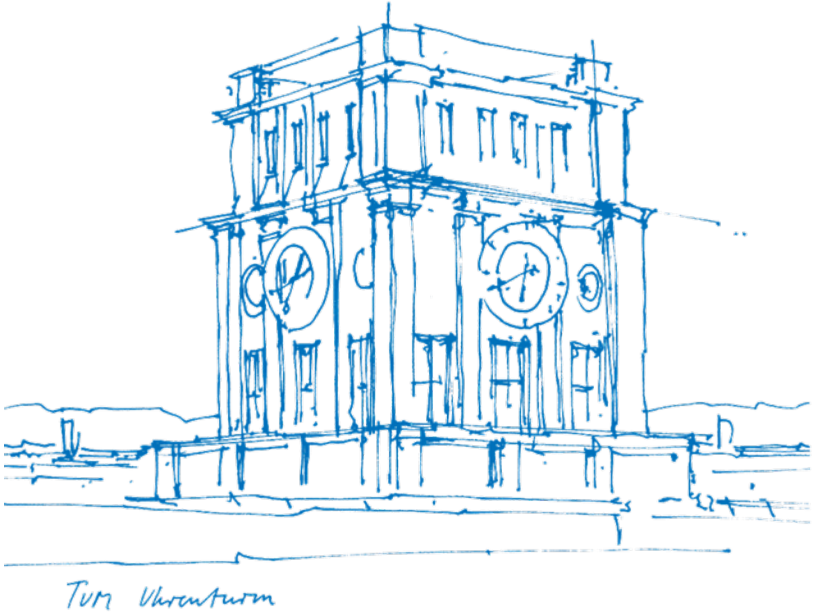
Checkmate with AI

Alexandros Stathakopoulos, Mohanad Kandil
Technische Universität München
Heilbronn, 09. Dezember 2024



Agenda

1. Historical Background
2. Introduction to RL
3. Introduction to Deep Q-Learning
4. Discussion of Algorithms
5. Project Planning

Historical Background

Historical Progression

- Rooted in trial-and-error learning (Thorndike's Law of Effect)
- Incorporates dynamic programming and Markov Decision Processes

Advance in Games

- **Samuel's Checkers Player (1959)**: Demonstrated RL in game strategies
  - Used a value function for state evaluation
  - Pioneered the use of **minimax strategy** combined with RL concepts
- **TD-Gammon (1992)**: Combined RL with neural networks for complex games (e.g., backgammon)

Introduction

Supervised

Data: (x,y)
x is the data, y is the label

Goal: Learn function to map
$x \longrightarrow y$

Apple Example:

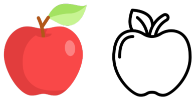
This object is an apple

UnSupervised

Data: (x,y)
x is the data, no labels!

Goal: Learn underlying structure

Apple Example:



This object is like the other object

Reinforcement Learning

Data: State Action Pairs

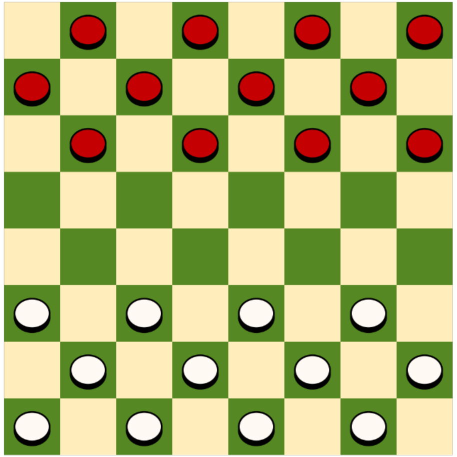
Goal: Maximize future rewards

Apple Example:

Eat this thing because it will keep you alive

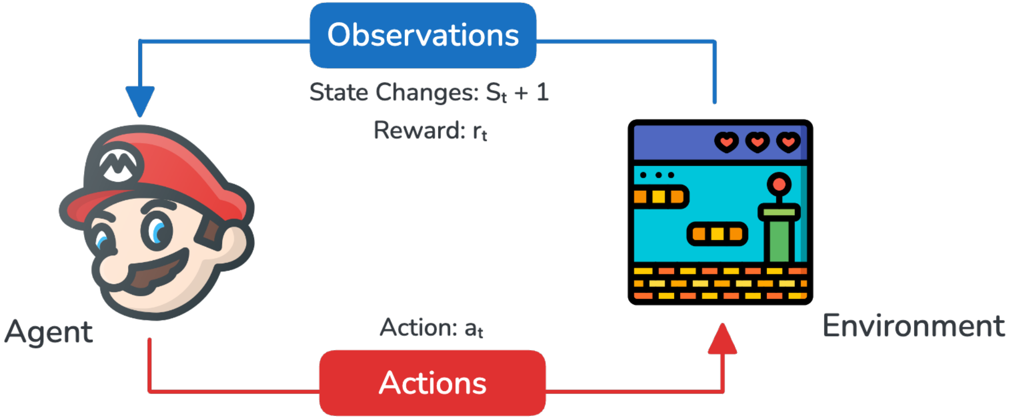
Introduction

- Checkers is a strategy game involving diagonal moves of pieces and captures by jumping over opponent pieces
- The objective of the project is to create an AI player using reinforcement learning
- Techniques: Q-learning and Deep Q-Networks (DQN)



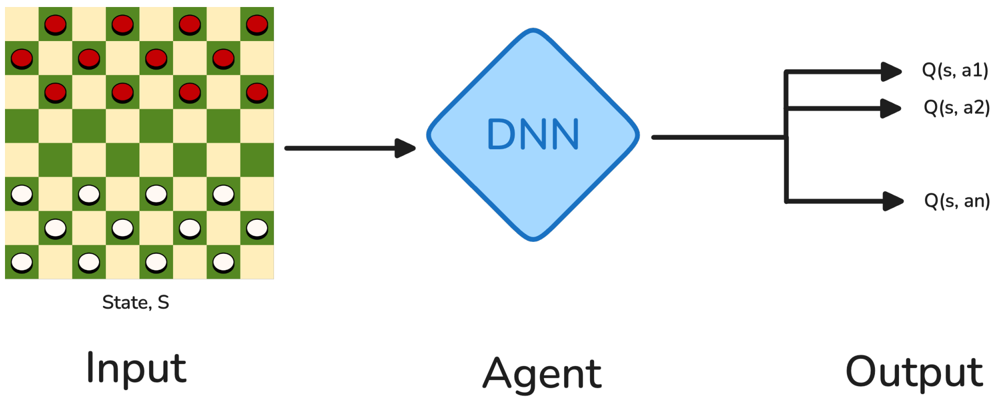
What is Reinforcement Learning?

- A machine learning paradigm focused on training agents to make sequences of decisions
- Key components:
  - **Agent**: Learns to act in an environment
  - **Environment**: The system with which the agent interacts
  - **Reward**: Feedback signal indicating the success of an action
- Goal: Maximize cumulative rewards over time



Deep Q-Learning

- reinforcement learning technique to find the optimal action-value function $Q(s, a)$
- It maps state-action pairs to their expected future rewards



Deep Q-Learning: Q-update rule

$$Q^{\text{new}}(s_t, a_t) \leftarrow Q(s_t, a_t) + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(r_t + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)}^{\text{temporal difference}}$$

Deep Q-Learning: Q-update rule

$$Q^{\text{new}}(s_t, a_t) \leftarrow Q(s_t, a_t) + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(r_t + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} - \underbrace{Q(s_t, a_t)}_{\text{old value}} \right)}^{\text{temporal difference}}$$

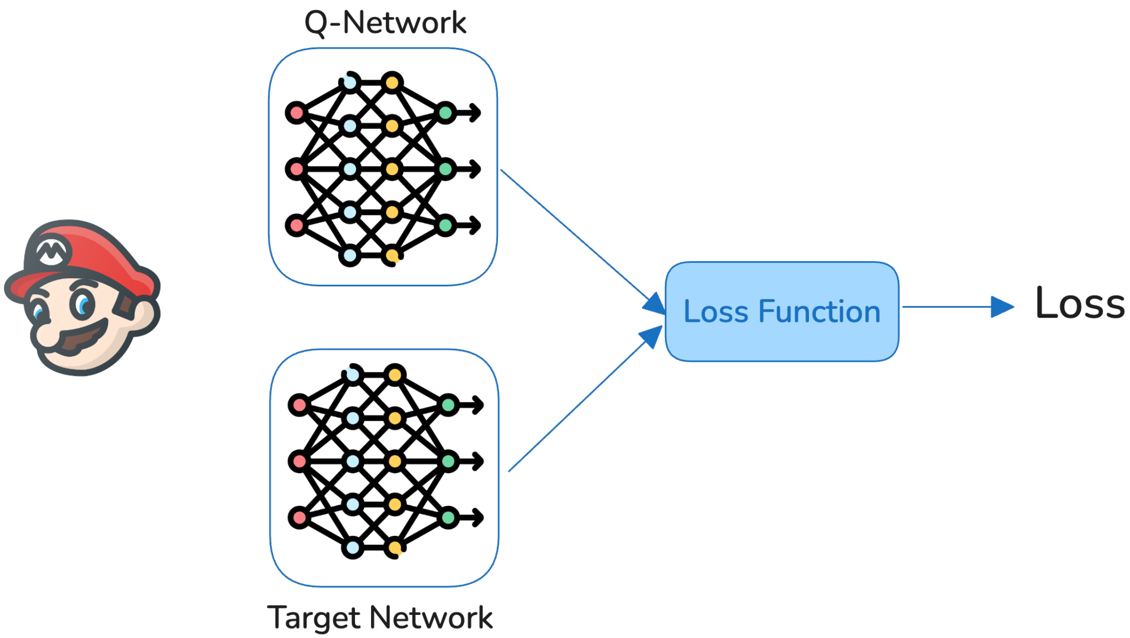
Challenge: Traditional Q-learning fails for large or continuous state spaces

Deep Q-Learning: Solution

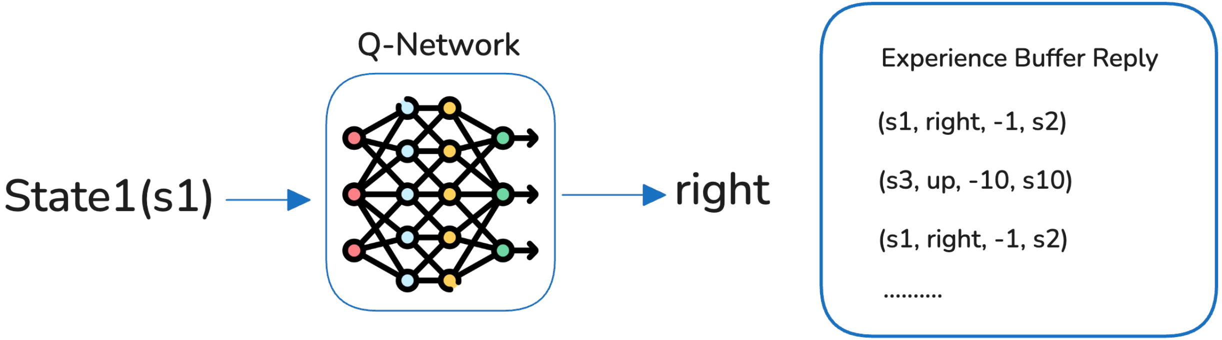
Use a **Deep Neural Network** (DNN) as a function approximator for $Q(s, a)$ instead of a table

- **Input**: State s, **Output**: Q-values for all actions a
- **Target Network**: stabilizes training by holding fixed weights for a few updates
- **Experience Replay**: samples random batches of past experiences (s, a, r, s') to break correlation in training data

Deep Q-Learning Algorithm



Deep Q-Learning Algorithm



Deep Q-Learning Algorithm

Use a **Deep Neural Network** (DNN) as a function approximator for $Q(s, a)$ instead of a table

Deep Q-Learning Algorithm

Use a **Deep Neural Network** (DNN) as a function approximator for $Q(s, a)$ instead of a table

1. Initialize:
   Q-Network $Q(s, a; \theta)$ and Target Network $Q'(s, a; \theta^-)$

Deep Q-Learning Algorithm

Use a **Deep Neural Network** (DNN) as a function approximator for $Q(s, a)$ instead of a table

1. Initialize:
   Q-Network $Q(s, a; \theta)$ and Target Network $Q'(s, a; \theta^{-})$
2. Action Selection:
   Use an $\varepsilon-$ greedy policy to balance exploration vs. exploitation

Deep Q-Learning Algorithm

Use a **Deep Neural Network** (DNN) as a function approximator for $Q(s, a)$ instead of a table

1. Initialize:
   Q-Network $Q(s, a; \theta)$ and Target Network $Q'(s, a; \theta^-)$
2. Action Selection:
   Use an $\varepsilon-$ greedy policy to balance exploration vs. exploitation
3. Store Experience:
   Add (s, a, r, s') to a replay buffer

Deep Q-Learning Algorithm

Use a **Deep Neural Network** (DNN) as a function approximator for $Q(s,a)$ instead of a table

1. Initialize:

   Q-Network $Q(s,a;\theta)$ and Target Network $Q'(s,a;\theta^{-})$
2. Action Selection:

   Use an $\varepsilon-$ greedy policy to balance exploration vs. exploitation
3. Store Experience:

   Add (s,a,r,s') to a replay buffer
4. Train the Network:
   - sample a minibatch from the buffer
   - Compute target Q-values: $y = r + \gamma \, max_{a'} \; Q'(s',a';\theta^{-})$
   - Minimize loss: $L(\theta) = (y - Q(s,a;\theta))^2$

Deep Q-Learning Algorithm

Use a **Deep Neural Network** (DNN) as a function approximator for $Q(s,a)$ instead of a table

1. Initialize:

   Q-Network $Q(s,a;\theta)$ and Target Network $Q'(s,a;\theta^{-})$
2. Action Selection:

   Use an $\varepsilon-$ greedy policy to balance exploration vs. exploitation
3. Store Experience:

   Add (s,a,r,s') to a replay buffer
4. Train the Network:
   - sample a minibatch from the buffer
   - Compute target Q-values: $y = r + \gamma\ max_{a'}\ Q'(s',a';\theta^{-})$
   - Minimize loss: $L(\theta) = (y - Q(s,a;\theta))^2$
5. Update Target Network:

   Periodically copy weights: $\theta^{-} \leftarrow \theta$

Key Improvements and Applications

Improvements:

- **Double DQN**: Reduces overestimation of Q-values
- **Dueling DQN**: Separates value and advantage functions
- **Prioritized Experience Replay**: Samples important experiences more frequently

Key Improvements and Applications

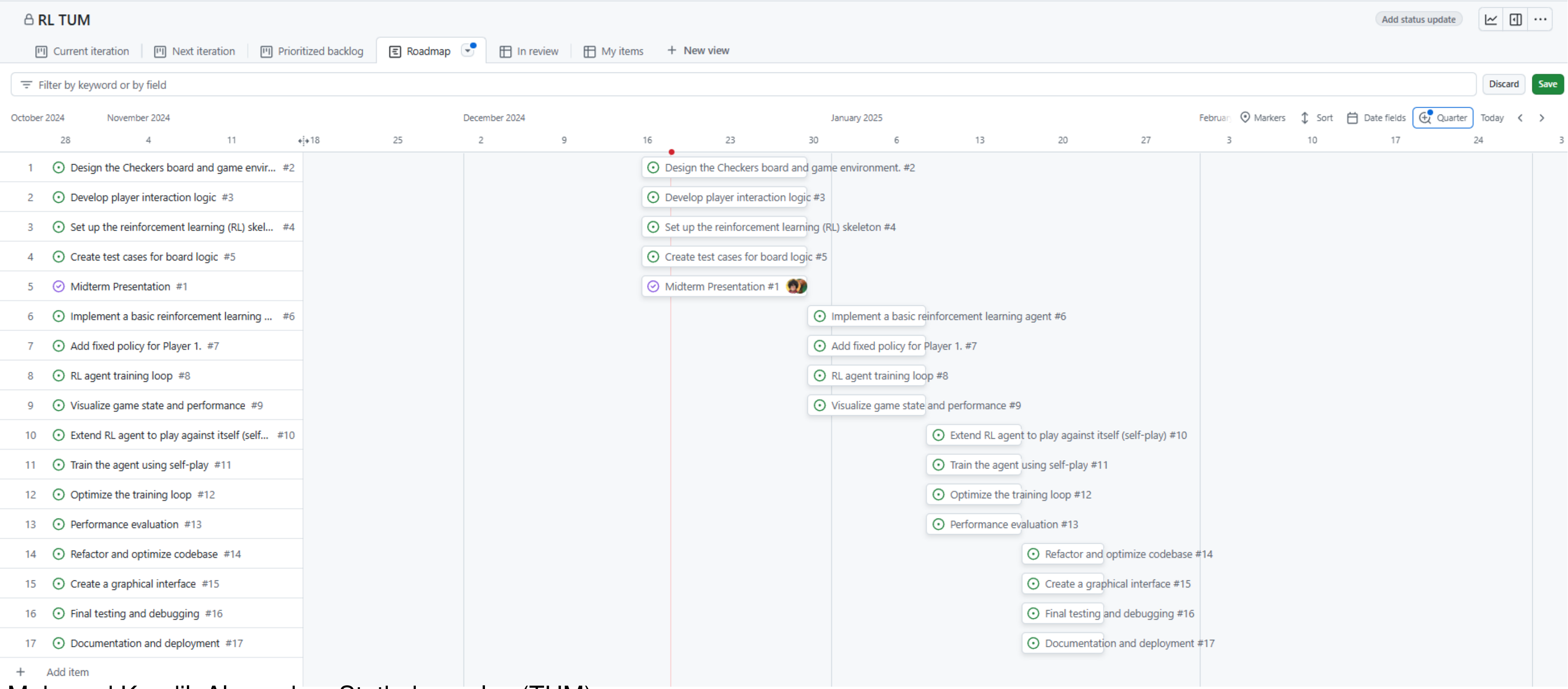
Improvements:

- **Double DQN**: Reduces overestimation of Q-values
- **Dueling DQN**: Separates value and advantage functions
- **Prioritized Experience Replay**: Samples important experiences more frequently

Applications:

- Games
- robotics
- autonomous systems

Project Planning & Timeline



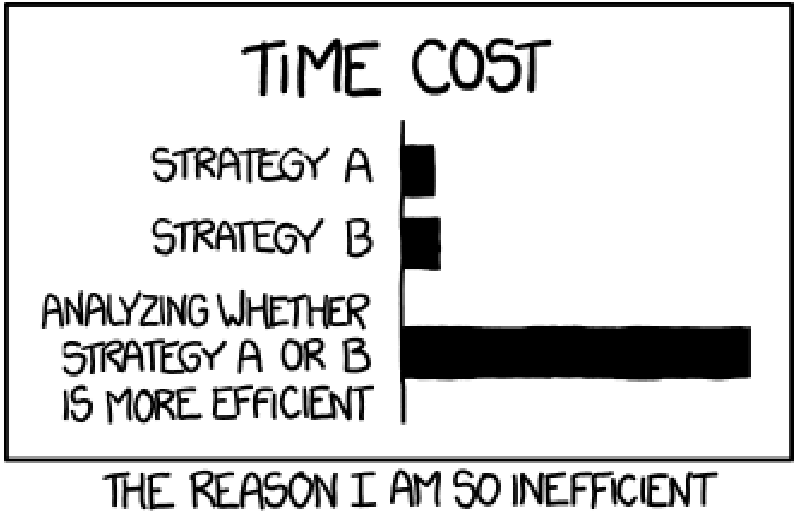
Software Stack

- Python
- numpy (for matrix calculations)
- pygame (fast UI)
- Web frameworks
- tensorflow (for ML implementation)

Future Work

- Fine-tuning the DQN model for improved decision-making
- Incorporating advanced techniques like Double DQN and Dueling DQN
- Testing against human players to evaluate real-world performance
- Extending the approach to other board games or strategy games (some side-projects)

Thank You



xkcd 1445