Swarm Reinforcement Learning with Graph Neural Networks

Bachelor's Thesis
of

Christian Burmeister

KIT Department of Informatics
Institute for Anthropomatics and Robotics (IAR)
Autonomous Learning Robots (ALR)

Referees: Prof. Dr. Techn. Gerhard Neumann
Prof. Dr. Ing. Tamim Asfour

Advisor: Niklas Freymuth

Duration: January 1st, 2019 — June 31st, 2019

Erklärung

Ich versichere hiermit, dass ich die Arbeit selbstständig verfasst habe, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, die wörtlich oder inhaltlich übernommenen Stellen als solche kenntlich gemacht habe und die Satzung des Karlsruher Instituts für Technologie zur Sicherung guter wissenschaftlicher Praxis beachtet habe.

Karlsruhe, den 31. Juni 2019

Christian Burmeister

Zusammenfassung

Einseitige deutsche Zusammenfassung (*Abstract*) der Abschlussarbeit. Unabhängig von der Sprache der Abschlussarbeit *muss* eine deutsche Zusammenfassung verfasst werden.

Abstract

The one-page abstract of the thesis.

Table of Contents

Zusammenfassung iii

Abstract iv

1. Introduction 3
- 1.1. Multiagent Systems . . . 3

2. Fundamentals 5

3. Related Work 8

4. Problem and Approaches 9
- 4.1. Definition of the Problemdomain . . . 9

5. Experiments 10

6. Evaluation 11

7. Conclusion and Future Work 12
- 7.1. Conclusion . . . 12
- 7.2. Future Work . . . 12

8. Notepad 13
- 8.1. Artificial Intelligence . . . 13
  - 8.1.1. !Artificial Intelligence - A modern Approach . . . 13
- 8.2. Ant Colony Optimization . . . 17
  - 8.2.1. !Wikipedia Article . . . 17
  - 8.2.2. Ant Colony Optimization (ACO) . . . 17
- 8.3. Reinforcement Learning . . . 17
  - 8.3.1. !Algorithmia Blog . . . 17
  - 8.3.2. !Freecodecamp . . . 18

8.3.3. !RL Lectures from Deepmind 24
8.3.4. Reinforcement Learning - An Introduction 41
8.3.5. Medium Blog Post 42
8.3.6. RL - Base - DQN - Human-level control through deep reinforcement - 2015 42
8.3.7. RL - Base - A3C - Asynchronous Methods for Deep Reinforcement Learning - 2016 43
8.3.8. RL - Sur - State-of-the-art Reinforcement Learning Algorithms - 2020 43
8.4. Deep Learning 43
8.4.1. Probabilistic Deep Learning Book 43
8.4.2. Deep Learning Book 43
8.5. Deep Reinforcement Learning 45
8.5.1. Deep RL Bootcamp 45
8.6. Multi-Agent Systems 50
8.6.1. !MAS - An Introduction to Multi-Agent Systems - 2010 50
8.6.2. !Artificial Intelligence - A modern Approach 55
8.6.3. MAS - App - Neural Networks for Continuous Online Learning and Control - 2006 56
8.6.4. MAS - Base - The Multiagent Planning Problem - 2016 56
8.6.5. MAS - Con - Distributed Cooperative Control and Communication for Multi-agent Systems - 2021 56
8.6.6. MAS - Con - PRIMA 2020 Principles and Practice of Multi-Agent Systems - 2021 56
8.6.7. MAS - Con - Swarm Intelligence - 2010 57
8.6.8. MAS - Con - Swarm Intelligence - 2012 57
8.6.9. MAS - Con - Swarm Intelligence - 2014 57
8.6.10. MAS - Con - Swarm Intelligence - 2016 57
8.6.11. MAS - Con - Swarm Intelligence - 2018 57
8.6.12. MAS - Con - Swarm Intelligence - 2020 57
8.6.13. MAS - Evo - Co-evolutionary Multi-agent System with Predator-Prey Mechanism for Multi-objective Optimization - 2007 57
8.6.14. MAS - Het - Multiagent Systems A Survey from a Machine Learning Perspective - 2000 58
8.6.15. MAS - Hie - Hierarchical Control in a Multiagent System - 2007 . . . 58
8.6.16. MAS - Hie - Holonic - A Taxonomy of Autonomy in Multiagent Organisation - 2003 58
8.6.17. MAS - Multi-Agent Systems - A Survey - 2018 58
8.6.18. MAS - Sur - A survey of multi-agent organizational paradigms - 2004 58
8.6.19. MAS - Tra - Transfer Learning for Multi-agent Coordination - 2011 . 58
8.6.20. MAS - Tra - Transfer learning in multi-agent systems through paralllel transfer - 2013 58
8.7. Game Theory 59
8.7.1. Artificial Intelligence - A modern Approach 59
8.8. Multi-Agent Reinforcement Learning 61
8.8.1. MARL - A Comprehensive Survey of Multiagent Reinforcement Learning - 2008 61

8.8.2. MARL - AC - Networked Multi-Agent Reinforcement Learning in Continuous Spaces - 2018 . . . 62
8.8.3. MARL - AC - Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environment - 2017 . . . 62
8.8.4. MARL - AC - Actor-Attention-Critic for Multi-Agent Reinforcement Learning - 2019 . . . 62
8.8.5. MARL - Base - Multiagent Reinforcement Learning - Theoretical Framework and an Algorithm - 1998 . . . 62
8.8.6. MARL - Base - Deep Reinforcement Learning for Robot Swarms - 2019 - KIT . . . 63
8.8.7. MARL - Base - PettingZoo - Gym for Multi-Agent Reinforcement Learning - 2020 . . . 63
8.8.8. MARL - Base - Multi-agent reinforcement learning weighting and partitioning - 1999 . . . 63
8.8.9. MARL - Com - Learning to Communicate with Deep Multi-Agent Reinforcement Learning - 2016 . . . 63
8.8.10. MARL - Com - Coordinating multi-agent reinforcement learning with limited communication - 2013 . . . 63
8.8.11. MARL - Het - LIIR - Learning Individual Intrinsic Reward inMulti-Agent Reinforcement Learning - 2019 . . . 63
8.8.12. MARL - Het - An approach to the pursuit problem on a heterogeneous multiagent system using reinforcement learning - 2002 . . . 64
8.8.13. MARL - Hie - Hierarchical multi-agent reinforcement learning - 2006 64
8.8.14. MARL - MO - Reward shaping for knowledge-based multi-objective multi-agent reinforcement learning - 2017 . . . 64
8.8.15. MARL - Role - ROMA Multi-Agent Reinforcement Learning with Emergent Roles - 2020 . . . 64
8.8.16. MARL - Sca - GAMA - Graph Attention Multi-agent reinforcement learning algorithm for cooperation - 2020 . . . 64
8.8.17. MARL - Sca - Plan-based reward shaping for multi-agent reinforcement learning - 2016 . . . 64
8.8.18. MARL - Sca - Multi-Agent Reinforcement Learning Using Linear Fuzzy Model Applied to Cooperative Mobile Robots - 2018 . . . 65
8.8.19. MARL - Sca - Stabilising Experience Replay for Deep Multi-Agent Reinforcement Learning - 2017 . . . 65
8.8.20. MARL - Sca - Mean Field Multi-Agent Reinforcement Learning - 2018 65
8.8.21. MARL - Sca - A modular approach to multi-agent reinforcement learning - 2005 . . . 65
8.8.22. MARL - Sur - Multi-Agent Reinforcement Learning - a critical survey - 2003 . . . 65
8.8.23. MARL - Sur - Multi-Agent Reinforcement Learning A Selective Overview of Theories and Algorithms - 2021 . . . 65
8.8.24. MARL - Sur - Multi-Agent Reinforcement Learning A Report on Challenges and Approaches - 2018 . . . 66
8.8.25. MARL - Sur - A Review of Cooperative Multi-Agent Deep Reinforcement Learning - 2019 . . . 66

8.8.26. MARL - Sur - A Survey on Transfer Learning for Multiagent Reinforcement Learning Systems - 2019 66
8.8.27. MARL - Tra - Transfer Learning in Multi-agent ReinforcementLearning Domains - 2011 66
8.8.28. MARL - Tra - Parallel Transfer Learning in Multi-Agent Systems What, when and how to transfer - 2019 66
8.8.29. MARL - Tra - Transfer among Agents An Efficient Multiagent Transfer Learning Framework - 2020 66
8.8.30. MARL - Tra - Agents teaching agents a survey on inter-agent transfer learning - 2019 67
8.9. Graph Neural Networks 67
8.9.1. !Theoretical Foundations of Graph Neural Networks - 2021 67
8.9.2. Probabilistic Deep Learning Book 69
8.9.3. Geometric Deap Learning 69
8.9.4. GNN - App - Optimizing Large-Scale Fleet Management on a Road Network using Multi-Agent Deep Reinforcement Learning with Graph Neural Network - 2020 70
8.9.5. GNN - Deep Multi-Agent Reinforcement Learning with Relevance Graphs - 2018 70
8.9.6. GNN - Deep Implicit Coordination Graphs for Multi-agent Reinforcement Learning- 2020 70
8.9.7. GNN - Multi-Agent Game Abstraction via Graph Attention Neural Network - 2020 70
8.9.8. GNN - Scaling Up Multiagent Reinforcement Learning for Robotic Systems Learn an Adaptive Sparse Communication Graph - 2020 . . 70
8.9.9. GNN - Graphcomm A Graph Neural Network Based Method for Multi-Agent Reinforcement Learning - 2021 70
8.9.10. GNN - Towards Heterogeneous Multi-Agent Reinforcement Learning with Graph Neural Networks - 2020 71
8.9.11. GNN - The Emergence of Adversarial Communication in Multi-Agent Reinforcement Learning - 2020 71
8.9.12. GNN - Multi-Agent Deep Reinforcement Learning using Attentive Graph Neural Architectures for Real-Time Strategy Games - 2021 . . 71
8.9.13. GNN - Global-Localized Agent Graph Convolution for Multi-Agent Reinforcement Learning - 2021 71
8.9.14. GNN - Specializing Inter-Agent Communication in Heterogeneous Multi-Agent Reinforcement Learning using Agent Class Information - 2020 71
8.9.15. GNN - Collaborative Multiagent Reinforcement Learning by Payoff Propagation - 2006 71
8.9.16. Write up for GCNs - 2016 72
8.9.17. GNN - Sur - A Comprehensive Survey on Graph Neural Networks - 2019 72

Bibliography 73

A. Example Appendix **74**

Chapter 1.

Introduction

Introduction. The topic setup here is only exemplary and might be different for your thesis, talk to your supervisor.

1.1. Multiagent Systems

Use Cases:

- Multiagent systems can be used in game theory and financing
- Reconnaissance robots covering a wide area. Communication not always possible.
- Smart Grid for Electricity, Power allocation, energy management.
- Flow Line Systems
- Stock markets
- Competitive pricing strategies
- Load Balancing.
- Network Systems (IoT).
- Traffic Light Control
- Autonomous Driving, Vehicular networks
- Automating turbulence modelling (aircraft design, weather forecasting, climate prediction).
- Control Systems for industrial processes.
- Intrusion Detection
- resource allocation for UAV Networks
- Large Scale City Traffic (Cityflow).

- Spectrum Management of cognitive radio using MARL.

Aspects:

- Ant-Colony-Optimization, which can be used for learning.
- Emergent Behavior.
- Swarm Intelligence.
- multi-agent reinforcement learning.
- multi-agent learning.
- game theory

Chapter 2.

Fundamentals

This chapter will introduce the necessary concepts that need to be understood. The baseline is a bachelor's degree in computer science without any assumptions made about the elective studies.

Topics:

- multiagent/multibody Systems (MAS).
  - MAS Reinforcement Learning
    - They use stochastic games (Markov Games) as generalization of Markov Decision Processes.
  - Hierarchical MAS, Hierarchical Reinforcement Learning for MAS.
  - MAS with Cooperation and Competition.
  - Particle Swarms (nicht so meins).
  - Problems:
    - MAS Movement Problems (Potential Fields). Mean fields?
    - ? using MAS for Moving a Multi-Legged Robot (Spider-like) with a navigation problem design as a hierarchical MAS?
    - Path Planning Navigation with Heterogeneous Agents?
    - MAS Task Problems: Rendezvous, Pursuit Evasion (Single and one Evader) (Boid?), (MAS - Deep Reinforcement Learning for Robot Swarms - 2019 - KIT)
    - Multi-Agent Path Finding (MAPF). Scalability for this: For fixed space they get into each others way.
    - Collective Foraging: (Ants-kinda). Problem when communication only happens in an area, use local information exchange groups. Information

Transfer. (MAS - Swarm Intelligence - 2020), Preferential Foraging (MAS - Swarm Intelligence - 2018 - p.289)
    * Coverage: Multi-robot Information Gathering / Scouting. (MAS - Swarm Intelligence - 2020), Pattern Formation (MAS - Swarm Intelligence - 2016 - p.14)
    * Coalition: Heterogeneous Group of Agents. They have different skills / attrributes that affect the environment. Like an Ant Caste System? Some Agents have better sensors? Only some agents have some sensors? What if the specialization is taken to the extreme? (MAS - Swarm Intelligence - 2020). Limited Visibility Sensors (MAS - Swarm Intelligence - 2018 - p.56). Going from a homogeneous group of agents, to randomized specialities, to extrem specializations. How does it change? Mixed with an Hierarchical Approach? They need to find groups to work together? Some are fast (but cannot see much, there is an insect that cannot see while running), Some have good sensors. Communication range? Genetic Diversity, Task-Allocation and Task-Switching (MAS - Swarm Intelligence - 2016 - p.109)
    * Collective Gradient Perception: Using Abilities of other Agents to take advantage of the whole group. (Flocking) (MAS - Swarm Intelligence - 2020)
    * Indirect Communication through changing states in the environment (birds transport something via cable). Also like using Pheromone Trails (Quality-Sensitive Foraging through virtual pheromone trails). (MAS - Swarm Intelligence - 2018 - p.15 - p.147)
    * Control Architecture: Behavior Trees, FSM. (MAS - Swarm Intelligence - 2018 - p.42)
    * Maze-Like Environment with Ant Algorithms (MAS - Swarm Intelligence - 2018 - p.162)
    * Search and Rescue? (Kinda like Foraging?)
    * Disruption: Disrupting Aspects of the Swarm and how they react to it, Swarm Attack: (MAS - Swarm Intelligence - 2018 - p.225), Coherence of Collective Decision Making (MAS - Swarm Intelligence - 2018 - p.264)
    * Evolutionary Systems: NEAT (MAS - Swarm Intelligence - 2012 - p.98)
  - Graph-Based Visualisation for MAS. (MAS - Swarm Intelligence - 2010). How do you visualize them?
  - Transfer Learning for MARL/MAS
    * Some approaches for parallel transfor of different problems even for MARL Problems.
    * So you can transfer even in parallel.
    * But they only transfer between similar problems. Which would held if you can create a simpler version of your problem and make it more and more complex.
    * Are there transfer learning approaches for MAS/MARL, so that Learning can be transfered between agents? So that if you add agents the complexity isn't as steep?
  - Adaptive Learning for MAS?

  - Control System: Either fully self-organizing or completely centralized. Hybrid Control of Swarms (MAS - Swarm Intelligence - 2018 - p.69)
  - Simulation of MAS: ARGoS
  - Best-of-n Problem: Swarm selects best option out of n alternatives. (MAS - Swarm Intelligence - 2018 - p.251)
  - Sensory Errors for Foraging, Dynamic Task Partiotioning (MAS - Swarm Intelligence - 2016 - p.124), Task Partitioning Problem (MAS - Swarm Intelligence - 2012 - p.122)
  - Task Hierarchy, Multi-Objective
  - Random Walks as a search strategy (MAS - Swarm Intelligence - 2016 - p.196)
  - Critic: Centralized Critic or Learning individual intrinsic reward (LIIR)
  - Standardizing Testing Scenarios (PettingZoo).
  - Role concept to for MAS. Agents with similar role share similar behavior.
  - Modular Approach to MARL to remedy the poor scalability in the state-space in the number of partner agents.
  - Weighting and Partitioning to decrease complexity.
  - Hierarchical Groups of MAS where each epoch each Group (5 Agents) exchange Data for learning. And every 10 Epoch the groups exchange data for learning? Randomize these groups? (every few epoch?). Groups of 5 that get reshuffled every 10 Epoch or so.
  - holonic agent structure: fractals structure of MAS. holonic and heterogenous? Does this form naturally for extreme heterogenous?
  - holonic coalitions?
- GNN
  - How many Hops? Use 2-hop neighborhoods: https://youtu.be/H6oOhElB3yE?t=842 (Realworld networks have a small diameter => cannot afford 3-hops).
  - Which GNN Type for MARL? Convolutional, Attentional, Message-passing GNN.
  - Where do you apply GNNs? As the communication between the agents (preprocess, communication relevance or state representation), or the actual function to learn with RL? (The Graph needs to emcompass the q-function. The reward.), or a relation graph.
  - Don't learn the graph structure, learn the weights. MAS: Nodes are Agents, Edges are their communications.
  - Complete Graph: Weights dynamic threshhold. What an agent can see defined over the weights.
- RL
  - un-discounted rewards need to fulfill either an average reward (average reward RL) or certain technical conditions must be met so un-discounted MDP are guaranteed to terminate. (well defined). How does this work?
  - Uncertainty in MDP: Use bayesian to not solve one MDP but a distribution of MDP. Use the uncertainty in your representation of the MDP itself non explicit uncertainty. States could have different discount factors to model your uncertainty of informations you have of the environment.
  - transformations for a risk-sensitive MDP into a risk-unaware MDP.

Chapter 3.

Related Work

Related work.

Deisenroth et al. (2013)

Chapter 4.

Problem and Approaches

This Chapter is moreso a deep dive into the actual solution of the Swarm RL with GNN Algorithm. Our Architecture and stuff.

4.1. Definition of the Problemdomain

Chapter 5.

Experiments

This Chapter should talk about the experiments we run and how we are going to set them up. the tasks and some of the codebase.

Idee:

- Based on the KIT Paper "Deep Reinforcement Learning for Robot Swarms".
- Look at Approach from a handful of papers and start adjusting what KIT did by adding to it.
- Add GNN instead of mean of all the agents observations use a GNN here. (Mesasge Passing)
- Find Problems that the KIT-Approach cannot handle well or is just able to and show how my additions lets it tackle harder tasks.
- Improve Scalability on the KIT-Approach (to better see the Heterogeneity?)
- Aspects: Direct/Indirect Communication, Control-Architecture (centralized, hybrid, self-organization), Cooperative/Competetive Environments, Global/Decentralized Communication.
- Given the problem a completly self-organized approach would be super interesting (no centralized critic).
- Tasks: Start with one simple task. Can add more later. Predator-Prey, Foraging
- Heterogeneity: Would be cool to have 3 cases: fully-homogeneous, (reelle zahlen (0,1))-skill-heterogeneous, binary-skill-heterogeneous (communication + one skill)

Chapter 6.

Evaluation

This Chapter should will see the actual results in different tasks/environments, with different conditions or structures (like hetereogenity) and also the different approaches we tried and how well they performed against the baseline of the KIT algorithm.

Chapter 7.

Conclusion and Future Work

Some introductory paragraph.

7.1. Conclusion

Your conclusion.

7.2. Future Work

Your ideas about possible future works.

Chapter 8.

Notepad

Reference for short forms:
(App): Applications (AC): Actor-Critic (Base): Basic Papers (Book): Book (Com): Communication (Con): Conference (Evo): Evolutionary (Het): Heterogeneous (Hie): Hierarchy (MO): Multi-Objective (Role): Role-Based (Sca): Scaling (Sur): Survey/Reviews (Tra): Transfer Learning
! means done.

8.1. Artificial Intelligence

8.1.1. !Artificial Intelligence - A modern Approach

Agents and Environments (p.34)

- **agent**: anything that perceives its **environment** through **sensors** and acting upon that environment using **actuators**.
- **percept**: agent's perceptual inputs at any given instance. Percept sequence is a complete history of perception.
- agents choice of action decided upon the history of perception, but not anything it has not perceived.
- its behavior is described by the **agent function**, which is internally implemented by the **agent program**.

Rational Agent (p.36)

- **rational agent**: it does the correct thing. Correctness is determined by a performance measure, which is determined by the changed environment states.
- design **performance measures** according to what one actually wants in the environment, rather than according to how one thinks the agent should behave.
- rational depends on:
  - the performance measure that defines the criterion of success
  - the agent's prior knowledge of the environment.
  - The actions that the agent can perform.
  - The agent's percept sequence of data.
- depending on the measures the agent might be rational or not.
- an **omniscient agent** knows the actual outcome of its actions and can act accordingly, but this is impossible in reality.
- rationality maximizes expected performance, while perfection (omniscient) maximizes actual performance.
- agents can do actions in order to modify future percepts, called **information gathering, or exploration**.
- rational agents learn as much as possible from what it perceives.
- his knowledge can be augmented and modified as it gains experience.
- if the agent relies on the prior knowledge of its designer rather than on its own percepts, we say that the agent lacks **autonomy**.
- it should learn what it can to compensate for partial or incorrect prior knowledge.
- give it some initial knowledge and the ability to learn, so it will become independent of its prior knowledge.

Nature of Environments (p.40)

- **task environments**: the "problems" to which rational agents are the "solutions".
- Describe the task environment in the following aspects P(Performance measure), E(Environment), A(Actuators), S(Sensors).
- **fully observable**: the agent's sensors give it access to the complete state of the environment. All aspects that are relevant to the choice of actions
- **partially observable**: otherwise. Because of missing sensors or noise.
- no sensors: unobservable
- single-agent environments and multi-agent environments.
- multi-agent can be either competitive (chess) or cooperative (avoiding collisions maximizes performance).
- **communication** emerges as a rational behavior in multiagent environments.
- randomized behavior is rational because it avoids the pitfalls of predictability.
- **Deterministic**: next state of environment is completely determined by the current state and the action executed by the agent, otherwise it is **stochastic**.
- you can ignore uncertainty that arises purely from the actions of other agents in a multiagent environment.

- If the environment is partial observable, it could appear to be stochastic, which implies quantifiable outcomes in terms of probabilities.
- an environment is **uncertain** if it is not fully observable or not deterministic.
- **episodic**: the agent's experience is divided into atomic episodes. In each the agent receives a percept and performs a single episode. The next episode does not depend on the actions taken in previous episodes, otherwise it is **sequential**.
- When the environment can change while the agent is deliberating, then the environment is **dynamic** for that agent otherwise it is **static**.
- if the environment itself does not change with the passage of time but the agent's performance score does, then we say the environment is **semi dynamic**.
- **discrete/continuous** applies to the state of the environment, to the way time is handled, and to the percepts and actions of the agents.
- **known vs. unknown**: refers to the agent's state of knowledge about the "laws of physics" of the environment. Known environment, the outcomes for all actions are given, otherwise the agent needs to learn how it works. An environment can be known, but partially observable (solitaire: I know the rules but still unable to see the cards that have not yet been turned over)
- hardest case: partially observable, multiagent, stochastic, sequential, dynamic, continuous, and unknown
- **environment class**: multiple environment scenarios to train it for multiple situations.
- you can create an **environment generator**, that selects environments in which to run the agent.

Structure of Agents (p.46)

- agent = architecture (computing device) + program (agent program).
- agent programs take the current percept as input and return an action to the actuators.
- agent program takes the current percept, agent function which takes the entire percept history.
- **table driven agent**: Uses a table of actions indexed by percept sequences. This table grows way to fast and is therefore not practical.

Simple reflex agents

- **simple reflex agents**: Select the actions on the basis of the current percept, ignoring the rest of the history.
- **condition-action-rule**: these agents create actions in a specific condition (if-then). These connections can be seen as reflexes.
- uses an **interpret-input** function as well as a **rule-match** function.
- they need the environment to be fully observable. They could run into infinite loops.
- you can mitigate this by using randomization for the actions. Which is non-rational for single agent environments.

Model-based reflex agents

- keep track of the part of the world an agent cannot see now. It maintains some sort of **internal state** that depends on the percept history.
- agents needs to know how the world evolves independently of the agent and how the agent's own actions affect the world.
- with this it creates a **model** of the world hence it is called model-based agent.
- it needs to update this state given sensor data.
- this model is a **best guess** and does not determine the entire current state of the environment exactly.

Goal-based agents

- an agent needs some sort of **goal information** that describes situations that are desirable. This can also be combined with the model.
- Usually agents need to do multiple actions to fulfill a goal which requires **search** and **planning**.
- this also involves consideration of the future.
- the goal-based agent's behavior can be easily changed to go to a different destination by using a goal where a reflex agent needs completely now rules.

Utility-based agents

- goals provide a crude binary distinction between good and bad states.
- use an internal **utility function** to create a performance measure.
- if the external performance measure and the internal utility function agree, the agent will act rationally.
- if you have conflicting goals the utility function can specify the appropriate **tradeoff**.
- if multiple goals cannot be achieved with certainty, utility provides a way to determine the **likelihood** of success.
- a rational utility-based agent chooses the action that **maximizes the expected utility**.
- any rational agent must behave as if it possesses a utility function whose expected value it tries to maximize.
- a utility-based agent must model and keep track of its environment.

Learning Agents

- it allows the agent to operate in initially unknown environments and to become more competent than its initial knowledge alone might allow.
- 4 conceptual components: **learning element** (responsible for improvements), **performance element** (select external action), **critic** (gives feedback to change the learning element), **problem generator** (suggesting actions that lead to new and informative experiences).
- critic tells the learning element how well the agent is doing given a performance standard. It tells the agent which percepts are good and which are bad.
- problem generator allows for exploration and suboptimal actions to discover better actions in the long run.
- learning element: simplest case: learning directly from the percept sequence.

- the **performance standard** distinguishes part of the incoming percept as a reward or penalty that provides direct feedback on the quality of the agent's behavior.

How the components of agent programs work

- **atomic representation**: Each state of the world is indivisible. Algorithms like search and game-playing, Hidden Markov models and Markov decision models work like this.
- **factored representation**: splits up each state of a fixed set of variables or attributes which each can have a value. Used in constraint satisfaction algorithms, propositional logic, planning, Bayesian networks.
- **structured representation**: here the different states have connections to each other. Used in relational databases, first-order logic, first-order probability models, knowledge-based learning and natural language understanding.
- more complex representations are more **expressive** and can capture everything more concise.

8.2. Ant Colony Optimization

8.2.1. !Wikipedia Article

Ant Colony Optimization Algorithm, Wikipedia

- is used for solving computational problems which can be reduced to finding good paths through graphs.
- artificial ants locate optimal soluions by moving through a parameter space represneting all possible solutions.
- they record their positions and the quality of their solutions for later iterations to find better solutions (pheromones).

8.2.2. Ant Colony Optimization (ACO)

ACO - Ant Colony Optimization for learning Bayesian network - 2002

8.3. Reinforcement Learning

8.3.1. !Algorithmia Blog

Introduction to Reinforcement Learning

- **Policy Learning**: Policy is a function: (state) -> (action). (if you approach an enemy and the enemy is stronger than you, turn backwards).
- Can use Neural Nets to approximate complicated functions
- **Q-Learning / Value Functions**: (state, action) -> (value). It also adds in all of the potential future values that this action might bring you.
- Approximate Q-Learning Functions with Neural Nets: DQN (RL - DQN - Human-level control through deep reinforcement - 2015)
- Newer way to approximate Q-Functions: A3C (Tutorial, RL - A3C - Asynchronous Methods for Deep Reinforcement Learning - 2016)
- **Challenges**:
  - Reinforcement Learning requires a ton of training data, that other algorithms can get to more efficiently.
  - RL is a general algorithm. If the problem has a domain-specific solution that might work better than RL. Tradeoff between scope and intensity.
  - Most pressing Issue: Design of the reward function. it could get stuck in local optima

8.3.2. !Freecodecamp

An introduction to Reinforcement Learning

- $State\ S_t, Reward\ R_t, Action\ A_t$
- **Reward Hypothesis**: All goals can be described by the maximization of the expected cumulative reward: $G_t = \sum_{k=0}^{T} R_{t+k+1}$
- But as earlier rewards are more probable to happen you need to increase their perceived value. Therefore you need a factor $0 \leq \gamma < 1$.
- Large γ, Agent cares about long-term reward. Small γ, Agent cares more about short term reward.
- **Discounted Accumulative Rewards (return)**: $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1},\ where\ \gamma \in [0, 1)$
- **Episodic tasks**: starting point and an ending point (terminal state), this creates an episode.
- **Continuous Tasks**: Tasks that continue forever (no terminal state).
- Learning Methods: Collecting the rewards at the end of the episode for the feature (Monte-Carlo), or Estimate the rewards at each step (Temporal Difference Learning)
- **Monte-Carlo**: $V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)]$. Left-Side: $V(S_t)$ Maximum expected Future, Right-Side: $V(S_t)$ Former estimation of maximum expected future. α: learning rate.
- **TD-Learning**: $V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$. $R_{t+1} + \gamma V(S_{t+1})$ is the TD-Target. TD-Target is an estimation, by updating it via a one-step target.
- **Exploration/Exploitation Tradeoff**: Exploration (finding more information about the environment), Exploitation (using known information to maximize the reward). The Agent might find better rewards by doing exploration.
- **Value Based RL**: OPtimize the value function V(s), that tells us the maximum expected future reward.

  - The value of each state is the total amount of the reward an agent can expect to accumulate over the future, starting at that state.
  - $v_\pi(s) = \mathbb{E}_\pi[\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}|S_t = s]$. The Expected Reward given an State s.
  - The agent takes the state with the biggest expected reward.
- **Policy Based**: optimize the policy function $a = \pi(s)$, without using the value function, a being the action to take, given a state.
  - The policy can either be deterministic, or stochastic $\pi(a|s) = \mathbb{P}[A = a|S = s]$(output is a distribution probebility over actions.)
  - It directly indicates the best action to take for each step.
- **Model Based**: Model the environment. Each environment needs a different model foreach environment.
- Deep Reinforcement Learning: Uses deep neural networks to solve it.

Diving deeper into Reinforcement Learning with Q-Learning

- **Q-learning** is value-based RL.
- **Q(Quality)-Table** gives you foreach action-state pair a value which moves gives the best maximum expected future reward.
- you don't implement a policy, you improve the Q-table to always choose the best action. The values in the table need to be learned.
- Action-Value Function (Q-Function) takes state and action as input and returns the expeced future reward.
- $Q^\pi(s_t, a_t) = \mathbb{E}\ [\sum_{k=0}^{\infty}\gamma^k R_{t+k+1}|s_t, a_t]$
- As we explore the environment, the Q-table will give us a better and better approximation by iteratively updating Q(s,a) using the **Bellman Equation**.
- Algorithm process: 1. Initialize Q-Table -> 2. Choose action a -> 3. perform action -> 4. measure reward -> 5. update Q -> goto 2.
  - 1. Initialize: e.g. initialize everything 0
  - 2-3. choose an action. Use the epsilon greedy strategy. $0 \leq \varepsilon \leq 1$ defines the exploration rate. It starts of with 1. We start of doing alot of random guesses what actions to choose (exploration). It is like a chance. We reduce the epsilon progressively to do more exploitation of the knowledge we gained.
  - 4-5. update q: We update Q with the Bellman equation (given a new state s' and a reward r): $newQ(s,a) = Q(s,a) + \alpha[\Delta Q(s,a)]$, $\Delta Q(s,a) = R(s,a) + \gamma \max(Q'(s',a')) - Q(s,a)$
  - $\max(Q'(s',a'))$: Maxium expected future reward given the new s' and all possible actions at that new state. The highest Q-value between possible actions from the new state s'.

An introduction to Deep Q-Learning: let's play Doom

- Instead of using a **Q-table**, use a Neural Network that takes a state and **approximates Q-values** for each action based on that state.
- In a videogame states can be associated with frames. you need multiple state inputs (like 4).

- preprocessing is important to reduce the complexity of the states to reduce the computation time needed for training.
- **temporal limitation**: you need multiple frames to percept motion in the environment.
- using convolutional layers with ELU. Use fully connected layers with ELU and one output layer that produces the Q-value estimataion for each action.
- Making more efficient use of observed experience using experience Replay:
  - **Avoid forgetting previous experiences**: given that we use sequential samples from interactions with our environment, the network tends to forget the previous experiences. You could use previous experiences by learning it multiple times.
  - reducing correlation between experiences: every action affects the next state, the sequence of experiences can be highly correlated. If we train in sequential order we might risk the agent bein influenced by it. Two strategies:
  - stop learning while interacting with the environment. Play a little randomly to explore the state space. Then recall these experiences and learn from then, then play again with the updated value function.
  - This way you have better set of examples. This prevents reinforcing the same action over and over.
- $\Delta w = \alpha[(R + \gamma\, max_a \hat{Q}(s', a, w)) - \hat{Q}(s, a, w)] \bigtriangledown_w \hat{Q}(s, a, w)$
- $\Delta w = \alpha * TD - Error * Gradient\ of\ our\ Prediction$

Improvements in Deep Q Learning: Dueling Double DQN, Prioritized Experience Replay, and fixed Q-targets

- Fixed Q-targets:
  - We calculate **TD-Error** (aka the loss), but we don't have any idea of the real TD-target. Bellman equation states that the TD-Target is the reward of taking that action at that state plus the discounted highest Q-value for the next state.
  - But we use the weights for the target and the Q-value and therefore our Q-value and our target value shifts.
  - **Q-Targets**: Using a seperate network with a fixed parameter (w-tilde) for estimating the TD-Target. At every tau step, we copy the parameters from our DQN network to update the target network:
  - $\Delta w = \alpha[(R + \gamma\, max_a \hat{Q}(s', a, \tilde{w})) - \hat{Q}(s, a, w)] \bigtriangledown_w \hat{Q}(s, a, w),\ At\ every \tau step : \tilde{w} \leftarrow w$
- **Double DQN**: Handles the problem of the overestimation of Q-values.
  - TD-Target = Q-target = reward + discounted max-q.
  - How are we sure the best action for the next state ist the action with the highest Q-value, it depends on what actions we tried and what neighbors we explored.
  - In the beginning of the training the max-q value will obviously b noisy and can lead to false positives. Learning will be complicated.
  - Solution: When computing q-target, use two networks to decouple the action selection from the target Q-value generation
  - Use our DQN network to select what is the best action to take for the next state (the action with the highest Q-value). We use our target network to calculate the target Q-value of taking that action at the next state.

  - $argmax_a Q(s',a) = DQN\ choose\ action\ for\ next\ state,\ Q(s', argmax_a Q(s',a)) = Target\ network\ calculates\ the\ qvalue.$
  - $Q(s,a) = r(s,a) + \gamma Q(s', argmax_a Q(s',a))$
  - this helps us reduce the overestimation of q values and helps us train faster and have more stable learning.
- **Dueling DQN (aka DDQN)**: Seperate the estimator into two parts:
  - Q(s,a) can be decomposed as the sum of: V(s): the value of being at that state. A(s,a): the advantage of taking that action at that state (how much better it is to all other actions).
  - With DDQN, we seperate the estimator using two streams one for V(s) and one for A(s,a) and then combine these two streams through a special aggregation layer to get an estimate of Q(s,a). Two streams in the NN.
  - By decoupling the estimation we can learn which states are valuable without having to learn the effect of each action at each state.
  - Being able to calculate V(s) can be useful for state where their actions do not affect the environment in a relevant way.
  - Aggregation: Simply adding both streams will be problemantic for the back propagation, you can force the advantage function estimator to have 0 advantag at the chosen action. To do that, we subtract the average advantage of all actions possible of the state.
  - $Q(s,a;\theta,\alpha,\beta) = V(s;\theta,\beta) + (A(s,a;\theta,\alpha) - \frac{1}{\mathscr{A}} \sum_a' A(s,a';\theta,\alpha))$
  - $\theta : common\ network\ parameters,\ \alpha : advantage\ stream\ parameters,\ \beta : value\ stream\ parameters,\ the\ s$
  - This helps us accelerate the training. This helps us find much more relaible Q-values for each action by decoupling the estimation between two streams.
- **Prioritized Experience Replay**: Some experiences may be more important than others for our training, but might occur less frequently.
  - If we sample the experiences randomly these rich experiences that occur rarely have practilly no chance to be selected.
  - Use a priority. where there is a big difference between our prediction and the TD target, since it means that we have a lot to learn about it.
  - We use the absolute value of the magnitude of our TD-error: $p_t = |\delta_t| + e$, e = const, that assures that no experience has no 0 probability.
  - Put that priority in the experience of each replay buffer to select the experiences.
  - Do not go greedy prioritization: overfitting!. Stochastic prioritization: $P(i) = \frac{p_i^a}{\sum_k p_k^a}$, a reintroduces some randomness, a = 0 pure uniform randomness, a = 1 only select the experiences with the highest priorities.
  - To combat over-fitting by prioritization of high-priority samples use Importance sampling weights (IS): $(\frac{1}{N} * \frac{1}{P(i)})^b$, b = controls how much the w affects learning. Close to 0 at the beginning of learning and annealed up to 1 over the duration of training. Because these weights are more important in the end of learning when our q-values begin to converge.
  - To sort the replays use an unsorted sumtree

An introduction to Policy Gradients with Cartpole and Doom

- in policy-based methods we directly learn the policy function that maps state to action. we directly parameterize π
- Deterministic policies are used in deterministic environments. stochastic policy is used when the environment is uncertain. We call this process a Partially Observable Markov Decision Process (POMDP).
- **Advantage of Policy Gradients**:
  - **convergence**: policy-based methods have better convergence properties. value-based methods might oscillate alot. Policy based methods follow gradients we converge on a local maximum (worst case), or global maximum (best case).
  - Policy gradient are more effective in **high dimensional action spaces**: as Deep Q-learning is that their prediction assign a score for each eaction at each time step, given the current state.
  - Policy gradients **can learn stochastic policies**: value functions can't. In Policy we donÄt need to implement an exploration/explotation trade off.
- **Disadvantages of Policy Gradients**:
  - Alot of the time, they converge on a **local maximum** rather than on the global optimum.
  - **Slower convergence**: Then Deep Q-Learning.
- **Policy Search**: We have our policy π that has a parameter θ. THis pi outputs a probability distribution of actions.
  - $\pi_\theta(a|s) = P[a|s]$
  - Good policy: theta that maximizes the score function: $J(\theta) = E_{\pi\theta}[\sum \gamma r]$
  - **Steps**: 1st: Measure the quality of policy with a policy score function, 2nd: use policy gradient ascent to find best parameter theta that improves our policy.
  - **1st Step**: The Policy Score function J(theta):
    * Episodic environment: Calculate the mean of the return from the first time Step (G1): $J_1(\theta) = E_\pi[G_1 = \sum_{k=0}^{\infty} \gamma^k R_{1+k}] = E_\pi(V(s_1))$. We want a policy that optimizes G1, as this will be the best policy.
    * Continious Environment: We can use the average value, because we can't rely on a specific start state and their values are now weighted by the probability of the occurrence of the respected state: $J_{avgv}(\theta) = E_\pi(V(s)) = \sum d(s)V(s),\ where\ d(s) = \frac{N(s)}{\sum_s' N(s')}$
    * N(s) = Number of occurrences of the state.
    * use the average reward per timestap: $J_{avR}(\theta) = E_\pi(r) = \sum_s d(s) \sum_a \pi_\theta(s,a) R_s^a$. sum over a: Probability that I take this action a from that state under this policy, Rsa: immediate reward that I get.
  - **2nd Step**: Policy gradient ascent.
    * To maximize the score function J(theta), we need to do gradient ascent on policy parameters.
    * We use gradient ascent as the score function is not an error function (there we would use gradient descent.)
    * Goal: $\theta^* = \underset{\theta}{argmax} E_{\pi\theta}[\sum_t R(s_t, a_t)]$, Score function: $J(\theta) = E_\pi[R(\tau)]$
    * Problem: How do we estimate the Gradient with respect to theta, when the gradient depends on the unknown effect of policy changes on the state distribution?

    - Solution: $\bigtriangledown_\theta J(\theta) = E_\pi[\bigtriangledown_\theta (log\pi(\tau|\theta))R(\tau)], \pi(\tau|\theta) : policy\,function, R(\tau) : score\,function$
    - Update Rule: $\Delta\theta = \alpha * \bigtriangledown_\theta (log\pi(s,a,\theta))R(\tau)$
    - R(tau): High value: it means thatn on average we took actions that lead to high rewards. If it is low, we want to push down the probabilities of the actions seen.
  - Policy gradient can be improved with Proximal Policy Gradients (ensure that the deviations from the previous policy stays relatively small) and Actor Critic (a hybrid between value-based algorithms and policy-based algorithms).

An intro to Advantage Actor Critic methods: let's play Sonic the Hedgehog!

- **Actor Critic**: Hybrid method. Use two neural networks: A Critic that measures how good the action takesn is (value-based) and an Actor that controls how our agent behaves (policy-based).
- State of the art: **Proximal Policy Optimization (PPO)**, is based on Advantage Actor Critic.
- **Policy Gradient Problem**: Reward is done for-each episode, so small bad decisions will be averaged out. And we won't find an optimal policy.
- Use TD-Learning: $\Delta\theta = \alpha * \bigtriangledown_\theta * (log\pi(S_t, A_t, \theta)) * Q(S_t, A_t)$. We do update each step sou we don't use the total rewards R(t). The Critic model approximates the value function.
- The critic will help to find the policy and update their own way to provide better feedback.
- Actor: $\pi(s,a,\theta)$ Critic: $\hat{q}(s,a,w)$
- Weights: Policy: $\Delta\theta = \alpha \bigtriangledown_\theta (log\pi_\theta(s,a)) * \hat{q}_w(s,a)$, Value: $\Delta w = \beta(R(s,a) + \gamma\hat{q}_w(s_{t+1}, a_{t+1}) - \hat{q}_w((s_t, a_t))\ \bigtriangledown_w \hat{q}_w(s_t, a_t)$
- **Process**: At each time-step: current State St into Actor and Critic. Policy outputs Action At and receives a new State and a reward.
- The Critic computes the value of taking that action at that state and the actor updates is policy parameters (weights) using this q-value.
- To reduce the Variability: Use Advantage function: $A(s,a) = Q(s,a) - V(s)$ Q(s,a): q-value for action a in state s, V(s): average value of that state.
- This function calculates the extra reward I get if I take this action. A(s,a) > 0: our gradient is pushed in that direction, A(s,a) < 0: our gradient is pushed in the opposite direction.
- Use the TD-Error as an good estimator: $A(s,a) = r + \gamma V(s') - V(s)$
- Strategies: Synchronous: **A2C** (Advantage Actor Critic), Asynchronous: **A3C** (Asynchronous Advantage Actor Critic).
- A3C uses different agents in parallel on multiple instances of the environment. Each worker will update the global network asynchronously.
- Problem of A3C: Link. Because of asynchronous nature of A3C, some workers will be playing with older version of the parameters, thus the aggregating update will not be optimal. In A2C it waits for each actor to finish before updating the global parameters. Therefore the training will be more cohesive and faster.

- Each worker in A2C will ahve the same set of weights since, contrary to A3C, A2C updates all their workers at the same time. YOu can create multiple versions of environments and then execute them in parallel.

Proximal Policy Optimization (PPO) with Sonic the Hedgehog 2 and 3

8.3.3. !RL Lectures from Deepmind

RL Course by DeepMind
RL Course by DeepMind - Part 1: Introduction

- Actions may have long term consequences and rewards may be delayed. May need to sacrifice immediate reward to gain more long-term reward.
- $Observation\ O_t, Reward\ R_t, Action\ A_t, History\ H_t\ (sequence\ of\ O_t,\ A_t,\ R_t)$
- $State\ S_t$ (simpler information to determine what happens next, usually function of history: $S_t = f(H_t)$
- State Definitions:
  - enviroment state S_t^e is the enviroments private representation. Environment state not visible to the agent.
  - agent state S_t^a is the agents internal representation. Used to pick next action. $S_t^a = f(H_t)$
  - markov (property) state $A\ state\ S_t\ is\ Markov\ iff:\ \mathbb{P}[S_{t+1}|S_t] = \mathbb{P}[S_{t+1}|S_1,...,S_t]$. You only need the current state to infer the next state or the future. A helicopter state needs velocity. Otherwise you need the complete history to calculate velocity if it only stored position.
  - environment state S_t^e and the history H_t is Markov.
- Environments:
  - fully observability: agent directly observes environment state $O_t = S_t^a = S_t^e$. This is a Markov decision process (MDP).
  - partial observability: $S_t^a \neq S_t^e$. This is a partially observable Markov decision process (POMDP). Agent constructs it's own S_t^a.
  - partial observability state: complete history $S_t^a = H_t$, beliefs: $S_t^a = (\mathbf{P}[S_t^e = s^1],...,\mathbf{P}[S_t^e = s^n])$, recurrent NN: $S_t^a = \sigma(S_{t-1}^a W_s + O_t W_o)$ (linear transformation)
- Inside an RL Agent
  - policy (agent's behavior), value function (how good is state-action pair), model (agents representation of the environment).
  - model: predicts what the environment will do next. you don't need to do models.
  - Transitions: $\mathscr{P}$ predicts next state (dynamics). Rewards $\mathscr{R}$ predicts next immediate reward
  - e.g.: $\mathscr{P}_{ss'}^a = \mathbf{P}[S = s'|S = s, A = a]$, $\mathscr{R}_s^a = \mathbf{E}[R|S = s, A = a]$
  - model-free agent: Policy and/or Value Function and no model.
  - model-based agent: Policy and/or Value Function and a model. first build the dynamics of the environmant es with the model
- Problems with RL

- RL-Problem: Environment initially unknown and the agent learns by interaction.
- Planning-Problem: Environment-model is known from the start.
- Prediction: evaluate the future (given a policy) vs. Control: optimise the future (find the pest policy)

RL Course by DeepMind - Part 2: Markov Decision Processes

- Markov Processes:
  - Markov Decision Processes Describe the environment for RL and is fully observable.
  - State Transition: $\mathscr{P}_{ss'} = \mathbb{P}[S_{t+1} = s'|S_t = s]$.
  - This allows a Matrix to be defined: $\begin{pmatrix} \mathscr{P}_{11} & \dots & \mathscr{P}_{1n} \\ \vdots & & \\ \mathscr{P}_{n1} & \dots & \mathscr{P}_{nn} \end{pmatrix}$. Each Row sums up to 1
  - Markov Process: tuple $\langle \mathscr{S}, \mathscr{P} \rangle$. S is a (finite) set os states. and P is a state transition probability matrix.
- Markov Reward Processes:
  - A MRP is a Markov Processes with the additions: tuple $\langle \mathscr{S}, \mathscr{P}, \mathscr{R}, \gamma \rangle$.
  - R is a reward function $\mathscr{R}_s = \mathbb{E}[R_{t+1}|S_t = s]$ and $\gamma \in [0,1]$ is a discount factor.
  - G_t is the total discounted reward form time-step t. Value function v(s) (see above).
  - Bellam Equation: $v(s) = \mathbb{E}[G_t|S_t = s] = \mathbb{E}[R_{t+1} + \gamma G_{t+1}|S_t = s] = \mathbb{E}[R_{t+1} + \gamma v(S_{t+1})|S_t = s]$
  - This allows: $v(s) = \mathscr{R}_s + \gamma \sum_{s' \in S} \mathscr{P}_{ss'} v(s')$
  - In Matrix form: $\begin{bmatrix} v(1) \\ \vdots \\ v(n) \end{bmatrix} = \begin{bmatrix} \mathscr{R}_1 \\ \vdots \\ \mathscr{R}_n \end{bmatrix} + \gamma \begin{bmatrix} \mathscr{P}_{11} & \dots & \mathscr{P}_{1n} \\ \vdots & & \\ \mathscr{P}_{n1} & \dots & \mathscr{P}_{nn} \end{bmatrix} \cdot \begin{bmatrix} v(1) \\ \vdots \\ v(n) \end{bmatrix}$.
- Markov Decisions Processes:
  - A MDP Is a Markov reward Process with a finite set of actions. The State Transition and reward function now also depend on the action chosen.
  - stochastic policy: $\pi(a|s) = \mathbb{P}[A_t = a|S_t = s]$. They depend only on the current state. Policies are stationary (time-independent).
  - The state sequence given by any policy is itself a markov process (chain) $\langle \mathscr{S}, \mathscr{P}^\pi \rangle$. If we add the rewards we got through this policy induced sequence we get a MRP $\langle \mathscr{S}, \mathscr{P}^\pi, \mathscr{R}^\pi, \gamma \rangle$.
  - So: $\mathscr{P}^\pi_{s,s'} = \sum_{a \in \mathscr{A}} \pi(a|s) \mathscr{P}^a_{s,s'}$ and $\mathscr{R}^\pi_s = \sum_{a \in \mathscr{A}} \pi(a|s) \mathscr{R}^a_s$
  - So the transition dynamics and rewards are averaged over what our policy gives us.
  - state-value function $v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$
  - action-value function $q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$.
  - bellman equation for state-value functions: $v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s]$
  - bellman equation for action-value functions: $q_\pi(s,a) = \mathbb{E}_\pi[R_{t+1} + \gamma q_\pi(S_{t+1}, A_{t+1})|S_t = s, A_t = a]$
  - V-Step: $v_\pi(s) = \sum_{a \in \mathscr{A}} \pi(a|s) q_\pi(s,a)$. For a given state we average the actions we can take

- Q-Step: $q_\pi(s,a) = \mathscr{R}_s^a + \gamma \sum_{s' \in \mathscr{S}} \mathscr{P}_{ss'}^a v_\pi(s)$. For a given state how good is it to do a given action we averate the situations we could go to.
- Equation for v_π: $v_\pi(s) = \sum_{a \in \mathscr{A}} (\mathscr{R}_s^a + \gamma \sum_{s' \in \mathscr{S}} \mathscr{P}_{ss'}^a v_\pi(s))$. state-value relates to the state-value of the next step.
- Equation for q_π: $p_\pi(s,a) = \mathscr{R}_s^a + \gamma \sum_{s' \in \mathscr{S}} \mathscr{P}_{ss'}^a \sum_{a \in \mathscr{A}} \pi(a|s) q_\pi(s,a)$. q-value relates to the q-value of the next step.
- Bellman Expectation Equation in Matrix form: $v_\pi = \mathscr{R}^\pi + \gamma \mathscr{P}^\pi v_\pi$. Direct Solution $v_\pi = (I - \gamma \mathscr{P}^\pi)^{-1} \mathscr{R}^\pi$. $\mathscr{P}^\pi$ and $\mathscr{R}^\pi$ are averages.
- Optimality
  * optimal state-value function $v_*(s) = \underset{\pi}{max}\, v_\pi(s)$. optimal action-value function $q_*(s,a) = \underset{\pi}{max}\, q_\pi(s,a)$
  * discounted reward does not have the problem of infinite loops of positive rewards.
  * un-discounted rewards need to fulfill either an average reward (average reward RL) or certain technical conditions must be met so un-discounted MDP are guaranteed to terminate. (well defined).
  * partial ordering over policies: $\pi \geq \pi' \; if \; v_\pi(s) \geq v'_\pi(s), \forall s$
  * Optimal Policy: For any MDP there exists at least one an optimal policy thate is better or erqual to all other policies: $\pi_* \geq \pi, \forall \pi$
  * All optimal policies achieve the optimal value and optimal action-value function.
  * optimal policy through optimal q-function $\pi_*(a|s) = \begin{cases} 1 & if \; a = \underset{a \in \mathscr{A}}{argmax}\, q_*(s,a) \\ 0 & otherwise \end{cases}$.
  * There is always a deterministic optimal policy for any MDP.
  * Bellman Optimality Equation for v*. $v_*(s) = \underset{a}{max}\, \mathscr{R}_s^a + \gamma \sum_{s' \in S} \mathscr{P}_{ss'}^a v_*(s')$. This is a 1-step look ahead.
  * Bellman Optimality Equation for q*. $q_*(s,a) = \mathscr{R}_s^a + \gamma \sum_{s' \in S} \mathscr{P}_{ss'}^a \underset{a'}{max}\, q_*(s',a')$.
  * Bellman Optimality Equation is non-linear. No closed form solution (in general). Needs iterative solution: value-iteration, policy-iteration, Q-learning, Sarsa.

- Infinite and continuous MDPs
  - countably infinite state and/or action spaces: straightforward
  - continuous state and/or action spaces: Closed form for linear quadratic model (LQR)
  - continuous time: Requires partial differntial equations (Hamilton-Jacobi-Bellman).
- Partially observable MDPs
  - A POMDP is a tuple $\langle \mathscr{S}, \mathscr{A}, \mathscr{O}, \mathscr{P}, \mathscr{R}, \mathscr{Z}, \gamma \rangle$ with hidden states. Hidden Markov Model with actions.
  - $\mathscr{O}$ is a finite set of observations.
  - $\mathscr{Z}$ is an observation function, $\mathscr{Z}_{s'o}^a = \mathbb{P}[O_{t+1} = 0 | S_{t+1} = s', A_t = a]$
  - History changes $H_t = A_0, O_1, R_1, ..., A_{t-1}, O_t, R_t$
  - Beilief state is a probability over states conditioned on the history $b(h) = (\mathbb{P}[S_t = s^1 | H_t = h], ..., \mathbb{P}[S_t = s^n | H_t = h])$
  - A POMDP can be reduced to an infinite history tree or an infinite belief state tree.

- Undiscounted, average reward MDPs
  - Ergodic Markov Process: Is recurrent (each state is visited an infinite number of times) and Aperiodic (each state is visited without any systematic period)
  - An ergodic Markov process has a limiting stationary distribution $d^{\pi}(s) = \sum_{s' \in S} d^{\pi}(s') \mathscr{P}_{s's}$
  - An MDP is ergodic if the Markov chain induced by any policy is ergodic.
  - Average reward per time-step $\rho^{\pi} = \lim\limits_{T \to \infty} \frac{1}{T} \mathbb{E}[\sum_{t=1}^{T} R_t]$
  - extra reward due to starting from satte s: $\tilde{v}_{\pi}(s) = \mathbb{E}_{\pi}[\sum_{k=1}^{\infty}(R_{t+k} - \rho^{\pi})|S_t = s]$

RL Course by DeepMind - Part 3: Planning by Dynamic Programming

- Introduction:
  - **Dynamic** sequential or tomporal component to the problem.
  - **Programming** optimising a "programm", i.e. policy
  - can be used when problems can be divided into subproblems they can be solved individually and the result can be combined again.
  - property: Optimal substructure, Principle of optimality applies, Optimum by divide-solve-combine. Optimum of the pieces tell you about optimum of your problem.
  - property: Overlapping subproblems. They occur multiple times. Can cache and reuse Solutions.
  - MDP satisfy both these properties because of the bellman equation and the value function that stores and reuses solutions.
  - Dynamic Programming can be used for planning in an MDP. Planning: We already learned everything, now we need to solve the problem.
  - Plan to solve prediction problem: Given MDP and policy, output: value function of the policy.
  - Plan to solve control problem: Given MDP, output: optimal value function and therefore policy.
- Policy Evaluation:
  - Iterative Policy Evaulation by using Bellman expectation backup. $v_1(0 = no\ reward\ anywhere) \to v_2 \to ... \to v_{\pi}$.
  - Synchronous backup: $At\ each\ iter\ k+1\ \forall s \in S$: Update $v_{k+1}(s)\ from\ v_k(s')$. s' is sucessor state of s.
  - For each state make a 1step look-ahead with the bellman equation that uses the current value function as input. The result is the value for this state for the next value function. Then go over each state to have all the values for the next value function.
  - Asynchronous Backup later.
  - This converges to the best value function (proven later).
  - $v_{k+1}(s) = \sum_{a \in \mathscr{A}} \pi(a|s)(\mathscr{R}_s^a + \gamma \sum_{s' \in S} \mathscr{P}_{ss'}^a v_k(s'))$. $\mathbf{v^{k+1}} = \mathscr{R}^{\pi} + \gamma \mathscr{P}^{\pi} + \mathbf{v^k}$.
- Policy Iteration:
  - start of with arbitray value function. It doesn't matter where you start you will always end with the optimal policy as an MDP always has atleast one.

- Improve Policy: Step 1: **Evaluate** a given policy: $v_\pi(s) = \mathbb{E}[R_{t+1} + \gamma R_{t+2} + ...|S_t = s]$. Create value function of that policy. This needs multiple iterations of the bellman expectation backup.
- Improve Policy: Step 2: **Improve** the policy by acting greedily with respect to value function: $\pi' = greedy(v_\pi)$
- In general you need to iterate between these two steps. policy iteration always converges to π^*.
- acting greedily always makes the policy deterministic. Acting Greedily $\pi'(s) = \underset{a\in\mathscr{A}}{argmax} q_\pi(s,a)$
- the total reward if we acted greedily is at least as much as before we greedified it. $v_\pi(s) \leq v_{\pi'}(s)$.
- When improvements stops you have satisfied the bellman optimality equation. Therefore the policy we end up with is $v_*(s)$ and is optimal.
- Modified Policy Iteration:
  * you may not need to iterate until the value function is fixed as a crude approximation would already lead to the same greedy policy as the one you get after the value function is fixed. you may be able to save iterations.
  * with an ε-convergenc of value function or early stopping after k iterations. Both still converge on the optimal policy.
  * Why not update policy every iteration (k = 1): this is equivalent to value iteration (next section).

• Value Iteration:
- if my first action i choose is optimal and the policy i use after that is optimal, then the policy is optimal.
- Principle of Optimality: A policy $\pi(a|s)$ achieves the optimal value from state s, $v_\pi(s) = v_*(s)$ iff: For any state s' reachable from s π achieves the optimal value from state s', $v_\pi(s') = v_*(s')$.
- using the bellman optimality equation:
- If we know the solution to subproblems $v_*(s')$. Then solution $v_*(s)$ an be found by one-step lookahead: $v_*(s) \leftarrow \underset{a\in\mathscr{A}}{max} \mathscr{R}_s^a + \gamma \sum_{s'\in S} \mathscr{P}_{ss'}^a v_*(s')$
- value iteration: apply these updates iteratively. The values propagate through the states and we end up with the optimal value function.
- Intuition: start with final rewards (one step before goal) and work backwards. We use this here through our entire statespace.
- Iterative Value Iteration by using Bellman optimality backup. $v_1(0 = no\ reward\ anywhere) \rightarrow v_2 \rightarrow ... \rightarrow v_*$.
- Synchronous backup: $At\ each\ iter\ k+1\ \forall s \in S$: Update $v_{k+1}(s)\ from\ v_k(s')$. s' is sucessor state of s.
- Intermediate values functions in this iterative process may not correspond to any valid policy.
- $v_{k+1}(s) = \underset{a\in\mathscr{A}}{max} (\mathscr{R}_s^a + \gamma \sum_{s'\in S} \mathscr{P}_{ss'}^a v_k(s'))$. $\mathbf{v^{k+1}} = \underset{\mathbf{a}\in\mathscr{A}}{\mathbf{max}} (\mathscr{R}^\mathbf{a} + \gamma \mathscr{P}^\mathbf{a} + \mathbf{v_k})$.
- Prediction-Problem: Use Bellman Expectation Equation in the Iterative Policy Evaluation Algorithm.
- Control-Problem: Use Bellman Expectation Equation and Greedy Policy Improvement in the Policy Iteration Algorithm.

- Control-Problem: Use Bellman Optimality Equation in the Value Iteration Algorithm.
- Complexity for $v_\pi(s)$, $\mathcal{O}(mn^2)$. Complexity for $q_\pi(s,a)$, $\mathcal{O}(m^2n^2)$. For m actions and n states.

- Extensions to Dynamic Programming:
  - Asynchronous Dynamic Programming:
    - Once you created the new value for the new value function you can use this for the other states instead of having to do all states on the old values first. Can reduce computation. If you slected all states at least sometimes you are guaranteed to converge.
    - **In-Place Dynamic Programming**: Synchronous: you have two copies of the value function (new and old) in in-place you overwrite the old values in your one copy of the value function.
    - **Prioritised Sweeping**: Some measure how important how it is to update a state with the Bellman error $|\underset{a \in \mathcal{A}}{max} \left(\mathcal{R}_s^a + \gamma \sum_{s' \in S} \mathcal{P}_{ss'}^a v(s')\right) - v(s)|$
    - this uses a priority queue. Backup the state with the largest remaining Bellman error.
    - **Real-Time Dynamic Programming**: only select the states that the agent actually visits. Collect real samples and look which states to update.
  - Full-width and sample backups:
    - DP uses full-width backups: Every step we consider all possible branching factor (considering all actions and all states). For that we need to know the dynamics.
    - So we use just sample particular trajectory.
    - Use Sample backups. Using sample rewards and sample transitions instead of complete reward function and transition dynamics.
    - Advantages: Model-free, Breaks curse of dimensionality, Cost of backup is constant.
  - Approximate Dynamic Programming:
    - ???
- Contraction Mapping:
  - ??? Shows that they convert to the optimal. the solution is unique

RL Course by DeepMind - Part 4: Model-Free Prediction

- Monte-Carlo Learning
  - Monte Carlo (MC) learns from complete episodes (no bootsraping) and is model free.
  - **First-Visit Monte-Carlo Policy Evaluation**:
    - The first time-step t that state s is visited in this particular episode (might be visited more than once).
    - Increment counter $N(s) \leftarrow N(s) + 1$, and total return $S(s) \leftarrow S(s) + G_t$, Estimation $V(s) = S(s)/N(s)$.
    - Law of large numbers: $V(s) \rightarrow v_\pi(s)$ as $N(s) \rightarrow \infty$
  - **Every-Visit Monte-Carlo Policy Evaluation**:

    * every time-step t that state s is visited in this particular episode (might be visited more than once).
    * Increment counter $N(s) \leftarrow N(s)+1$, and total return $S(s) \leftarrow S(s)+G_t$, Estimation $V(s) = S(s)/N(s)$.
    * Law of large numbers: $V(s) \rightarrow v_\pi(s)\ as\ N(s) \rightarrow \infty$
  - Incremental Mean with sequence of means $(\mu_1, \mu_2, ...)$ $\mu_k = \frac{1}{k}\sum_{j=1}^{k} x_j = \mu_{k-1} + \frac{1}{k}(x_k - \mu_{k-1})$
  - **Incremental Monte-Carlo Updates**: Counter is incremented the same, but $V(S_t) \leftarrow V(S_t) + \frac{1}{N(S_t)}(G_t - V(S_t))$.
  - non-stationary Problem, forget old episodes: $V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t))$
- Temporal-Difference Learning
  - Temporal-Difference Learning (TD) learns from incomplete episodes (bootstrapping) and is model free.
  - TD(0): Update by one step do not use the actual return but the estimated return $R_{t+1} + \gamma V(S_{t+1})$ which is called the TD Target.
  - $V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t))$
  - TD error: $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$
  - TD can learn before knowing the final outcome (every step, MC must wait until end of episode).
  - TD can learn without the final outcome from incomplete sequences and therefore works in continuing (non-terminating) environments (MC only for episodic).
- TD vs MC
  - Return G_t is unbiased. TD Target is a biased estimate of $v_\pi(S_t)$
  - TD has low variance, some bias, MC has high variance, zero bias. the State is noisy but the bias from V is not equal to v_π.
  - MC also works for POMDP but TD(0) does not. But it's estimates with MC might not be good.
  - MC also updates the intermediate values not only the state we start with.
  - MC and TD converge $V(s) \rightarrow v_\pi(s)$ as experience $\rightarrow \infty$
  - MC converges to solution with minimum mean-squared error $\sum_{k=1}^{K}\sum_{t=1}^{T_k}(G_t^k - V(s_t^k))^2$ and exploits the Markov property (usually more effective there).
  - TD(0) converges to solution of max lilekihood markov model and does not exploits the Markov property (usually more effective in non-Markov).
  - MC: One full trajectory (that terminates), TD-0: One Step along the way. Dynamic Programming: One-Step Lookahead to compute the full expectation where you need to know the dynamics for the environment.
  - Bootstrapping: update involves an estimate, Sampling: update samples an expectation.
- $TD(\lambda)$
  - Let TD target look n steps into the future. $n = 1$ is TD(0), $n = \infty$ is MC (or n is termination).
  - n-step return: $G_t^{(n)} = \sum_{k=1}^{n} \gamma^{k-1} R_{t+k} + \gamma^n V(S_{t+n})$
  - n-step TD-learning $V(S_t) \leftarrow V(S_t) + \alpha(G_t^{(n)} - V(S_t))$
  - Forward-view:

    * Forward View of TD(γ): Using weights to average the n-step returns $(1-\gamma)\gamma^{n-1}$.
    * Forward Return: $G_t^{\lambda} = (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}G_t^{(n)}$.
    * Forward-view TD(γ): $V(S_t) \leftarrow V(S_t) + \alpha(G_t^g amma - V(S_t))$.
    * Update value function towards the λ-return. It looks into the future.
  - Backward-view:
    * online (backward) update: do updates immediately. offline updates: Do the update at the end of the episode.
    * Eligibility Traces: Combines recency and frequency of events as the cause of states $E_0(s) = 0,\ E_t(s) = \gamma\lambda E_{t-1}(s) + \mathbf{1}(S_t = s)$
    * TD-Error $\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t)$
    * update: $V(S) \leftarrow V(s) + \alpha\delta_t E_t(s)$
  - TD-0 is TD(lambda) with lambda = 0. TD(1) is similar to every-visit MC.

RL Course by DeepMind - Part 5: Model-Free Control

- Introduction
  - On-policy: Learn about policy π from experience sampled from π. Off-policy: Learn about policy π from experiences sampled from μ.
- On-Policy Monte-Carlo Control
  - Greedy MC policy evaluation: $\pi'(s) = \underset{a\in\mathscr{A}}{argmax}\ Q(s,a)$
  - Policy Improvement with e-Greedy: probability $1-\varepsilon$ choose greedy, probability ε choose random.
  - $\pi(a|s) = \begin{cases} \varepsilon/m + 1 - \varepsilon & if a^* = \underset{a\in\mathscr{A}}{argmax}\ Q(s,a) \\ \varepsilon/m & otherwise \end{cases}$
  - the stochasticity of the e-greedy policy improvement ensures that we at some reate et least explore everythings in the environment.
  - greedy action selection for model is a problem as you might only explore the most immediate reward/greedy and get stuck on a local maximum. epsion-greedy: it is guaranteed that you make progress.
  - after one episode you can already update the value function to something slightly better might aswell use this.
  - GLIE Monte-Carlo Control:
    * Greedy in the Limit with Infinite Exploration (GLIE).
    * All state-action pairs are explored infinitely many times.
    * Then the policy converges on a greedy policy.
    * epsilon-greedy is GLIE if epsislon is redecued to zero at $\varepsilon_k = \frac{1}{k}$.
    * kth Episode sampled from π.
    * $\forall S_t\ and\ A_t : N(S_t,A_t) \leftarrow N(S_t,A_t) + 1,\ Q(S_t,A_t) \leftarrow Q(S_t,A_t) + \frac{1}{N(S_t,A_t)}(G_t - Q(S_t,A_t))$
    * GLIE Monte-Carlo Control converges to the optimal action-value function $Q(s,a) \rightarrow q_*(s,a)$
- On-Policy Temporal-Difference Learning

  - TD has the following advantages over MC: Lower variance, online, imcomplete sequences.
  - On-Policy Control with Sarsa: Policy Evaluation Sarsa (λ), $Q \approx q_\pi$, Policy improvement: ε-greedy policy improvement.
  - Sarsa converges to th eoptimal action-value function iff: GLIE sequence of policies $\pi_t(a|s)$ and for the step-sizes $\alpha : \sum_{t=1}^{\infty} \alpha_t = \infty$, $\sum_{t=1}^{\infty} \alpha_t^2 < \infty$
  - n-Step Sarsa:
    * n-step Q-return $q_t^{(n)} = R_{t+1} + \gamma R_{t+2} + ... + \gamma^{n-1} R_{t+n} + \gamma^n Q(S_{t+n})$
    * Update $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha(q_t^{(n)} - Q(S_t, A_t))$
    * Forward weight $q_t^\lambda = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} q_t^{(n)}$
    * Forward Sarsa $Q(S_t, A_t) = \leftarrow Q(S_t, A_t) + \alpha(q_t^\lambda - Q(S_t, A_t))$
    * Eligibility Traces $E_0(s,a) = 0$, $E_t(s,a) = \gamma\lambda E_{t-1}(s,a) + \mathbf{1}(S_t = s, A_t = a)$
    * Backward Sarsa $\delta_t = R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)$, $Q(s,a) \leftarrow Q(s,a) + \alpha \delta_t E_t(s,a)$
- Off-Policy Learning
  - Evaluate target policy $\pi(a|s)$, while following behaviour policy $\mu(a|s)$
  - Learn about optimal policy while following exploratory policy.
  - Importance Sampling:
    * Estiamte the expectation of a different distribution $\mathbb{E}_{X \approx P}[f(X)] = \sum P(X) f(X) = \mathbb{E}_{X \approx Q}[\frac{P(X)}{Q(X)} f(X)]$
    * Weight return G_t according to similiarity between policies
    * Monte-Carlo $G_t^{\pi/\mu} = \frac{\pi(A_t|S_t)}{\mu(A_t|S_t)} \frac{\pi(A_{t+1}|S_{t+1})}{\mu(A_{t+1}|S_{t+1})} \cdots \frac{\pi(A_T|S_T)}{\mu(A_T|S_T)}$
    * Update value towards corrected return $V(S_t) \leftarrow V(S_t) + \alpha(G_t^{\pi/\mu} - V(S_t))$
    * TD $V(S_t) \leftarrow V(S_t) + \alpha(\frac{\pi(A_t|S_t)}{\mu(A_t|S_t)}(R_{t+1} + \gamma V(S_{t+1})) - V(S_t))$
  - Q-Learning control: Learning of action-values without importance sampling.
    * Next action is chosen using behaviour policy $A_{t+1} \sim \mu(\cdot|S_t)$ but consider an alternative $A' \sim \pi(\cdot|S_t)$
    * target policy π is greedy w.r.t. Q(s,a) $\pi(S_{t+1}) = \underset{a'}{argmax}\, Q(S_{t+1}, a')$
    * the behaviour policy μ is e.g. ε-greedy w.r.t Q(s,a)
    * Update $Q(S,A) \leftarrow Q(S,A) + \alpha(R + \gamma \underset{a'}{max}\, Q(S', a') - Q(S,A))$
    * Q-learning control converges to the optimal action-value function

	Full Backup (DP)	Sample Backup (TD)
Bellman Expectation Equation for $v_\pi(s)$	Iterative Policy Evaluation	TD Learning
Bellman Expectation Equation for $q_\pi(s,a)$	Q-policy Iteration	Sarsa
Bellman Expectation Equation for $q_*(s,a)$	Q-value Iteration	Q-Learning

RL Course by DeepMind - Part 6: Value Function Approximation

- Introduction

- For Large scale MDPs a lookup table is not feasible and it is to slow to learn. Therefore use function approximation.
- Generalise from seen states to unseen states and Update parameter w using MC or TD learning.
- Types $Input: s,\ Output: \hat{v}(s, \mathbf{w}), Input: s, a\ Output: \hat{v}(s, a, \mathbf{w}), Input: s,\ Output: \hat{q}(s, a_1, \mathbf{w}), ..., \hat{q}(s, a_m, \mathbf{w})$

- Incremental Methods
  - Use gradient of a differentiable function $J(\mathbf{w})$ to adjust the weights $\Delta\mathbf{w} = -\frac{1}{2}\alpha \bigtriangledown_{\mathbf{w}} J(\mathbf{w})$.
  - Goal: minimise mean-squared error between approximation $\hat{v}$ and true function v_π, $J(\mathbf{w}) = \mathbb{E}_\pi[(v_\pi(S) - \hat{v}(S, \mathbf{w}))^2]$
  - Stochastic gradient descent uses samples $\Delta\mathbf{w} = \alpha(v_\pi(S) - \hat{v}(S, \mathbf{w})) \bigtriangledown_w \hat{v}(S, \mathbf{w})$
  - Linear Function Approximation
    * Represent state by a feature vector $x(S) = \begin{pmatrix} x_1(S) \\ \vdots \\ x_n(S) \end{pmatrix}$
    * value function by linear combination of features $\hat{v}(S, w) = x(S)^T w = \sum_{j=1}^n x_j(S) w_j$
    * Objective function $J(w) = \mathbb{E}_\pi[(v_\pi(S) - x(S)^T w)^2]$
    * Update rule $\bigtriangledown_w \hat{v}(S, w) = x(S),\ \Delta w = \alpha(v_\pi(S) - \hat{v}(S, w))x(S)$
    * Table loolip for features $x^{table}(S) = \begin{pmatrix} \mathbf{1}(S = s_1) \\ \vdots \\ \mathbf{1}(S = s_n) \end{pmatrix}$
    * Table function $\hat{v}(S, w) = \begin{pmatrix} \mathbf{1}(S = s_1) \\ \vdots \\ \mathbf{1}(S = s_n) \end{pmatrix} \cdot \begin{pmatrix} w_1 \\ \vdots \\ w_n \end{pmatrix}$
  - Incremental Prediction Algorithm
    * As the target $v_\pi(s)$ is not given to us we need a supstitude, a target.
    * MC-Target $v_\pi(s) = G_t$, TD(0)-Target $v_\pi(s) = TD - Target = R_{t+1} + \gamma\hat{v}(S_{t+1}, w)$, TD(lambda) $v_\pi(s) = G_t^\lambda$
    * MC: G-t is unbiased, but noisy. We get supervised learning to "training data" $\langle S_1, G_1 \rangle, ...\langle S_T, G_T \rangle$
    * MC: converges to local optimum epen on non-linear functions.
    * TD(0): TD-target is biased sample. We still get supervised learning similar to MC.
    * TD(0): converges (close) to global optimum.
    * TD(lambda): Better
  - Incremental Control Algorithm
    * Evaluation: Approximate policy evaluation, Improvement ε-greedy policy improvement.
    * Approximate action-value function $\hat{q}(S, A, w) \approx q_\pi(S, A)$
    * Minimise Mean-Squared error $J(w) = \mathbb{E}_\pi[(q_\pi(S, A) - \hat{q}(S, A, w))^2]$
    * Use Stochastic Gradient Descent: $\Delta w = \alpha(q_\pi(S, A) - \hat{q}(S, A, w))\nabla_w \hat{q}(S, A, w)$
    * Linear: use feature vector to represent action-value function by linear combination $x(S, A)$ (Feature Vector).

        * Linear: Stochastic Gradient Descent: $\Delta w = \alpha(q_\pi(S,A) - \hat{q}(S,A,w))x(S,A)$
        * Targets: For MC: G_t, TD(0): $R_{t+1} + \gamma\hat{q}(S_{t+1},A_{t+1},w)$, Forward-TD(lambda): q_t^λ
- Batch Methods
    - Batch samples to find best fitting value function for our training data (batch).
    - The Batch samples are the experiences so far.
    - Least Squares Prediction
        * Given: Approximation $\hat{v}(s,w)$ and Oracle $v_\pi(s)$
        * Dataset of state-value pairs: $\mathscr{D} = \{\langle s_1, v_1^\pi\rangle, \langle s_T, v_T^\pi\rangle\}$
        * Least-squars minimizes sum-squared error between approximation and target oracle.
        * $LS(w) = \sum_{t=1}^T (v_t^\pi - \hat{v}(s_t,w))^2 = \mathbb{E}_\mathscr{D}[(v^\pi - \hat{v}(s,w))^2]$
        * This Dataset can be called experience replay
        * Stoachstic Gradient Decent: 1st: Sample state,value from experience, 2nd: apply $\Delta w = \alpha(v^\pi - \hat{v}(s,w))\nabla_w \hat{v}(s,w)$
        * Stochastic Gradient Descent converges to $w^\pi = \underset{w}{argmin}\ LS(w)$
        * DQN uses experience replay and fixed Q-targets
    - Linear Least Squares Prediction.
        * In Linear cases the normal prediction takes longer than necessary, so the following approach solves it directly:
        * $w = (\sum_{t=1}^T x(s_t)x(s_t)^T)^{-1} \sum_{t=1}^T x(s_t)v_t^\pi$
        * Direct solution scales $O(N^3)$ for N features.
        * In practice we do not know v_t^π, therefore we use Monte-Carlo, TD(0), or TD(lambda) as approximation.
    - Least Squares Policy Prediction
        * policy evaluation: use least squares Q-learning.
        * q-learning Update: $\delta = R_{t+1} + \gamma\hat{q}(S_{t+1},\pi(S_{t+1}),w) - \hat{q}(S_t,A_t,w)$, $\Delta w = \alpha\delta x(S_t,A_t)$
        * LSTDQ algorithm: solve fot total update = 0
        * $w = (\sum_{t=1}^T x(S_t,A_t)(x(S_t,A_t) - \gamma x(S_{t+1},\pi(S_{t+1})))^T)^{-1} \sum_{t=1}^T x(S_t,A_t)R_{t+1}$

RL Course by DeepMind - Part 7: Policy Gradient Methods

- Introduction
    - Advantages of Policy-Based RL
        * Better convergence properties
        * Effective in high-dimensional or continuous action spaces. Quality based methods need a max(). Policy-Based does not need that.
        * Can learn stochastic policies.
    - Disadvantages of Policy-Based RL
        * Typically converge to a local rather than global optimum.
        * Evaluating a policy is typically inefficient and high variance.
    - State Aliasing: Two states that look the same from your feature-vector representation. This can happen when a MDP does not hold the markov property all the time.

- When State Aliasing occurs a stochastic policy can do better than a deterministic one.
- Policy Objective Functions
  * Goal: given policy $\pi_\theta(s,a)$ with parameters θ, find best θ. How do we measure quality of policy?
  * $d^{\pi_\theta}(s)$ is stationary distribugion of Markov chain for π_θ
  * episodic environments: use start value: $J_1(\theta) = V^{\pi_\theta}(s_1) = \mathbb{E}_{\pi_\theta}[v_1]$
  * continuing environments: use average value: $J_{avV}(\theta) = \sum_s d^{\pi_\theta}(s) V^{\pi_\theta}(s)$
  * or: average reward per time-step $J_{avR}(\theta) = \sum_s d^{\pi_\theta}(s) \sum_a \pi_\theta(s,a)\mathscr{R}_s^a$
  * or: discounted average: $\frac{1}{1-\gamma} J_{avV}(\theta)$
- Policy based reinforcement learning is an optimisation problem. Find θ that maximises $J(\theta)$
- Approaches without gradient: Hill climbing, Simplex/ amoeba / Nelder Mead, Genetic algorithms.
- Usually greater efficiency with gradient: Gradient descent, conjugate gradient, quasi-newton

• Finite Difference Policy Gradient
  - parameter update: $\Delta\theta = \alpha\nabla_\theta J(\theta)$, where α is step-size.
  - policy gradient: $\nabla_\theta J(\theta) = \begin{pmatrix} \frac{\partial J(\theta)}{\partial \theta_1} \\ \vdots \\ \frac{\partial J(\theta)}{\partial \theta_n} \end{pmatrix}$
  - Computing Gradients by finite differences
    * naive approach
    * for each dimension $k \in [1,n]$:
    * Estimate kth partial derivative of objective function w.r.t θ
    * By pertubing θ by small amount ε in kth dimension: $\frac{\partial J(\theta)}{\partial \theta_k} \approx \frac{J(\theta+\varepsilon u_k) - J(\theta)}{\varepsilon}$
    * u_k is unit vector with 1 in kth component, 0 elsewhere.
    * Uses n evaluations to compute policy gradient in n dimensions.
    * Simple, noisy, inefficient.
    * Also works for non-differentiable policies.

• Monte-Carlo Policy Gradient
  - Likelihood Ratios:
    * Assume policy π_θ is differentiable whenever it is non-zero and we know the gradient $\nabla_\theta \pi_\theta(s,a)$
    * Likelihood ratio: $\nabla_\theta \pi_\theta(s,a) = \pi_\theta(s,a)\frac{\nabla_\theta \pi_\theta(s,a)}{\pi_\theta(s,a)} = \pi_\theta(s,a)\nabla_\theta log\pi_\theta(s,a)$
    * score function: $\nabla_\theta log\pi_\theta(s,a)$. This is similar to maximum likelihood!
    * Softmax policy: Weight actions using linear combination of features $\phi(s,a)^T\theta$.
    * Softmax policy: Probability of action $\pi_\theta(s,a) \propto e^{\phi(s,a)^T\theta}$
    * Softmax policy: score function $\nabla_\theta log\pi_\theta(s,a) = \phi(s,a) - \mathbb{E}_{\pi_\theta}[\phi(s,\cdot)]$ (how much more do i get compared to the usual)
    * Gaussian policy: Mean using linear combination of features $\mu(s) = \phi(s,a)^T\theta$. Variance can be fixed or parametrised.
    * Gaussian policy: Policy is Gaussian $a \sim \mathcal{N}(\mu(s), \sigma^2)$
    * Gaussian policy: score function $\nabla_\theta log\pi_\theta(s,a) = \frac{(a-\mu(s))\phi(s)}{\sigma^2}$

- Policy Gradient Theorem
  * One-Step MDPs: $J(\theta) = \mathbb{E}_{\pi_\theta}[r] = \sum_{s\in\mathscr{S}} d(s) \sum_{a\in\mathscr{A}} \pi_\theta(s,a)\mathscr{R}_{s,a}$, $\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta log\ \pi_\theta(s,a)r]$
  * Policy Gradient Theorem: For anny differentiable $\pi_\theta(s,a)$, and any objective function J the policy gradient is:
  * $\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta log\ \pi_\theta(s,a)Q^{\pi_\theta}(s,a)]$, $Q^{\pi_\theta}(s,a)$ is the long-term value.
  * Monte-Carlo Policy Gradient uses v_t as an unbiased sample.

• Actor-Critic Policy Gradient
- use critic to estimate $Q_w(s,a) \approx Q^{\pi_\theta}(s,a)$. This is similar to policy evaluation
- Critic: Updates action-value function parameters w, Actor: Updates policy paramters θ, in direction suggested by critic.
- Uses approximate policy gradient: $\nabla_\theta J(\theta) \approx \mathbb{E}_{\pi_\theta}[\nabla_\theta log\ \pi_\theta(s,a)Q_w(s,a)], \nabla\theta = \alpha\nabla_\theta log\ \pi_\theta(s,a)Q_w(s,a)$
- Simple: Linear approx: $Q_w(s,a) = \phi(s,a)^T w$. Critic: Updates w by linear TD(0). Actor: Updates θ by policy gradient.
- Compatible Function Approximation
  * Approximation of policy gradient introduces bias and may not find the right solution. a good function approximation can still be not biased.
  * If the following two conditions are satisfied:
  * 1. Value function approximator is compatible to the policy: $\nabla_w Q_w(s,a) = \nabla_\theta log\ \pi_\theta(s,a)$
  * 2. Value function parameters w minimise mean-squared error: $\varepsilon = \mathbb{E}_{\pi_\theta}[(Q^{\pi_\theta}(s,a) - Q_w(s,a))^2]$
  * Then the policy gradient is exact: $\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta log\ \pi_\theta(s,a)Q_w(s,a)]$
- Advantage Function Critic
  * We can subtract baseline function B(s) from the poolicy gradient to reduce variance. Can use state value function $B(s) = V^{\pi_\theta}(s)$.
  * rewrite policy gradient using the advantage function: $A^{\pi_\theta}(s,a) = Q^{\pi_\theta}(s,a) - V^{\pi_\theta}(s)$
  * $\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta log\ \pi_\theta(s,a)A^{\pi_\theta}(s,a)]$
  * The critic can reduce variance by extimating both $V^{\pi_\theta}(s)$ and $Q^{\pi_\theta}(s,a)$ with two parameter vectors and $V_v(s)$, $Q_w(s,a)$. Then update both value functions.
  * Value function TD-Error: $\delta^{\pi_\theta} = r + \gamma V^{\pi_\theta}(s') - V^{\pi_\theta}(s)$
  * Use it for the policy gradient: $\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}[\nabla_\theta log\ \pi_\theta(s,a)\delta^{\pi_\theta}]$
  * In practice we use one set of parameters for TD-Error: $\delta_v = r + \gamma V_v(s') - V_v(s)$
- Eligibility Traces
  * Critic Estimations for the value function target:
  * MC: $\Delta\theta = \alpha(v_t - V_\theta(s))\phi(s)$
  * TD(0): $\Delta\theta = \alpha(r + \gamma V(s') - V_\theta(s))\phi(s)$
  * TD(lambda): $\Delta\theta = \alpha(v_t^\lambda - V_\theta(s))\phi(s)$
  * Backward view TD with eligibility: $\delta_t = r_{t+1} + \gamma V(s_{t+1}) - V(s_t)$, $e_t = \gamma\lambda e_{t-1} + \phi(s_t)$, $\Delta\theta = \alpha\delta_t e_t$
  * Actor Estimations for the policy gradient:
  * MC: $\Delta\theta = \alpha(v_t - V_v(s_t))\nabla_\theta log\ \pi_\theta(s_t,a_t)$
  * TD(0): $\Delta\theta = \alpha(r + \gamma V_v(s_{t+1}) - V_v(s_t))\nabla_\theta log\ \pi_\theta(s_t,a_t)$
  * TD(lambda): $\Delta\theta = \alpha(v_t^\lambda - V_v(s_t))\nabla_\theta log\ \pi_\theta(s_t,a_t)$

        - Backward view TD with eligibility: $\delta = r_{t+1} + \gamma V_v(s_{t+1}) - V_v(s_t),\ e_{t+1} = \gamma\lambda e_t + \nabla_\theta log\ \pi_\theta(s_t, a_t),\ \Delta\theta = \alpha\delta e_t$
    - Natural Policy Gradient
        - A good ascent direction can speed up convergence.
        - The vanilla gradient is sensitive to a policy that is reparametrised.
        - Natural policy gradient finds ascent direction that is cosest to vanila gradient, when changing policy
        - $\nabla_\theta^{nat} \pi_\theta(s,a) = G_\theta^{-1} \nabla_\theta \pi_\theta(s,a)$
        - $G_\theta = E_{\pi_\theta}[\nabla_\theta log\ \pi_\theta(s,a)\ \nabla_\theta log\ \pi_\theta(s,a)^T]$. The fisher information matrix.
        - Natural policy gradient simplifies: $\nabla_\theta^{nat} J(\theta) = G_\theta w = w$. In the direction of the critic parameters.

RL Course by DeepMind - Part 8: Integrating Learning an Planning

- Introduction
    -
- Model-Based Reinforcement Learning
    -
    - Learning a Model
        -
    - Planning with a Model
        -
- Integrated Architectures
    - Dyna
        -
- Simulation-Based Search
    - Monte Carlo Search
        -
    - Temporal-Difference Search
        -

RL Course by DeepMind - Part 9: Exploration and Exploitation

- Introduction
    - Dilemma: Exploitation: Make the best decision given current information (acting according to max) v.s. exploration: gather more information.
    - Best long-term strategy may involve short-term sacrifices.
    - Naive Exploration: Add noise to greedy policy ε-greedy.
    - Optimistic Initialisation: Assume the best until proven otherwise.
    - Optimism in the Face of Uncertainty: Prefer actions with uncertain values.
    - Probability Matching: Select actions according to probability they are best.
    - Information State Search: Lookahead search incorporating value of information.
    - State-action exploration: Systematically explore state space / action space.
    - Parameter exploration: Pick different parameters and try for a while.
- Multi-Armed Bandits

- tuple $\langle A, R \rangle$. A is a set of m actions, R is reward probability distribution. Goal is to maximise cumulative reward $\sum_{t=1}^{T} r_t$
- Regret
  * mean reward: $Q(a) = \mathbb{E}[r|a]$, optimal value $V^* = Q(a^*) = \underset{a \in A}{max}\, Q(a)$
  * regret: opportunity loss: $l_t = \mathbb{E}[V^* - Q(a_t)]$, total regret $L_t = \mathbb{E}[\sum_{t=1}^{T} V^* - Q(a_t)]$
  * maximise cumulative reward = minimise total regret
  * count: $N_t(a)$ how often action a was selected. gap: $\Delta_a = V^* - Q(a)$ (difference of action and optimal action).
  * Regret as gaps and counts: $L_t = \sum_{a \in A} \mathbb{E}[N_t(a)]\Delta_a$.
  * Goal: small counts for large gaps => minimizes regret.
  * always explore and never explore will have linear total regret.
- Greedy and ε-greedy algorithms
  * greedy: Estimation $\hat{Q}_t(a) = \frac{1}{N_t(a)} \sum_{t=1}^{T} r_t \mathbb{1}(a_t = a)$. choose highest value: $a_t^* = \underset{a \in A}{argmax}\, \hat{Q}_t(a)$. Has linear total regret.
  * ε-greedy : With 1 - ε select $a = \underset{a \in A}{argmax}\, \hat{Q}(a)$ otherwise select random. Has linear total regret.
  * Optimistic Initilisation: Initialise Q(a) to high value => encourages systematic exploration early on => greedy or ε-greedy still has linear total regret.
  * Optimistic: Update action value by Monte-Carlo: $\hat{Q}_t(a_t) = \hat{Q}_{t-1} + \frac{1}{N_t(a_t)}(r_t - \hat{Q}_{t-1})$
  * Decaying ε-greedy: use decay schedule e.g.: $c > 0$, $d = \underset{a|\Delta_a>0}{min}\, \Delta_i$, $\varepsilon_t = min\{1, \frac{c|A|}{d^2 t}\}$
  * Decying has logarithmic asymptotic total regret but needs advance knowledge of gaps.
- Lower Bound
  * Hard problems have similar-looking arms with different means.
  * This is described fomally by the gap Δ_a and the similarity $KL(\mathscr{R}^a || \mathscr{R}^{a^*})$
  * Asymptotic total regret is at least logarithmic in number of steps: $\lim_{t\to\infty} L_t \geq log\ t \sum_{a|\Delta_a>0} \frac{\Delta_a}{KL(\mathscr{R}^a || \mathscr{R}^{a^*})}$
- Upper Confidence Bound
  * The more uncertain we are about an action-value the more important it is to explore that action => optimisim in the face of uncertainty.
  * Estimate an upper confidence $\hat{U}_t(a)$ for each action value. Such that $Q(a) \leq \hat{Q}_t(a) + \hat{U}_t(a)$ with high probability.
  * Select an action maximising Upper Confidence Bound (UCB): $a_t = \underset{a \in A}{argmax}\, \hat{Q}_t(a) + \hat{U}_t(a)$
  * Using $U_t(a) = \sqrt{\frac{2log(t)}{N_t(a)}}$ we have the UCB1 algorithm: $a_t = \underset{a \in A}{argmax}\, Q(a) + \sqrt{\frac{2log(t)}{N_t(a)}}$
  * UCB achievbes logarithmic asymptotic total regret: $\lim_{t\to\infty} L_t \leq 8log(t) \sum_{a|\Delta_a>0} \Delta_a$
- Bayesian Bandits

    * Bayersian bandits exploit prior knowledge of rewards $p[\mathscr{R}]$
    * computes posterior distribution of rewards $p[\mathscr{R}|h_t]$, where $h_t = a_1, r_1, ..., a_{t-1}, r_{t-1}$ is the history.
    * Better performance if prior knowledge is accurate.
    * e.g. assume reward distribution is Gaussian: $\mathscr{R}_a(r) = \mathscr{N}(r; \mu_a, \sigma_a^2)$
    * probability matching: select action a according to probability that a is optimal: $\pi(a|h_t) = \mathbb{P}[Q(a) > Q(a'), \forall a' \neq a|h_t]$
    * probability matching is optimistic in the face of uncertainty. They have a higher probability of being max.
    * Thompson sampling: $\pi(a|h_t) = \mathbb{P}[Q(a) > Q(a'), \forall a' \neq a|h_t] = \mathbb{E}_{\mathscr{R}|h_t}[\nVdash(a = \underset{a \in A}{argmax}\ Q(a))]$
    * Bayes law to compute $p[\mathscr{R}|h_t]$, sample a reward distribution from posterior, compute action-value function $Q(a) = \mathbb{E}[\mathscr{R}_a]$
    * then selection action maximising value on sample: $a_t = \underset{a \in A}{argmax}\ Q(a)$
  - Information State Search
    * Quantify the value of information. Gain is higher in uncertain situation.
    * bandits as sequential decision-making problems.
    * At each step there is an information state $\tilde{s} = f(h_t)$ derived from the history.
    * Each action a causes information state transition $\tilde{\mathscr{P}}^a_{\tilde{s},\tilde{s}'}$
    * This defines MDP in augmented information state space $\tilde{\mathscr{M}} = \langle \tilde{\mathscr{S}}, \mathscr{A}, \tilde{\mathscr{P}}, \mathscr{R}, \gamma \rangle$
    * This MDP can be solved by reinforcement learning
- Contextual Bandits
  - A contextual bandit is a normal multi armed bandit with a State distribution S and $\mathscr{R}^a_s(r) = \mathbb{P}[r|s,a]$ the reward distribution.
  - Linear UCB
    * Estimate value function with a linear function approximator $Q_\theta(s,a) = \phi(s,a)^T \phi \approx Q(s,a)$
    * Estimate parameters by least squares: $A_t = \sum_{\tau=1}^t \phi(s_\tau, a_\tau)\phi(s_\tau, a_\tau)^T,\ b_t = \sum_{\tau=1}^t \phi(s_\tau, a_\tau) r_\tau,\ \theta_t = A_t^{-1} b_t$
    * Add on a UCB $U_\theta(s,a) = c\sigma$. UCB is standard deviations above the mean.
    * Select action maximising upper confidence bound $a_t = \underset{a \in A}{argmax}\ Q_\theta(s_t, a) + c\sqrt{\phi(s_t,a)^T A_t^{-1} \phi(s_t,a)}$
- MDPs
  - Optimistic Initialisation
    * Model-Free: Initialise action-value function Q(s,a) to $\frac{r_{max}}{1-\gamma}$. This enourages systematic exploration of states and actions.
    * Model-Based: Construct optimistic model of the MDP. Initialise transitions to r_{max}. This enourages systematic exploration of states and actions.
  - Optimisim in the Face of Uncertainty
    * Model-Free: Maximise UCB on action-value function: $Q^\pi(s,a) : a_t = \underset{a \in A}{argmax}\ Q(s_t, a) + U(s_t, a)$.
    * Estimate uncertainty in policy evaulation (easy), ignores uncertainty from policy improvement.

    * Maximise UCB on optimal action-value function: $Q^*(s,a) : a_t = \underset{a\in A}{argmax}\, Q(s_t,a) + U_1(s_t,a) + U_2(s_t,a)$.
    * Estimate uncertainty in policy evaulation (easy), plus uncertainty from policy improvement (hard).
    * Model-Based: Maintain posterior distribution over MDP and estimate transitions and rewards $p[\mathscr{P},\mathscr{R}|h_t]$
    * Use posterior to guide exploration.
  - Probability Matching
    * Use Thompson Sampling. But Sample an MDP $\mathscr{P},\mathscr{R}$ from posterior (not reward distribution.)
  - Information State Search
    * MDP uses augmented state $\langle s,\tilde{s}\rangle$. s is original MDP-state and $\tilde{s}$ is history statistic.
    * Each action causes a state transition and a information state transition.
    * MDP is know augmented to $\tilde{\mathscr{M}} = \langle \tilde{\mathscr{S}}, \mathscr{A}, \tilde{\mathscr{P}}, \mathscr{R}, \gamma \rangle$
    * Posterior distribution over MDP is information state: $\tilde{s}_t = \mathbb{P}[\mathscr{P},\mathscr{R}|h_t]$
    * Augmented MDP over $\langle s,\tilde{s}\rangle$ is called Bayes-adaptive MDP.

RL Course by DeepMind - Part 10: Classic Games

- Game Theory
  - Optimality in Games
    * we consider all the policies of all the players called the joint policy
    * What is the optimal policy π^i for the ith player?
    * If all players fix their policies π^{-i}, then best response is $\pi^i_*(\pi^{-i})$
    * Nash equilibrium is a joint policy for all players: $\pi^i = \pi^i_*(\pi^{-i})$ such that every player's policy is a best response.
  - Single-Agent and Self-Play Reinforcement Learning
    * Best response is solution to single-agent RL problem: Game is reduced to an MDP. The other players become part of the environment.
    * Nash equilibrium is fixed-point of self-play RL. Agents play against themselves and their policies.
    * Experience is generated by playing games between agents: $a_1\tilde{\pi}^1$, $a_2\tilde{\pi}^2$
    * Each agent learns best response to other players. One player's policy determines another player's environment.
    * All players are adapting to each other.
    * Not all RL methods converge on a fixpoint. If we ever reach a fixpoint that is an nash equilibrium.
    * There can be multiple nash-equilibria
  - Two-Player Zero-Sum Games
    * A two-player game has two (alternating) players.
    * A zero sum game has equal and opposite rewards for player 1 and player 2: $R^1 + R^2 = 0$
  - Perfect and Imperfect Information Games
    * perfect information or markov game is fully observed.

        * imperfect information game is partially observed.
- Minimax Search
    -
- Self-Play Temporal-Difference Learning
    -
- Combining Reinforcement Learning and Minimax Search
    -
- Reinforcement Learning in Imperfect-Information Games.
    - Players have different information states and therefore seperate search trees.
    - There is one node for each information state that summarises what a player knows.
    - Many real states may share the same information state.
    - Can be solvec by iterative forward-search methods (e.g. Cunterfactual regret minimization).
    - Or Self-play reinforcement Learning (e.g. Smooth UCT)
    - Smooth UCT Search:

8.3.4. Reinforcement Learning - An Introduction

Reinforcement Learning 17.4 Designing Reward Signals (p.491)

- designing reward signal is a critical so that the agent reaches the goal the designer actually desires.
- some problems involve goals that are difficult to translate into reward signals.
- reinforcement agents can discover unexpected ways to make their environment deliver reward.
- often it is found by trial-and-error search.
- If the learning is to slow the reward signal might be to sparse:
- sparse reward problem:
    - state-action pairs that trigger reward may be few and far between.
    - and reward for progress might not be able to detect. The Agent may wander aimlessly for a long time, sometimes called the "plateau problem"
    - tempting to adress this by rewarding subgoals that are important way stations for the goal.
    - This may lead the agent to behave differently and they might not achieve to overarching goal.
    - Bettwer way: augment value-function approximation with guesses of what it should be or parts of it schould be.
    - $v_0 : \mathscr{S} \to \mathbb{R}$ is our initial guess for optimal value function v_*. With linear features:
    - $\hat{v}(s,w) = w^T x(s) + v_0(s)$
    - This works for arbitrary nonlinear approximators and v0, though it might not accelerate learning.
    - effective approach: **shaping technique**:

  - shaping involves changing the ward signal as learning proceeds. It starts as not being sparse and graudaly modifying it to reward sparsly so that it suits the actual problem.
  - The agent faces a sequence of increasingly-difficult reinforcement learning problems.
  - Each State is not as hard as the previous one as some basic knowledge exists.
  - It has similarities to transfer learning.
- imitation learning
  - no idea what the wards should be, but there is another person or agent whose behavior can be observed.
  - Benefit from the expert agent, but leave open the possibility of eventually performing better.
  - you can either drectly use the experts behavior (supervised learning) or by extracting reward signals with "inverse reinforcement learning".
  - inverse RL tries to recover the expert's reward signal from the expert's behavior alone.
  - This cannot be 100 % accurate.
  - This needs strong assumptions about environment dynamics or feature vectors in which the reward is linear.
- automate trial-and-error
  - reward signal is a perameter of the learning algorithm.
  - define a space of feasible candidates and apllying an optimization algorithn.
  - it runs a new agent for a few steps and scores the overall result against a high-level objective function which has the deisgners true goal.
  - this can be improved with online gradient ascent. The gradient comes from the high-level function
- given that the agent has restraints like computational power or partial observability the agents actual goal might differ from the designer's goal.
- the performance comparison against the high-level function is very sensitive to details in the reward signal in subtle ways.

8.3.5. Medium Blog Post

Medium Blog Post

8.3.6. RL - Base - DQN - Human-level control through deep reinforcement - 2015

RL - Base - DQN - Human-level control through deep reinforcement - 2015

8.3.7. RL - Base - A3C - Asynchronous Methods for Deep Reinforcement Learning - 2016

RL - Base - A3C - Asynchronous Methods for Deep Reinforcement Learning - 2016

8.3.8. RL - Sur - State-of-the-art Reinforcement Learning Algorithms - 2020

RL - Sur - State-of-the-art Reinforcement Learning Algorithms - 2020

8.4. Deep Learning

8.4.1. Probabilistic Deep Learning Book

Probabilistic Deep Learning Book

8.4.2. Deep Learning Book

Deep Learning Book
Chapter 11: Practical Methodology

- Determine your goals: which error metric, your target value for this error metric.
- Establish a working end-to-end pipeline as soon as possible, including the estimation of the appropriate performance metrics.
- Instrument the system well to detmin bottlenecks in performance. Which component is performing bad? Is it due to overfitting, underfitting or a software defect?
- Repeatedly make incremental changes such as gathering new data, adjusting Hyperparameters, or changing algorithms based on findings of your instrumentation.
- Performance Metrics:
  - Usually Impossible to achieve zero error. Bayes errore defines the minimum error rate.
  - May not be able to gather more data.
  - Either from previous results or the real-world problem you can infer the minimum error rate you need.
  - precision: fraction of detections reported by the model that were correct.
  - recall: fraction of true events that were detected.
  - You can plot a PR curve with precision on the y-axis and recall on the x-axis.
  - You choose to report a result whenever its score exceeds some threshhold. By varying it you can trade precision for recall.

  - Combine them into an F-Score $F = \frac{2pr}{p+r}$. To have a performance of the classifier in a single number.
  - You can also look at the total area lying benath the PR curve.
  - Some systems do not give a response if they are not confidence enought. Here coverage is a good metric. It is the fraction of examples for which the machine system is able to produce a response.
  - Important: which performance metric to improve ahead of time and focus on that.
- Default Baseline Models:
  - Which base method to use when you start out as a baseline:
  - reasonable optimization Algorithm: SGD + Momentum and decaying learning rate.
- Get more Data?
  - Is the performance of the training set acceptable? Is it more it still needs to learn more.
  - You can then increase the size of the model by adding more layers or more hidden units to each layer.
  - Or try improving the learning altorithm. Improve the hyperparameters.
  - If fine tuned algorithms do not work well, the problem might lie in the data itself.
  - Testset performance poor? more data!
  - Create curves showing relationship between training set size and generalization error. So you can predict how much data to add
- Selecting Hyperparameters:
  - Manual:
    * You need to understand the relationship between hyperparameters, training error, generalization error and computational ressources.
    * Goal: lower generalization error subject to runtime and memory budget.
  - Automatic:
    *
  - Grid Search:
    *
  - Random Search:
    * First define a marginal distribution for each hyperparameter (Bernoulli/-Multinoulli for binary/discrete params, uniform distribution on log-scale for positive real-value hyperparams).
    * We may often want to run repeated versions of random search, to refine the search based on the results of the first run.
    *
  - Model-Based Hyperparameter Optimization
    * Simple Settings: feasable to compute the gradient of some differntiable error measure.
    * Can build a model of the validation set error. And propose new guesses by optimization within this model.
    * Most models use a Bayesian regresision model.
    * Optimization is a tradeoff of exploration and exploitation.
    * Drawback: Needs an entire run or epoch to see if the parameter where wrong. A human doing this manually can see this earlier.

- Debugging Strategies
  - A problem: If one part of the model is proken, the other parts can adapt and still get acceptable performance.
  - Two kinds of strategies: Design a case that is so simple that the correct behavior can be predicted or we design a test that exercises one part of the Neural net implrementation in isolation.
  - Visualize the model in action: view the detections and results of your network.
  - Visualize the worst mistakes: View the data that has the lowest confidence which might give you insight into if the data has been processed or labled.
  - Reason about software using training and test-error: test-error might be calculated increicctly.
  - Fit a tiny dataset: Even small models can fit tiny dataset. If it can't there might be a software defect.
  - Compare back-propagated derivatives to numerical derivatives: Maybe your back-propagation is wrong.
  - Monitor histograms of activations and gradient: The preactivation value of hidden units can tell us if the units saturate or how often they do. Are some units always off?

8.5. Deep Reinforcement Learning

8.5.1. Deep RL Bootcamp

Deep RL Bootcamp
Lecture 1: Intro to MDP and Exact Solution Methods
Covers Value Iteration and Policy Iteration. Also covered by David Silver (Part 3)
Lecture 2: Sample-based Approximations and Fitted Learning
Covers Tabular Learning Methods. Also covered by David Silver ()
Lecture 3: DQN and Variants
Covers Experience Replay (David Silver Part 6), Fixed Q-Targets (Freecodecamp Part 4)

- Experience Replay gives you the most benefit of the Q-Learning Stability techniques.
- But using both Experience Replay and Fixed Q-Targets ist optimal.
- DQN uses Huber loss for Bellman error: $L_\delta(a) = \begin{cases} \frac{1}{2}a^2 & for|a| \leq \delta, \\ \delta(|a| - \frac{1}{2}\delta), & otherwise \end{cases}$
- It helps to anneal the exporation rate. it starts at 1 and decreases.
- Neural Fitted Q Iteration (Riedmiller, 2005):
  - Trains neural networks with Q-learning.
  - Alternates between collcecting new data and fitting a new Q-function to all previous experience with batch gradient descent.
  - DQN is an online Variant of Neural Fitted Q Iteration.

- Lin's Networks (Long-Ji Lin, 1993):
  - Introduced experience replay.
  - This network does nit share parameters among other actions. Each actions has different parameters.
- Double DQN
  - There is an bias in DQN.
  - Double DQN maintains two sets of weights θ and θ'.
  - θ is for selection the best action, θ' is for evaluating the best action.
  - Loss: $L_i(\theta_i) = \mathbb{E}_{s,a,s',r}(r + \gamma Q(s', arg\ \underset{a'}{max}\ Q(s', a'; \theta); \theta_i') - Q(s, a; \theta_i))^2$
  - Usually Better.
- Prioritised Experience Replay
  - Replay transitions in proportion to absoulte Bellman error: $|r + \gamma \underset{a'}{max}\ Q(s', a'; \theta') - Q(s, a; \theta)|$
  - Leads to much faster learning as supposed to replaying with equal probability.
- Dueling DQN
  - Value-Advantage decomposition of Q $Q^\pi(s,a) = V^\pi(s) + A^\pi(s,a)$
  - Dueleing DQN: $Q(s,a) = V(s) + A(s,a) - \frac{1}{|\mathscr{A}|}\sum_{a=1}^{|\mathscr{A}|} A(s,a)$
  - The last Layer is not a single Q-Layer but two Layers. One for the Value Function and one for the Advantage Function.
  - "Dueling Network Architectures for Deep Reinforcement Learning", Wang et al 2016
- Noisy Nets for Exploration
  - Add noise to network parameters for better exploration
  - Standard linear layer $y = wx + b$
  - Noisy linear Layer $y = (\mu^w + \sigma^w \odot \varepsilon^w)x + \mu^b + sigma^b \odot \varepsilon^b$
  - $\varepsilon^w, \varepsilon^b$ contain noise.
  - σ^w, σ^b are learned parameters that determine the amount of noise.

Lecture 4a: Policy Gradients and Actor Critic

- Why Policy Optmization:
  - policy can be simpler than q or v.
  - value-function: does not prescribe actions. This might need a model of the dynamics.
  - action-value-function: needs to be able to efficiently solve and argmax over q: Challenge for continuous / high-dimensional action spaces.

Lecture 5: Natural Policy Gradients, TRPO, PPO

- this lecture: once you have your advantage estimate how do you update your policy with that?
- Limitations of Vanilla Policy Gradient Methods:

- Hard to choose stepsizes: Input data is nonstationary due to changing policy => observation and reward distribution change.
- Can partially be adressed with normalization techniques.
- Also: Bad step is damaging since it affects visitation distribution. Too Far => Bad Policy => Can't recover!
- Sample Efficiency: Only one gradient step per environment sample. Depent on scaling of coordinates.

- What Loss to optimize
  - Policy gradients $\hat{g} = \hat{\mathbb{E}}_t[\nabla_\theta log\ \pi_\theta(a_t|s_t)\hat{A}_t]$ hat = empirical!
  - Loss: $L^{PG}(\theta) = \hat{\mathbb{E}}_t[log\ \pi_\theta(a_t|s_t)\hat{A}_t]$.
  - This loss should not be optimized too far (to a solution), as if you just take this loss it might diverge to infinity.
  - Noisy estimate => radical changes in policy.
  - Another Version $L^{IS}_{\theta_{old}}(\theta) = \hat{\mathbb{E}}_t[\frac{\pi_\theta(a_t,s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t]$
  - at $\theta = \theta_{old}$, state-actions are samples using the old one.
  - We get the same gradient. In practice L^{IS} is not much different than the logprob version, for reasonably small policy changes.
- Trust Region Policy Optimisation
  - Idea: A function I want to optimise and some local approximation of that function which is only accurate locally. We have a trust region where we trust our local approximator.
  - Function to optimise: $\underset{\theta}{maximize}\ \hat{\mathbb{E}}_t[\frac{\pi_\theta(a_t,|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t]$
  - If you differentiate this you get the policy gradient!
  - with a constraint: we are not too far from the starting point of that approximation.
  - Example: Euclidian Distance between starting point θ_{old} and final point θ is small.
  - Better: KL-Divergence: $\hat{\mathbb{E}}_t[KL[\pi_{\theta_{old}}(\cdot|s_t), \pi_\theta(\cdot|s_t)]] \leq \delta$
  - We could penalize the constraint problem by $\underset{\theta}{maximize}\ Function\ to\ optimize - \beta \cdot KL - Divergence$
  - Method of Lagrange multipliers: optimality point of delta-constrained problem is also an optimality point of beta-penalized problem for some beta.
  - Practice: delta is easier to tune.
- Monotonic Improvement Result: If you look at the KL penalized objective. Then in theory if we use max KL instead of mean KL in penalty, then we get a lower (= pessimistic) bound on policy performance. This is the theory for the result.
- Maximize the max over KL in the state-space => we are guaranteed to improve the policy with the penalized objective and a lower bound.
- objective: $\underset{\theta}{maximize}\ \sum_{n=1}^{N} \frac{\pi_\theta(a_n|s_n)}{\pi_{\theta_{old}}(a_n|s_n)}\hat{A}_n$ constraint: $\bar{KL}_{\pi_{\theta_{old}}}(\phi_\theta) \leq \delta$. $\hat{A}_n$ is an estimation.
- TRPO can sole constrained optimization problem efficiently by using conjugate gradient. Related to natural policy gradients and natural actor critic
- Solving KL Penalized Problem:
  - The Penalized Version is calculated using a linear approximation of the objective and a quadratic approximation of the KL term
  - $\underset{\theta}{maximize}\ g \cdot (\theta - \theta_{old}) - \frac{\beta}{2}(\theta - \theta_{old})^T F(\theta - \theta_{old})$

- where $g = \frac{\partial}{\partial\theta} L_{\pi_{\theta_{old}}}(\pi_\theta)|_{\theta=\theta_{old}}$, $F = \frac{\partial^2}{\partial^2\theta} \bar{KL}_{\pi_{\theta_{old}}}(\pi_\theta)|_{\theta=\theta_{old}}$
  - Solution: $\theta - \theta_{old} = \frac{1}{\beta} F^{-1} g$, F is the Fisher Information matrix, g is policy gradient. This is called the natural policy gradient.
  - Use Conjugate Gradient Algorithm to approximately solve: Fx = g.
- Solving KL Constrained Problem:
  - TO-DO!
- For PPO use the penality version. Do a SGD on above objective for some number of epochs.
- If KL too high, then increase beta. If KL is too low, decrease beta
- Review:
  - suggested optimizing surrogate loss L^{PG} or L^{IS}
  - suggested using KL to constrain size of update.
  - Correspondes to natural gradient step $F^{-1}g$ under linear quadratic approximation.
  - Can solve for this step approximately using conjugate gradient method.
- Linear-quadratic approximation + penalty => natural gradient.
- No contraint => policy iteration.
- Eucleidean penalty instead of KL => vanilla policy gradient.
- Limitations of TRPO
  - Hard to use with architectures with multiple outputs, e.g. policy and value functions
  - Empirically performs poorly on tasks requiring deep CNNs and RNNs.
  - Conjugate Gradient makes implementation more complicated.
- KFAC
  - Make Approximation of Fischer Information Matrix, that exploits the structure of Neural Networks. Block Diagonal Approximation of the weight Matrix.
  - TO-DO
- ACKTR: Combine A2C with KFAC Natural Gradient.
  - Combined with A2C, gives excellent on continuous control from images.
  - We already need the log pi for policy gradient, so it does not take extra computation.
  - Matrix inverses can be computed asynchronously.
  - Limitation: Straightforward for Feedforward nets and Convolutions. Less straightforward for RNNs with shared weights.
- Can use Clipping Objective for Proximal Policy Optimization. (TO-DO)
- This is a bit better than TRPO on continuous control. Compatible with multi-output networks and RNNs.

Lecture 6: Nuts and Bolts of Deep RL Experimentation

- Approaching New Problems:
  - New Algorithm:
    * Run experiments quickly.
    * Do Hyperparameter search.
    * Interpret and visualize learning process: state visitation, value function (fitting), state distribution etc.

    - Construct toy problems where your idea will be strongest and weakest, where you have a sense of what it should do
    - Counterpoint: don't overfit algorithm to contrived problem
    - Useful: medium-sized problems that you're intimately familiar with
  - new Task: Provide good input features, shape reward function
  - POMDP Design
    - Visualize random policy: does it sometimes exhibit desired behavior?
    - Plot time series for observations and rewards. Are they on a reasonable scale?
    - Histogram observations and rewards.
  - Run Your Baseline: Don't expect them to work with default parameters.
  - Recommended: Cross entropy method (Learning Tetris using the noisy cross-entropy method), Well-tuned policy gradient or Q-learning methods, alternative.
  - Early in tuning process you may need a huge number of samples. It might need a "burn-in" period.
  - Don't get deterred if you cannot published work to run directly.
- Ongoing Development and Tuning:
  - Explore sensitivity to each parameter: If too sensitive, it doesn't really work you just got lucky.
  - Look for health indicators: VF fit quality, policy entropy, Update size in output space and parameter space, Standard diagnostics for deep networks.
  - If reusing code, regressions occur => run a battery of benchmarks occasionally.
  - Always use different random seeds so that your algorithm is not influenced by one seed alone.
  - Always try to simplify the algirithm. Different tricks basically do a similar thing (like in whitening).
  - Favor simplicity in algorithm which normally leads to generalization.
  - Automate your experiments!
- General Tuning Strategies for RL:
  - Whitening:
    - If observations have unknown range, standardize them
    - compute running estimate of mean and standard deviation.
    - $x' = clip((x-\mu)/\sigma, -10, 10)$
    - Rescale the rewards, but don't shift mean. That affects the agent.
    - Standardize prediction targets (e.g. value function) the same way.
  - Generally important parameters:
    - Discount: gamma = 0.99 => ignore rewards delayed by more than 100 timesteps.
    - Low gamma works well for well-shaped reward.
    - with TD(lambda) methods, can get away with high gamma when lambda < 1
    - Continuous timesteps (like games) are usually discretisized to time steps (like frameskips). Can the action frequency the agent has actually solve the problem?
    - Also: look the random exploration: If you do the same action multiple times in a row you tend to explore further. Choose an interesting diskretisation
    - Look at min/max/stdev of episode returns, along with means.

        * Look at episode lengths: sometimes useful information: Solving problem faster, losing game slower
- Policy Gradient Strategies:
    - Premature drop in policy entropy => policy gets deterministic => no exploring => no learning (drops eventually but not super fast).
    - Alleviate by using entropy bonus or KL penalty.
    - KL $KL[\pi_{old}(\cdot|s), \pi_{old}(\cdot|s)]$
    - How to measure entropy: discrete: analytically, continuous policy: gaussian policy => compute differential entropy.
    - Action-Space Entropy is what we are talking about. State-Space Entropy is to hard to calculate on anything nontrivial.
    - KL = 0.01: Small update, 10: big update
    - KL spike => drastic loss of performance.
    - No learning progress might mean steps are too large
    - explained variance = $\frac{1-Var[empirical\ return-predicted\ value]}{Var[empirical\ return]}$
    - Policy Initialization: Determines initial state visitation.
    - Zero or tiny final layer, to maximize entropy
- Q-Learning Strategies:
    - Optimize memory usage carefully: you'll need it for replay buffer.
    - Learning rate schedules.
    - Exploration schedules.
    - DQN converges slowly
- Miscellaneous Advice:
    - Don't get stuck on problems. Some algorithms do really well on some problems but bad on another.
    - Techniques from supervised learning don't necessrily work in RL: batch norm, dropout, big networks.

8.6. Multi-Agent Systems

8.6.1. !MAS - An Introduction to Multi-Agent Systems - 2010

MAS - An Introduction to Multi-Agent Systems - 2010
Benefits of using MAS in large systems

- Increase in the speed and efficiency of the operation due to parallel computation and asynchronous operation.
- Graceful degradation whone one or more of the agent fail, thus increasing realibility and robustness of the system.
- Scalability and flexibility - Agents can be added as and when necessary.
- Cost Reduction: Individual agents cost much less than a centralized architecture

- Reusability: Agents with a modular structure can be easily replaced in other systems or be upgraded more easily than a monolithic sysetm.

Challenges of using MAS in large systems

- environment: An agents action modify its own environment but also that of its neighbours. therefore they need to predict the action of the other agents so that they can reach a goal. This can be an unstable system. Environment dynamic: Is the effect caused by other agents or by the variation in the environment?
- perception: limited sensing range => each agent only has partial observability for the environment. Therefore the decisions reached might be sub-optimal.
- Abstraction: ???
- conflict resolution: lack of global view => conflict. therefore information on constraints, action preferences and goal prioritoes must be shared between agents. When to communicate what to which agent?
- Inference: Single-Agent: State-Action-Space can be mapped with trial and error. Multi-agent: each agent may or may not interact with each other. If they are heterogenous, they might even compete and have different goals. You need a fitting inference machanism

Classification: Internal Architecture

- homogeneous: all agents have the same internal architecture (Local Goals, Sensor Capabilities, Internal states, Inference Mechanism and Possible Actions). In a typical distributed environment, overlap of sensory inputs is rarely present
- Heterogeneous: agents may differ in ability, structure and functionality. Because of the dynamics and location the actions chosen might differ between agents. their local goals may contradict the objective of other agents.

Classification: Agent Organization

- hierarchical: typical: tree-structure. At different heights, different levels of autonomy. data from lower levels flow upwards. Control signal flows from high to low in the hierarchy.
  - simple: the decision making authority is a single agent of highest level. BUT: single point of Failure
  - uniform: authority is distributed among the various agents, for better efficiency, fault tolerance, graceful degradation. Decisions made by agent with appropriate information. (MAS - TrafficControl - Neural Networks for Continuous Online Learning and Control - 2006)
- holonic: fractal structure of several holons. Self-repeating. Used for large organizational behaviours in manufacturing and business.
  - An agent that appears as a single entity might be composed of many sub-agents. They are not predetermined, but form through commitments.
  - Each holon has a head agent that communicates with the environment or with other agents in the environment. It is selected either randomly, through a rotation policy, or selected by resource availability, communicaton capability.

  - Holons can be nested to form Superholons.
  - compare to tree: in Holons cross tree interactions and overlapping of holons is allowed.
  - pro: abstraction good degree of freedom, good agent autonomy.
  - contra: abstraction makes it difficult for other agents to predict the resulting actions of the holon.
- coalitions: group of agents come together for a short time to increase utility or performance of the individual agents in a group. they cease to exist when the performance goal is achieved.
  - coalition may have either a flat or a hierarchical architecture.
  - It may have an leading agent to act as a representative.
  - overlap is allowed. this increased complexity of computation of the negotiation strategy.
  - You can have one coaltion with all agents => maximum performance of system. Impractical due to restraints on communication and resources.
  - minimize amount of colations: because of the cost of creating and dissolving a colation group.
- teams: agents work together to increase the overall performance of the group, rather than working as individual agents.
  - their interactions can be arbitrary and the goals and roals can vary with the performance of the group.
  - large team size is not beneficial under all conditions. some compromises must be made.
  - large teams offer a better visibility of the environment. but is slower computation wise. Learning-Performance Tradeoff.
  - computation cost usually much greater than coalitions.

Classification: Communication

- local communication: agents directly communicate similar to message passing. there is no place to store information. creates distributed architecture. used in: (25),(37),(38).
- blackboards: a group of agents share a data repository which is provided for efficient storage.
  - can hold design data and control knowledge, accessable by the agents.
  - control shell: notfies the agent when relevant data is available.
  - single point of failure.
- agent communication language (ACL): common framework for interaction and information sharing. (40).
  - procedural approach: modelled as a sharing of the precedural directives. Shared how an agent does a specific task or the entire working of the agent itself. Script Languages often used. Disadvantage: necessitiy of providing information on the recipient agent, which is in most cases partially known. Also how to merge the scripts into one executable. Not preferred method.
  - declarative approach: sharing of statements for definitions, assumptions assertions, axioms etc. Short declarative statements as length increases probability of information corruption. Example: ARPA knowledge sharing effort.

  - Best known inner languages: Knowledge Interchange Format. Information exchange is implicitly embedded in KIF. But the package size grows with the increase in embedded information. Solution: High-level Languages like KQML (Knowledge QUery and Manipulation Language)

<u>Classification: Decision making in Multi-Agent Systems</u>

- undercainty: effects of a specific actions on the environment and dynamics because of the other agents.
- Methodology to try and find a joint action or equilibrium point which maximizes the reward of every agent.
- Typically modelled with game theory method. Strategic games:
  - a set of players (agents)
  - Foreach player, there is a set of actions
  - Foreach player, the prefeernces over a set of actions profiles
  - payoff with the combination of action, a joint-action, that is assumed to be predefined.
  - all actions are observable forall agents.
  - make the assumption that all participating agents are rational.
- Nash equilibrium: for a payoff matrix: An action profile (joint-action), where no player can do better by choosing one of the actions differently, given that the other player chose a specific action.
- there might be multiple nash equilibrium, so that there is no dominant solution. Here the coordination of MAS is needed to find a solution.
- Iterated Elimination Method: Strongly dominated actions are iteratively eliminated. This fails if there are no strictly dominated actions available.

<u>Classification: Coordination</u>

- agents work in parallel, therefore they need to be coordinated or synchronize the actions to ensure stability of the system.
- other reasons: prevent chaos, meet global constraints, utilize distributed resources, prevent conflicts, improve efficiency.
- achievable with constraints on the joint actions or by using informatil collated from neighbouring agents. Used to find the equilibrium action.
- payoff matrix necessary might be difficult to determine. It increases expenentially in the number of agents and action choices.
- dividing the game into subgames: roles (permitted actions is reduced, good for distributed coordination or centralized coordination)
- Coordination via Protocol.
  - negotioation to arrive an approdiate solutions.
  - Agents assume the role of manager (divide the problem) and contractor (who deals with the subproblems).
  - The manager and contractor are working in a bidding system.
  - Example: FIPA model

  - disadvantage: assumption of the existence of an cooperative agent. It is very communication intensive
- Coordination via Graphs: Problem is subdivided into easer problems. Assume the payoffs can be linear combinated from the local payoffs of the sub-games. THen just eliminate agents to find the optimal joint.
- Can also use belif models. Internal models of an agent on how he believes the environment works (needs to differentiate between environment and effects of other agents).

Classification: Learning

- active learning: analysing the observations to creat a belief or internal model of the corresponding situated agent's environment.
  - can be performed by using a deductive, inductive or probabilistic reasoning approach.
  - deductive: inference to explain an instance or state-action sequence using his knowledge. It is deduced or inferred from the original knowledge it is nothing new. It could form new parts of the knowledge base. uncertainty is usually disregarded (not good for real-time)
  - inductive: learning from observations of state-action pair. Good when environment can be presented in terms of some generalized statements. they use the correlation between observations and the action space.
  - probabilistic: assumption: knowledge base or belief model can be represented as probabilities of occurrence of events. observations of the environment is used to predict the internal state of the agent. Good example: Bayesian learning. Difficult for MAS, as the joint probability scales poorly in the number of agents.
- reactive learning: updating belief without having the actual knowledge of what needs to be learnt.
  - useful when the underlying model of the agent or the environment is not known clearly and are black boxes.
  - can be ssen in agents which utilize connections systems such as NN.
  - can use reactive multi-agent feed forward neural networks.
  - they depend on the application domain and are therefore rarely employed in real world scenarios.
- learning based on consequences:
  - learning methods based on evaluation of the goodness of selected action. like in reinforcement learning.
  - programming the agents using reward and punishment scalar signals without specifying how the task is to be achieved.
  - learnt through trial and error and interaction with the environment.
  - usually used when action space is small and descret. Recent developments allow them to work in continious and large state-action space scenarios.
  - An agent is usally represented as a Markov Decision Process.
  - Expectaation operator optmal policy is the argmax of the Q-value, which uses the bellman equation. Bellman equation is solved iteratively.
  - The solution is referred to as q-learning method.

- For MAS the reinforcement learning method has the problem of combinatorial explosion in the state-action pairs.
- The information must be passed between the agents for effective learning.

8.6.2. !Artificial Intelligence - A modern Approach

Multiagent Planning (p.425)

- each agent tries to achieve is own goals with the help or hindrance of others
- wide degree of problems with various degrees of **decomposition of the monolithic agent**.
- multiple concurrent effectors => **multieffector planning** (like type and speaking at the same time).
- effectors are physically decoupled => **multibody planning**.
- if relevant sensor information foreach body can be pooled centrally or in each body like single-agent problem.
- When communication constraint does not allow that: **decentralized planning problem**. planning phase is centralized, but execution phase is at least partially decoupled.
- single entity is doing the planning: one goal, that every body shares.
- When bodies do their own planning, they may share identical goals.
- **multibody**: centralized planning and execution send to each.
- **multiagent**: decentralized local planning, with coordination needed so they do not do the same thing.
- Usage of **incentives** (like salaries) so that goals of the central-planner and the individual align.

Multiple simultaneous actions

- **correct plan**: if executed by the actors, achieves the goal. Though multiagent might not agree to execute any particular plan.
- **joint action**: An Action for each actor defined => joint planning problem with branching factor bn̂ (b = number of choices).
- if the actors are **loosely coupled** you can describe the system so that the problem complexity only scales linearly.
- standard approach: pretend the problems are completely decoupled and then fix up the interactions.
- **concurrent action list**: which actions must or most not be executed concurrently. (only one at a time)

Multiple agents: cooperation and coordination

- each agent makes its own plan. Assume goals and knowledge base are shared.
- They **might choose different plans** and therefore collectively not achieve the common goal.

- **convention**: A constraint on the selection of joint plans. (cars: do not collide is achieved by "stay on the right side of the road").
- widespread conventions: social laws.
- absence of convention: use communication to achieve common knowledge of a feasible joint plan.
- The agents can try to **recognize the plan other agents want to execute** and therefore use plan recognition to find the correct plan. This only works if it is unambiguously.
- an **ant** chooses its role according to the local conditions it observes.
- ants have a convention on the importance of roles.
- ants have some learning mechanism: a colony learns to make more successful and prudent actions over the course of its decades-long life, even though individual ants live only about a year.
- Another Example: **Boid**
- If all the boids execute their policies, the flock inhibits the emergent behavior of flying as a pseudorigid body with roughly constant density that does not disperse over time.
- **most difficult multiagent** problems involve both cooperation with members of one's own team and competition against members of opposing teams, all without centralized control.

8.6.3. MAS - App - Neural Networks for Continuous Online Learning and Control - 2006

MAS - App - Neural Networks for Continuous Online Learning and Control - 2006

8.6.4. MAS - Base - The Multiagent Planning Problem - 2016

MAS - Base - The Multiagent Planning Problem - 2016

8.6.5. MAS - Con - Distributed Cooperative Control and Communication for Multi-agent Systems - 2021

MAS - Con - Distributed Cooperative Control and Communication for Multi-agent Systems - 2021

8.6.6. MAS - Con - PRIMA 2020 Principles and Practice of Multi-Agent Systems - 2021

MAS - Con - PRIMA 2020 Principles and Practice of Multi-Agent Systems - 2021

8.6.7. MAS - Con - Swarm Intelligence - 2010

MAS - Con - Swarm Intelligence - 2010

8.6.8. MAS - Con - Swarm Intelligence - 2012

MAS - Con - Swarm Intelligence - 2012

8.6.9. MAS - Con - Swarm Intelligence - 2014

MAS - Con - Swarm Intelligence - 2014

8.6.10. MAS - Con - Swarm Intelligence - 2016

MAS - Con - Swarm Intelligence - 2016

8.6.11. MAS - Con - Swarm Intelligence - 2018

MAS - Con - Swarm Intelligence - 2018

8.6.12. MAS - Con - Swarm Intelligence - 2020

MAS - Con - Swarm Intelligence - 2020

8.6.13. MAS - Evo - Co-evolutionary Multi-agent System with Predator-Prey Mechanism for Multi-objective Optimization - 2007

MAS - Evo - Co-evolutionary Multi-agent System with Predator-Prey Mechanism for Multi-objective Optimization - 2007

8.6.14. MAS - Het - Multiagent Systems A Survey from a Machine Learning Perspective - 2000

MAS - Het - Multiagent Systems A Survey from a Machine Learning Perspective - 2000

8.6.15. MAS - Hie - Hierarchical Control in a Multiagent System - 2007

MAS - Hie - Hierarchical Control in a Multiagent System - 2007

8.6.16. MAS - Hie - Holonic - A Taxonomy of Autonomy in Multiagent Organisation - 2003

MAS - Hie - Holonic - A Taxonomy of Autonomy in Multiagent Organisation - 2003

8.6.17. MAS - Multi-Agent Systems - A Survey - 2018

MAS - Multi-Agent Systems - A Survey - 2018

8.6.18. MAS - Sur - A survey of multi-agent organizational paradigms - 2004

MAS - Sur - A survey of multi-agent organizational paradigms - 2004

8.6.19. MAS - Tra - Transfer Learning for Multi-agent Coordination - 2011

MAS - Tra - Transfer Learning for Multi-agent Coordination - 2011

8.6.20. MAS - Tra - Transfer learning in multi-agent systems through paralllel transfer - 2013

MAS - Tra - Transfer learning in multi-agent systems through paralllel transfer - 2013

8.7. Game Theory

8.7.1. Artificial Intelligence - A modern Approach

Game Theory (p.666)

- perfect information = fully observable, imperfect information = partially observable.
- single-move games:
  - only one action can be chosen per game.
  - **Payoff Matrix**: For 2 players it shows what action-tuple gives which reward. Also known as strategic/normal form.
  - strategy = policy, pure strategy = deterministic policy, mixed strategy = probabilistic policy.
  - **strategy profile**: assignment of strategy to each player. the game's aoutcome is then a numeric value for each player.
  - **solution**: each player adopts a rational strategy.
  - **dominant strategy**: a policy that is always the best.
  - s **strongly dominates** s' for player p if the outcome of s is always better then s' for any strategy of any other player.
  - s **weakly dominates** s' for player p if the outcome of s is atleast once better then s' and as good as s' for any strategy of any other player.
  - Outcome is **pareto optimal** if there is no other outcome that all players would prefer.
  - Outcome is **pareto dominated** by another outcome if all players would prefer the other outcome.
  - Every player has a dominant strategy, then all the strategies are called an **dominant strategy equilibrium**. Which is generally formed if no player can benefit by switching strategies.
  - equilibrium = local optimum. Every game has at least one equilibrium an Nash equilibrium.
  - dominant strategy equilibrium is a nash equilibrium. Some games have nash equilibria but no dominant strategies.
  - repeated game: multiple runs.
  - multiple acceptable solutions: use the unique pareto-optimal nash equilibrium if it exists. Every game has at least one.
  - coordination game: communication so players can negotiate before a game the soltuons they take to be mutually beneficial.
  - zero-sum game: the sum of the payoffs for all players is zero or a constant. To solve these take one player as the maximizer => maximin technique.
  - zero-sum games have maximin nash equilibria. Every zero-sum game has a maximin equilbrium when you allow mixed strategies.
  - approach for finding equilibria in non-zero-sum games:
    - 1. Enumerate all possible subset of actions that might form mixed strategies. This is exponential in the number of action.

    * 2. Foreach strategy enumerated in 1. check to see if it is an equilibrium.

- repeated games
  - repeated game: players face the same choices repeatedly, but each tim ewith knowledge of the history ofa ll player's previous choices.
  - So the game is played multiple round. But the last one has nothing to influence so it used the dominant strategy for single-move game. Then the second to last has nothing to influence => induction => just play the same as single-move => not optimal.
  - Use a chance so the players do not know when the game will end.
  - if the agents cannot store the entire history they do not know when the game will end.
- sequential games
  - sequence of turns that need not be all the same. Represented by a game tree called the extensive form.
  - non-deterministic actions can be created by having the player's action be deterministic and then another action that is randomly chosen.
  - There could be a chance player that acts randomly to introduce distributions.
  - perfect recall: each player remembers all their own previous actions.
  - extensive form allows to always find solutions beause it represents the belief states of all players at once. Which is important if you strategy depends on the other players strategies.
  - extensive games can be converted to a normal-form game to solve it. By having using all possible state history combinations for the other player in the payoff matrix (does not scale well). This can usally be solved with linear programming.
  - alpha-beta pruning works good for large game trees but does not work well for imperfect information.
  - Alternative: sequence form: Is linear in the soze of the tree. it repsresnts not strategies in a node but paths through the tree which scales in the amount of possible endstates.
  - use feature spaces as abstractions of a game to create a smaller tree.
  - just using the equilibrium strategy gives you the perfect solution if the other players also use the equilibrium strategy. If the other player makes mistake you need to capitalize on that.
- game theory does not deal with with continuous states and actions.
- cournot competition is an extension that can halb in the contiuous space.
- game theory assumes the game is known. if the actions are not known beforehand or the other players are not fully rational.
- This can be olved with a Bayes-Nash equilibrium that expresses a player's belief about the other player's likely strategies.

Stochastic Games (p.177) This is more about game trees and alpha-beta pruning so technically not applicable for my work. Partial Observable Games (p.180) This is more about game trees and alpha-beta pruning so technically not applicable for my work.

8.8. Multi-Agent Reinforcement Learning

8.8.1. MARL - A Comprehensive Survey of Multiagent Reinforcement Learning - 2008

MARL - A Comprehensive Survey of Multiagent Reinforcement Learning - 2008
Benefits

- can be parallelized.
- can use experience sharing via communication, or with a teacher-learner relationship.
- Failure of one agent can be covered by other agents.
- insertion of new agents => scaleable.
- MARL Complexity: Exponential in number of agents.
- exploration (new knowledge) - exploitation (current knowledge) - Tradeoff.
- They explore about the environment and other agents.
- need for coordination.

Application Domains

- simulation better than real-life (better scalability and robustness).
- Distributed Control: for controlling processes (for larger industry plants).
  - avenue for future work.
  - used for traffic, power or sensory networks.
  - could also be used for pendulum systems.
- Robotic Teams (Multirobot):
  - simulated 2D space.
  - navigation: Reach a goal with obstacles. Area sweeping (retrival of objects (also cooperative)).
  - pursuit: Capture a prey robot.
- Automated Trading: Exchange goods on electronic markets with negotiation and auctions.
- Resource Management: Cooperatie team to manage resources or as clients. (routing, load balancing).

Practicallity and Future works

- Scalability Problem: Q-functions do not scale well with the size of the state-action space.
  - Approximation needed: for discrete large state-action spaces, for continous states and discrete actions or continious state and action.
  - Heuristic in nature and only work in a narrow set of problems.
  - Could use theoretical results on single-agent approximate RL.
  - also could use discovery and exploitation of the decentralized, modular structure of the multiagent task.

- MARL without prior knowledge is very slow.
  - Need humans to teach the agent.
  - shaping: first simple task then scale them.
  - could use reflex behavior.
  - Knowledge about the task structure.
- Incomplete, uncertain state measurements could be handled with partiall observability techniques (Markov decision process).
- Multiagent Goals needs a stable learning process for the environment and an adaption for the dynamics of other agents.
- using game-theory-based analysis to apply to the dynamics of the environment.

8.8.2. MARL - AC - Networked Multi-Agent Reinforcement Learning in Continuous Spaces - 2018

MARL - AC - Networked Multi-Agent Reinforcement Learning in Continuous Spaces - 2018

8.8.3. MARL - AC - Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environment - 2017

MARL - AC - Multi-Agent Actor-Critic for Mixed Cooperative-Competitive Environment - 2017

8.8.4. MARL - AC - Actor-Attention-Critic for Multi-Agent Reinforcement Learning - 2019

MARL - AC - Actor-Attention-Critic for Multi-Agent Reinforcement Learning - 2019

8.8.5. MARL - Base - Multiagent Reinforcement Learning - Theoretical Framework and an Algorithm - 1998

MARL - Base - Multiagent Reinforcement Learning - Theoretical Framework and an Algorithm - 1998

8.8.6. MARL - Base - Deep Reinforcement Learning for Robot Swarms - 2019 - KIT

MARL - Base - Deep Reinforcement Learning for Robot Swarms - 2019 - KIT

8.8.7. MARL - Base - PettingZoo - Gym for Multi-Agent Reinforcement Learning - 2020

MARL - Base - PettingZoo - Gym for Multi-Agent Reinforcement Learning - 2020

8.8.8. MARL - Base - Multi-agent reinforcement learning weighting and partitioning - 1999

MARL - Base - Multi-agent reinforcement learning weighting and partitioning - 1999

8.8.9. MARL - Com - Learning to Communicate with Deep Multi-Agent Reinforcement Learning - 2016

MARL - Com - Learning to Communicate with Deep Multi-Agent Reinforcement Learning - 2016

8.8.10. MARL - Com - Coordinating multi-agent reinforcement learning with limited communication - 2013

MARL - Com - Coordinating multi-agent reinforcement learning with limited communication - 2013

8.8.11. MARL - Het - LIIR - Learning Individual Intrinsic Reward inMulti-Agent Reinforcement Learning - 2019

MARL - Het - LIIR - Learning Individual Intrinsic Reward inMulti-Agent Reinforcement Learning - 2019

8.8.12. MARL - Het - An approach to the pursuit problem on a heterogeneous multiagent system using reinforcement learning - 2002

MARL - Het - An approach to the pursuit problem on a heterogeneous multiagent system using reinforcement learning - 2002

8.8.13. MARL - Hie - Hierarchical multi-agent reinforcement learning - 2006

MARL - Hie - Hierarchical multi-agent reinforcement learning - 2006

8.8.14. MARL - MO - Reward shaping for knowledge-based multi-objective multi-agent reinforcement learning - 2017

MARL - MO - Reward shaping for knowledge-based multi-objective multi-agent reinforcement learning - 2017

8.8.15. MARL - Role - ROMA Multi-Agent Reinforcement Learning with Emergent Roles - 2020

MARL - Role - ROMA Multi-Agent Reinforcement Learning with Emergent Roles - 2020

8.8.16. MARL - Sca - GAMA - Graph Attention Multi-agent reinforcement learning algorithm for cooperation - 2020

MARL - Sca - GAMA - Graph Attention Multi-agent reinforcement learning algorithm for cooperation - 2020

8.8.17. MARL - Sca - Plan-based reward shaping for multi-agent reinforcement learning - 2016

MARL - Sca - Plan-based reward shaping for multi-agent reinforcement learning - 2016

8.8.18. MARL - Sca - Multi-Agent Reinforcement Learning Using Linear Fuzzy Model Applied to Cooperative Mobile Robots - 2018

MARL - Sca - Multi-Agent Reinforcement Learning Using Linear Fuzzy Model Applied to Cooperative Mobile Robots - 2018

8.8.19. MARL - Sca - Stabilising Experience Replay for Deep Multi-Agent Reinforcement Learning - 2017

MARL - Sca - Stabilising Experience Replay for Deep Multi-Agent Reinforcement Learning - 2017

8.8.20. MARL - Sca - Mean Field Multi-Agent Reinforcement Learning - 2018

MARL - Sca - Mean Field Multi-Agent Reinforcement Learning - 2018

8.8.21. MARL - Sca - A modular approach to multi-agent reinforcement learning - 2005

MARL - Sca - A modular approach to multi-agent reinforcement learning - 2005

8.8.22. MARL - Sur - Multi-Agent Reinforcement Learning - a critical survey - 2003

MARL - Sur - Multi-Agent Reinforcement Learning - a critical survey - 2003

8.8.23. MARL - Sur - Multi-Agent Reinforcement Learning A Selective Overview of Theories and Algorithms - 2021

MARL - Sur - Multi-Agent Reinforcement Learning A Selective Overview of Theories and Algorithms - 2021

8.8.24. MARL - Sur - Multi-Agent Reinforcement Learning A Report on Challenges and Approaches - 2018

MARL - Sur - Multi-Agent Reinforcement Learning A Report on Challenges and Approaches - 2018

8.8.25. MARL - Sur - A Review of Cooperative Multi-Agent Deep Reinforcement Learning - 2019

MARL - Sur - A Review of Cooperative Multi-Agent Deep Reinforcement Learning - 2019

8.8.26. MARL - Sur - A Survey on Transfer Learning for Multiagent Reinforcement Learning Systems - 2019

MARL - Sur - A Survey on Transfer Learning for Multiagent Reinforcement Learning Systems - 2019

8.8.27. MARL - Tra - Transfer Learning in Multi-agent ReinforcementLearning Domains - 2011

MARL - Tra - Transfer Learning in Multi-agent ReinforcementLearning Domains - 2011

8.8.28. MARL - Tra - Parallel Transfer Learning in Multi-Agent Systems What, when and how to transfer - 2019

MARL - Tra - Parallel Transfer Learning in Multi-Agent Systems What, when and how to transfer - 2019

8.8.29. MARL - Tra - Transfer among Agents An Efficient Multiagent Transfer Learning Framework - 2020

MARL - Tra - Transfer among Agents An Efficient Multiagent Transfer Learning Framework - 2020

8.8.30. MARL - Tra - Agents teaching agents a survey on inter-agent transfer learning - 2019

MARL - Tra - Agents teaching agents a survey on inter-agent transfer learning - 2019

8.9. Graph Neural Networks

8.9.1. !Theoretical Foundations of Graph Neural Networks - 2021

Theoretical Foundations of Graph Neural Networks - 2021

- Goal: Exact same results for two **isomorphic graphs** (graphs that are the same the nodes are just arranged differently).
- Nodes: $x_i \in \mathbb{R}^k$ (features of node), feature matrix $\mathbf{X} = (x_1, ...x_n)^T \in \mathbb{R}^{k \times n}$
- By stacking the nodes in the matrix you have already ordered them (result should not depend on this).
- Operations that change the nod order: permutation matrices. They ahve exactly one 1 in every row and column, and zeroes everywhere examples. Left Multiplied: permute the rows. $P_{(2,4,1,3)}$: The numbers indicate where the 1 in the row is.
- **Permutation Invariance**: f is permutation invariant iff: $\forall P \in Permutation : f(PX) = f(X)$. Example: Deep Sets Model $f(X) = \phi(\sum_{i \in V} \psi(x_i))$. This is true for the entire data-set.
- **Permutation equivariance**: for identification on the node level. Seek functions that don't change the node order. f is permutation equivariant iff: $\forall P \in Permutation : f(PX) = Pf(X)$.
- **equivariance**: each node's row is unchanged by f. So foreach node we could define: $\forall i : h_i = \psi(x_i)$., the latent features h-i. Stacking h yielded: $H = f(x)$. The functions are applied indipendently to each node.
- Stacking equivarent functions with an invariant tail: $f(X) = \phi(\bigoplus_{i \in V} \psi(x_i)). \bigoplus\ is\ permutationinvariant\ aggreg$
- **Learning on Graphs**:
  - Represent Edges with adjacency matrix A: $a_{ij} = \begin{cases} 1 & (i,j) \in E \\ 0 & otherwise \end{cases}$. Edge features could be added aswell. permutation x-variance still holds.
  - xvariance on graphs: To not change edges: permutate rows and columns. Permutate with PAP^T.
  - **Invariance**: $f(PX, PAP^T) = f(X, A)$ $(A = Edges,\ X = Nodes)$
  - **Exvariance**: $f(PX, PAP^T) = Pf(X, A)$ $(A = Edges,\ X = Nodes)$
  - Neighbourhoods: Node i, it's 1-hop neighbors are defined as: $\mathcal{N}_i = \{j : (i, j) \in E \vee (j, i) \in E\}$. (Non-directed edges, node i is in it's own neighbourhood).
  - Multiset of features in the neighbourhood: $X_{\mathcal{N}_i} = \{\{x_j : j \in \mathcal{N}_i\}\}$. With a local function g as operating over this multiset: $g(x_i, X_{N_i})$

- Construct perm-equi function f(X,A) by applying g over all neighbourhoods: $f(\mathbf{X},\mathbf{A}) = \begin{pmatrix} g(x_1, X_{\mathcal{N}_1}) \\ g(x_2, X_{\mathcal{N}_1}) \\ \vdots \\ g(x_n, X_{\mathcal{N}_n}) \end{pmatrix}$. g should not depend on the order of the neighbourhood, it should be permu-invari.
- Once you have the latent-Graph via the GNN you can use them in a Node-classification, Graph-classification, or Link-prediction task.

- Message Passing in Graphs.
  - GNN Layer: Construct f(X,A) via the local function g (known as diffusion, propagation or message passing). F is refered ta as a GNN layer.
  - How to define g? Active research!
  - Classification thre flavoulrs of CNN:
  - Convolutional GNN:
    * constants c_{ij}. How much does Node i value the features of nodes j. They are coefficients for weighted combinations. The weights usually depend on A.
    * $h_i = \phi(x_i, \bigoplus_{j \in \mathcal{N}_i} c_{i,j} \psi(x_j))$.
    * Examples: ChebyNet, GCN (Graph Convolutional Network), SGC (Simplified Graph Convolutional Networks)
    * useful for homophilous graphs (similar edges) and scales well.
  - Attentional GNN:
    * neighbour features aggregated with implicit weights (via attention a). This weights are learnable.
    * $h_i = \phi(x_i, \bigoplus_{j \in \mathcal{N}_i} a(x_i, x_j) \psi(x_j))$.
    * Examples: MoNet, GAT (Graph Attention Network), GaAN (Gated Attention Network).
    * useful as a middle ground with respect to capacity and scale. Edges are not strict homophily, but you compute sclarar value in each edge.
  - Message Passing GNN:
    * sender and receiver work together to compute arbitrary vectors ("messages") to be sent across edges.
    * $h_i = \phi(x_i, \bigoplus_{j \in \mathcal{N}_i} \psi(x_i, x_j)).\psi(x_i, x_j)) = m_{ij}$.
    * Examples: Interaction Networks, MPNN (Message Passing Neural Networks), GraphNets
    * most generic GNN. May have scalability or learnability issues. Ideal for reasoning.

- Node embedding techniques:
  - embedding: Finding an Encoding, so that x-i are now the latent features of h-i.
  - a good representation should preserve the graph structure. This leads to the unsupervised objective: $optimise\ h_i\ and\ h_j\ to\ be\ nearby\ iff : (i, j) \in E$. They predict if there is an edge between the nodes.
  - Can use binary cross-entropy loss: $\sum_{(i,j) \in E} log \sigma(h_i^T h_j) + \sum_{(i,j) \notin E} log(1 - \sigma(h_i^T h_j))$

- local objective emulate Convolutional GNNs. Neighbouring nodes tend to highly overlap in n-step neighborhoods. A conv-GNN enforces similar features for neighbouring nodes by design.

- GNN and Natural Language Processing (NLP) correspond alot (nodes similar to words).
- Common assumption if you have no information about how the graph should look like: Assume a complete graph and then let the network infer the actual relations.
- Transformers: are fully connected attentional GNNs.
- Spectral GNNs:
  - Time Sequences can be imageind as a cyclical grid graph with a convolution over it. A node is a time-step and the convolution looks at the time step and it's immediate neighbors.
  - You don't need to know the convolutional operation if you know the eigenvalues with respect to the fourier basis (36:13)
  - convolutional GNN: $c_{ij} = (p_k(L))_{ij}$. Use a polynomial function p-k for the Laplacian matrix L = D - A. D being the Degree matrix. p-k is necessary to make the eigenvalue decomposition easier.
  - This means there is no spectral GNN and spatial GNN as they can be transformed into each other.
- Probabilistic Graphical Models:
  - Nodes are random variables and edges are dependencies between their distributions. THis is an Probabilistic graphical Model (PGMs). This helps you decompose a joint probability distribution.
  - Can use Markov Random Fields (MRF) to decompose the joint into a product of edge potentials.
  - Mean-field inference.
  - PGM corresponds to a message passing GNN.
- how powerful are GNNs?
  - untrained GNNs work well, as they are similar to random hashes. (Weisfeiler-Lemann Test). Also called 1-WL test.
  - Though some instances the isomorphism test fails.
  - GNNs can only be as powerful as the 1-WL test.
  - Can make the strongers by analysing failer cases.
  - Continous Features: Sums may not destinguish parts of the graph (2+2 = 4+0).
- curr

8.9.2. Probabilistic Deep Learning Book

Probabilistic Deep Learning Book

8.9.3. Geometric Deap Learning

Geometric Deap Learning Youtube

8.9.4. GNN - App - Optimizing Large-Scale Fleet Management on a Road Network using Multi-Agent Deep Reinforcement Learning with Graph Neural Network - 2020

GNN - App - Optimizing Large-Scale Fleet Management on a Road Network using Multi-Agent Deep Reinforcement Learning with Graph Neural Network - 2020

8.9.5. GNN - Deep Multi-Agent Reinforcement Learning with Relevance Graphs - 2018

GNN - Deep Multi-Agent Reinforcement Learning with Relevance Graphs - 2018

8.9.6. GNN - Deep Implicit Coordination Graphs for Multi-agent Reinforcement Learning- 2020

GNN - Deep Implicit Coordination Graphs for Multi-agent Reinforcement Learning- 2020

8.9.7. GNN - Multi-Agent Game Abstraction via Graph Attention Neural Network - 2020

GNN - Multi-Agent Game Abstraction via Graph Attention Neural Network - 2020

8.9.8. GNN - Scaling Up Multiagent Reinforcement Learning for Robotic Systems Learn an Adaptive Sparse Communication Graph - 2020

GNN - Scaling Up Multiagent Reinforcement Learning for Robotic Systems Learn an Adaptive Sparse Communication Graph - 2020

8.9.9. GNN - Graphcomm A Graph Neural Network Based Method for Multi-Agent Reinforcement Learning - 2021

GNN - Graphcomm A Graph Neural Network Based Method for Multi-Agent Reinforcement Learning - 2021

8.9.10. GNN - Towards Heterogeneous Multi-Agent Reinforcement Learning with Graph Neural Networks - 2020

GNN - Towards Heterogeneous Multi-Agent Reinforcement Learning with Graph Neural Networks - 2020

8.9.11. GNN - The Emergence of Adversarial Communication in Multi-Agent Reinforcement Learning - 2020

GNN - The Emergence of Adversarial Communication in Multi-Agent Reinforcement Learning - 2020

8.9.12. GNN - Multi-Agent Deep Reinforcement Learning using Attentive Graph Neural Architectures for Real-Time Strategy Games - 2021

GNN - Multi-Agent Deep Reinforcement Learning using Attentive Graph Neural Architectures for Real-Time Strategy Games - 2021

8.9.13. GNN - Global-Localized Agent Graph Convolution for Multi-Agent Reinforcement Learning - 2021

GNN - Global-Localized Agent Graph Convolution for Multi-Agent Reinforcement Learning - 2021

8.9.14. GNN - Specializing Inter-Agent Communication in Heterogeneous Multi-Agent Reinforcement Learning using Agent Class Information - 2020

GNN - Specializing Inter-Agent Communication in Heterogeneous Multi-Agent Reinforcement Learning using Agent Class Information - 2020

8.9.15. GNN - Collaborative Multiagent Reinforcement Learning by Payoff Propagation - 2006

GNN - Collaborative Multiagent Reinforcement Learning by Payoff Propagation - 2006

8.9.16. Write up for GCNs - 2016

Write up for GCNs - 2016

8.9.17. GNN - Sur - A Comprehensive Survey on Graph Neural Networks - 2019

GNN - Sur - A Comprehensive Survey on Graph Neural Networks - 2019

Bibliography

M. P. Deisenroth, G. Neumann, and J. Peters. *A survey on policy search for robotics*. now publishers, 2013.

Appendix A.

Example Appendix

This is an example for an appendix.