdot, w))$ by simulating millions of games against models of human contestants.
    - Weights of the ANN were trained using **TD(λ)** with backpropagation.
- **State Representation Features**:
    - Player scores.
    - Remaining DDs and their dollar values.
    - Total clue values left in the game.
    - Progression of the game and other relevant metrics.
- **Opponent Models**:
    - Watson simulated games using three models of human performance:
        - **Average Contestant**: Based on all game data.
        - **Champion**: Based on statistics from games featuring the top 100 players.
        - **Grand Champion**: Based on the top 10 players.
    - These models were derived from a fan-created archive of ~300,000 historical clues, capturing details like clue ordering, DD locations, and betting behavior.

F. Monte Carlo Simulation for Endgame DD Wagering

- Near the end of games, Monte Carlo simulations refined Watson's wagers:
    - Simulated games to completion for each potential bet.
    - Averaged outcomes to estimate the value of each wager.
- This approach improved endgame decision-making accuracy, where precise calculations had a significant impact on winning probability.

Optimizing Memory Control

A. Introduction

- Focuses on optimizing **Dynamic Random Access Memory (DRAM)** controllers using **reinforcement learning (RL)** techniques.
- The aim is to replace traditional fixed-policy controllers with an **online learning approach** that adapts dynamically to changing workloads.

B. Background: DRAM and Memory Controllers

1. **DRAM**:
    - Main memory for most computers, storing data as electrical charges in cells.
    - Characteristics:
        - **High capacity** and **low cost**.
        - Requires frequent refreshing of cells.
        - Efficient access is challenging due to **latency issues**.
2. **Memory Controller**:
    - Interface between the processor and DRAM.
    - Manages read/write requests, high bandwidth, and low latency for program execution.

C. Challenges in Memory Control

- DRAM has strict rules for accessing data:
    - Rows must be activated before reading/writing and precharged after use.
- Controllers must efficiently handle:
    - High **overhead** from activate/read/write/precharge commands.
    - **Concurrency issues** with modern multi-core processors sharing the same DRAM.
- Inefficient scheduling results in:
    - Increased **latency**.
    - Reduced **throughput**.

D. How DRAM Access Works

1. **Memory Cells**:
    - Arranged in rows and columns; data stored in capacitors that require periodic refreshing.
2. **Commands**:
    - **Activate**: Opens a row for access.
    - **Read/Write**: Accesses data from an open row.
    - **Precharge**: Closes the row to access another.
    - Timing and sequencing of these commands impact throughput and latency.

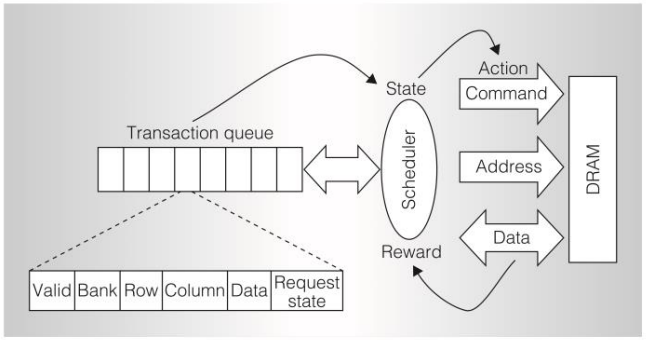

Figure 16.3: High-level view of the reinforcement learning DRAM controller. The scheduler is the reinforcement learning agent. Its environment is represented by features of the transaction queue, and its actions are commands to the DRAM system. ©2009 IEEE. Reprinted, with permission, from J. F. Martínez and E. İpek, Dynamic multicore resource management: A machine learning approach, *Micro, IEEE, 29*(5), p. 12.

E. Traditional Approaches

1. **Row Locality**:
    - Optimizes performance by processing as many requests as possible from an open row before moving to another row.
    - Reduces overhead from repeated activation and precharge commands.
2. **Scheduling Policies**:
    - Fixed scheduling policies like **First-Ready, First-Come-First-Serve (FR-FCFS)** prioritize requests based on readiness and arrival time.
    - Efficient but static, lacking adaptability to workload changes.

F. Reinforcement Learning-Based DRAM Controller

- **Markov Decision Process (MDP) Model**:
    - **State**: Current memory system status (e.g., transaction queue, open rows).
    - **Actions**: Commands like activate, read, write, precharge, or NoOp.
    - **Reward**: Signal indicating improvement in memory performance (e.g., higher throughput, lower latency).
    - **Goal**: Learn a policy that maximizes the overall reward.
- **Constraints**:
    - Actions like read/write are only valid after activation.
    - NoOp is used when no other action is valid.

G. Learning and Optimization

1. **Algorithm**:
    - **SARSA (State-Action-Reward-State-Action)** algorithm was used to iteratively refine the controller's policy.
2. **State Features**:
    - Number of requests in the transaction queue.
    - Time rows have been open.
    - Pending reads/writes.
3. **Tile Coding**:
    - State space is divided into smaller regions (tiles) for approximating the value function.
    - Tile-coded values are stored in **SRAM** for efficient access.

H. **On-Chip Implementation**:

- Value function stored in **static RAM (SRAM)** for quick access.
- Simulated on a **4-core, 4GHz chip**, with **10 processor cycles per DRAM cycle**, allowing multiple evaluations during each DRAM cycle.

I. Benchmarking and Performance

- Evaluated using **nine benchmark applications** including scientific and data-mining workloads.

- **Results**:
    - RL-based controller outperformed fixed-policy controllers by ~**8% on average**.
    - Dynamic adaptation allowed better handling of changing workload patterns.

J. Advantages of Online Learning

- Adapts dynamically to changing workloads, unlike fixed policies.
- Continuously improves scheduling based on workload demands, leading to consistent performance gains.

Human-Level Video Game Play

A. Introduction

1. **Challenge in Reinforcement Learning (RL):**
    - The primary challenge in applying RL to real-world problems lies in how **value functions** or **policies** are represented and stored.
    - For small, finite state spaces, a **lookup table** suffices. However, for larger problems, a **function approximation scheme** (linear or non-linear) is essential.
    - Function approximation relies on **features** that must:
        - Be accessible to the learning system.
        - Contain the information necessary for skilled performance.
2. **Traditional RL Approaches:**
    - Historically, **handcrafted features** (designed using human intuition) were used to encode state representations.
    - Example: TD-Gammon (Tesauro et al.):
        - Used features specific to backgammon for better performance.
3. **Breakthrough with DQN:**
    - DeepMind's **Deep Q-Network (DQN)** demonstrated that **deep learning** can automate feature extraction:
        - Removes the need for handcrafted features.
        - Achieves **human-level performance** on a variety of tasks.

B. Working of DQN

Input Representation

- **Raw Input**: 210×160-pixel RGB frames reduced to 84×84 grayscale.
- **Frame Stacking**: Last 4 frames stacked together (84×84×4) to partially address **partial observability** in games.
- **Purpose**:
    - Reduces memory and processing requirements.
    - Provides temporal context for decision-making.

Network Architecture

1. **Deep Convolutional Neural Network (CNN):**
    - **Three convolutional layers**:
        - 32 feature maps (20×20), 64 feature maps (9×9), and 64 feature maps (7×7).
    - **One fully connected hidden layer**:
        - 512 units.
    - **Output layer**:
        - 18 units (one for each possible action in the game).
    - **Activation Function**:
        - ReLU (max(0,x) for non-linearity.
2. **Action Representation**:
    - Each output unit corresponds to a possible game action.
    - The activation of an output unit represents the Q-value of the corresponding state-action pair.

Reward Signal

- Standardized across games:
    - **+1** for a score increase.
    - **-1** for a score decrease.
    - **0** otherwise.

- Ensures uniform scaling of rewards for all games.

Learning Algorithm

1. **Q-Learning Update Rule:**

$$w_{t+1} = w_t + \alpha \left[R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a, w_t) - \hat{q}(S_t, A_t, w_t) \right] \nabla \hat{q}(S_t, A_t, w_t)$$

   - w_t: Current network weights.
   - α: Learning rate.
   - R_{t+1}: Reward received after action A_t.
   - γ: Discount factor.
   - $\hat{q}$: Estimated Q-value.

2. **Gradient Computation:**
   1. Gradients are computed via **backpropagation**.
   2. **Mini-batches** of 32 experiences are used for smoother updates.

C. Key Features of DQN

1. Experience Replay

- Stores transitions $(S_t, A_t, R_{t+1}, S_{t+1})$ in a **replay memory**.
- Randomly samples mini-batches for updates.
- Advantages:
  - Reduces correlation between consecutive updates.
  - Reuses past experiences for improved data efficiency.

2. Target Network

- Maintains a separate **target network** with fixed weights for C steps.
- Targets are computed using the target network:

$$R_{t+1} + \gamma \max_a \tilde{q}(S_{t+1}, a, w)$$

- Stabilizes training by reducing oscillations caused by bootstrapping.

3. Error Clipping

- Clips the TD-error to $[-1, 1]$:

$$\text{TD-error} = R_{t+1} + \gamma \max_a \tilde{q}(S_{t+1}, a, w) - \hat{q}(S_t, A_t, w)$$

- Prevents large updates and improves stability.

4. RMSProp Optimizer

- Adjusts step size per weight based on the running average of gradient magnitudes.
- Accelerates convergence.

D. Atari 2600 as a Testbed

1. **Why Atari Games?**
    - The **Arcade Learning Environment (ALE)** provided a standardized platform for RL research using Atari games.
    - Atari 2600 games vary in:
        - **State-transition dynamics**.
        - **Action effects**.
        - **Policies** needed for success.
    - Despite their simplicity compared to modern games, they remain entertaining and challenging for humans.

2. **Setup for Human-Level Play:**
    - **Input**: DQN used raw video frames (grayscale 84×84×4 image stacks) for all 49 games.
    - **Training**: Each game was trained separately, with weights randomly initialized for each.
    - **Evaluation**:
        - Compared DQN scores with:
            - A **professional human player**.
            - A **random agent**.
            - Previous RL systems using handcrafted features.

E. Strengths of DQN

1. **Human-Level Play:**
    - Achieved **generalizable human-level performance** on diverse games without handcrafted features.
2. **Task Independence:**

- Same architecture and hyperparameters worked across all 49 games.

3. **Efficient Training:**
    - Experience replay and mini-batch updates made learning more stable and efficient.

F. Limitations of DQN

1. **Long-Term Planning:**
    - Failed on games requiring extensive planning (e.g., Montezuma's Revenge).
2. **Task-Specific Training:**
    - Separate training required for each game.
3. **Partial Observability:**
    - Stacking frames only partially addressed the lack of full observability in states.