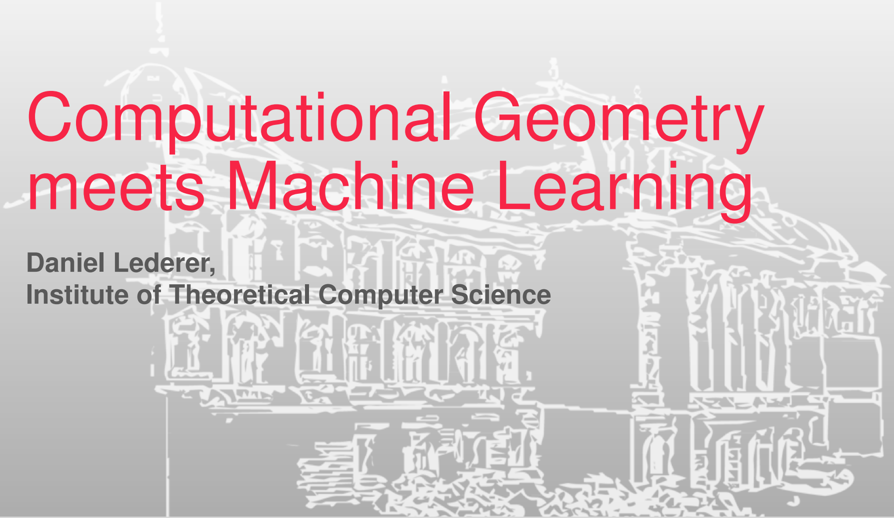


Computational Geometry meets Machine Learning

Daniel Lederer,
Institute of Theoretical Computer Science

Introduction

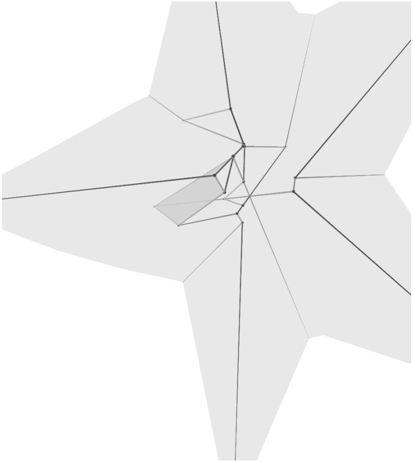
- We are interested in **straight skeletons** of (non-convex) polytopes in $\mathbb{R}^3$
- Straight skeleton structure $\mathcal{R}$ is defined by a mitered **boundary offsetting process**
- For each vertex v with $\deg(v) > 3$, its **offset surface** Σ is computed by the offset **plane arrangement** $\mathcal{A}(v)$
- Boundary of union of all relevant unbounded arrangement cells defines Σ, but ...
    - Σ may not be valid
    - Σ may contain bounded facets $\Rightarrow$ complicates $\mathcal{R}$

Cell Adding Process (CAP)

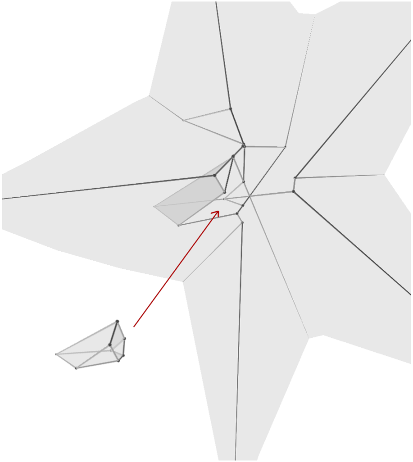
```
Let $\mathcal{F}$ be the set of bounded facets in $\Sigma$
while $\mathcal{F}$ is not empty:
  Choose $f \in \mathcal{F}$
  Let $c$ be the bounded cell upon $f$
  $\Sigma \leftarrow \Sigma \cup c$
  Update $\mathcal{F}$
```

- Add bounded arrangement cells to Σ until all facets become unbounded
- This is always possible by the structure of $\mathcal{A}(v)$
- After CAP, Σ is valid and contains only unbounded facets

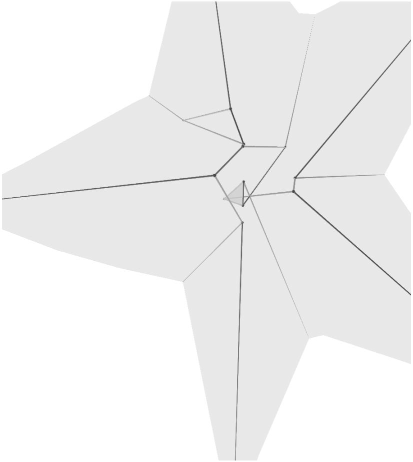
CAP Example



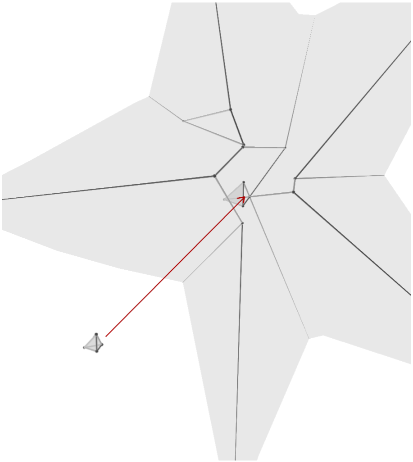
CAP Example



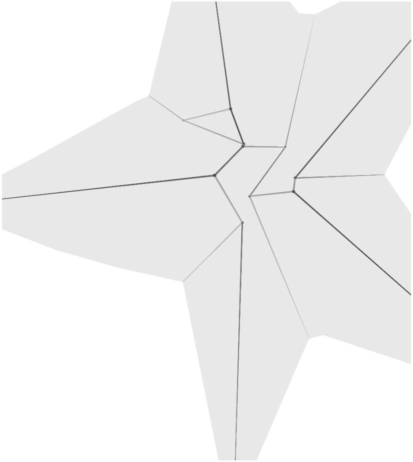
CAP Example



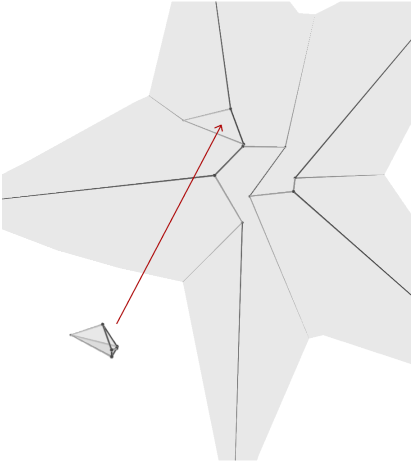
CAP Example



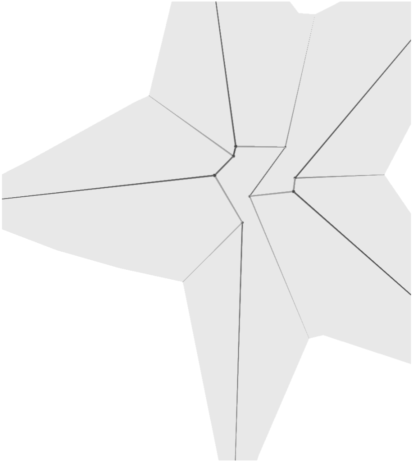
CAP Example



CAP Example



CAP Example



CAP Algorithm Analysis

```
Let $\mathcal{F}$ be the set of bounded facets in $\Sigma$
while $\mathcal{F}$ is not empty:
  Choose $f \in \mathcal{F}$
  Let $c$ be the bounded cell upon $f$
  $\Sigma \leftarrow \Sigma \cup c$
  Update $\mathcal{F}$
```

- $\mathcal{F}$ can easily be found
- Iteration over all facets in Σ

CAP Algorithm Analysis

```
Let $\mathcal{F}$ be the set of bounded facets in $\Sigma$
while $\mathcal{F}$ is not empty:
    Choose $f \in \mathcal{F}$
    Let $c$ be the bounded cell upon $f$
    $\Sigma \leftarrow \Sigma \cup c$
    Update $\mathcal{F}$
```

- There may be many facets in $\mathcal{F}$
- Which $f \in \mathcal{F}$ should we choose?
- Does it even matter which f we choose? **Yes!**

CAP Algorithm Analysis: Choose $f \in \mathcal{F}$

- For a bounded facet f there is a unique cell c upon f
- $\Sigma \cup c$ may create new bounded facets $f', f'', \ldots$ on Σ
  $\Rightarrow \mathcal{F}$ is extended by $f', f'', \ldots$
- Order of the facets we choose in CAP is crucial for the updated or final Σ
- If the order is chosen in an unfavorable way, it is no longer possible to get rid of all bounded facets! **Why?**
  - c may be unbounded for some f
  - If c is unbounded $\Rightarrow$ we must not $\Sigma \cup c$
    $\Rightarrow f$ may remain forever on Σ and in $\mathcal{F}$

CAP Algorithm Analysis

```
Let $\mathcal{F}$ be the set of bounded facets in $\Sigma$
while $\mathcal{F}$ is not empty:
  Choose $f \in \mathcal{F}$
  Let $c$ be the bounded cell upon $f$
  $\Sigma \leftarrow \Sigma \cup c$
  Update $\mathcal{F}$
```

- There is always a unique cell c upon f
- c can easily be determined from the structure of $\mathcal{A}(v)$
- If $f \in \mathcal{F}$ is chosen properly $\Rightarrow$ c is always bounded

CAP Algorithm Analysis

```
Let $\mathcal{F}$ be the set of bounded facets in $\Sigma$
while $\mathcal{F}$ is not empty:
  Choose $f \in \mathcal{F}$
  Let $c$ be the bounded cell upon $f$
  $\Sigma \leftarrow \Sigma \cup c$
  Update $\mathcal{F}$
```

- Merging c to Σ is always doable
- Updated Σ is always unique

CAP Algorithm Analysis

```
Let $\mathcal{F}$ be the set of bounded facets in $\Sigma$
while $\mathcal{F}$ is not empty:
  Choose $f \in \mathcal{F}$
  Let $c$ be the bounded cell upon $f$
  $\Sigma \leftarrow \Sigma \cup c$
  Update $\mathcal{F}$
```

- Σ was only updated locally around c
- $\mathcal{F}$ can thus be updated efficiently

Problem Definition

- Given is a (not necessarily valid) surface Σ that contains bounded facets
- We are looking for a **set of cells** $\mathcal{C}$ such that $\Sigma' = \Sigma \cup \mathcal{C}$ contains unbounded facets only $\Rightarrow \Sigma'$ is valid
- $\mathcal{C}$ certainly exists due to the structure of $\mathcal{A}(v)$
- $\mathcal{C}$ does not have to be unique $\Rightarrow \Sigma'$ may not be unique
- We are already satisfied with one instance of $\mathcal{C}$
- For a given Σ, **how to find $\mathcal{C}$?**

Current Solution Approach

- We extend $\mathcal{C}$ step by step according to the rules:
  1. If $f \in \mathcal{F}$ is not visible from $v \Rightarrow f$ cannot belong to $\Sigma' \Rightarrow$ add c to $\mathcal{C}$, for c upon f, to make f disappear
  2. Add any c to $\mathcal{C}$ for which no new bounded facets are created (c fits perfectly to the surface)
  3. Add any c to $\mathcal{C}$ that have at least 2 facets in common with the surface and all of them are connected
  4. If none of the above 3 criteria apply $\Rightarrow$ select any $f \in \mathcal{F}$ and add the corresponding bounded c to $\mathcal{C}$

Solution Discussion

- If rule (1) is applicable $\Rightarrow$ c surely belongs to the solution set $\mathcal{C}$
- Otherwise, (2), (3) or (4) is applicable (descending priority)
- There is room for improvement in the remaining three rules
- Current approach does not lead to the goal for all problem instances
- It is likely that a better heuristic algorithm exists

Different Solution Approach

- Given is a combinatorial optimization problem
- Finding $\mathcal{C}$ is probably NP-hard (open question)
- Different problem instances (data) but same problem structure
- *Idea:* Apply **Machine Learning** to learn patterns in the data to exploit problem structure
- Machine learning based methods are able to find new, previously unknown heuristics
- Automatically learn good heuristic algorithms for finding $\mathcal{C}$

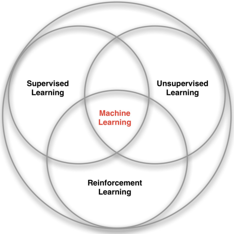
Machine Learning

- Machine Learning (ML) is a sub-discipline of Artificial Intelligence in which a machine learns intelligent behaviour
- ”A computer program is said to learn from experience *E* with respect to some task *T* and some performance measure *P*, if its performance on *T*, as measured by *P*, improves with experience *E*.” [1]
    - *T*: Driving on highways using vision sensors
    - *P*: Average distance traveled before an error
    - *E*: Images sequence & steering commands by humans [1]

[1] Mitchell, T. (1997). Machine Learning. McGraw Hill. p. 2. ISBN 978-0-07-042807-2.

Machine Learning Paradigms

- **Unsupervised Learning:**
  Learning properties and patterns in unlabeled data
- **Supervised Learning:**
  Learning a mapping of data (input) to labels (output)
- **Reinforcement Learning:**
  Learning the optimal policy of an agent interacting with an environment

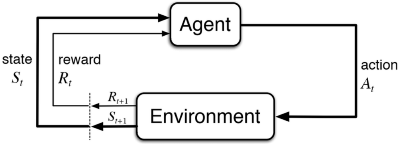


The three basic paradigms of ML

Reinforcement Learning (RL)

- An **agent** interacts with an **environment**
- During this time, the agent makes decisions in the form of **actions**
- Environment changes its internal **state** and emits a **reward** (weak feedback)
- Goal is to maximize the accumulated reward during the agent's lifetime
  - ⇒ Agent's behaviour is *reinforced* (therefore "reinforcement learning")
  - ⇒ Optimal **policy** is learned

Agent-Environment Interaction



- In a discrete timestep t the environment is in state S_t
- Agent chooses and performs an action A_t
- Environment changes its state to S_{t+1} and emits a reward R_{t+1} (next timestep)
- Based on that, the agent chooses the next action A_{t+1}, etc... $\Rightarrow S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, \ldots$

Our Agent & Environment

- Agent is *”an invisible character”*
- Agent decides on a specific $f \in \mathcal{F} \Rightarrow$ associated c is merged onto Σ
- Environment is *”anything outside of the agent”*
- Environment includes
  - surfaces
  - plane arrangements
  - merging dynamics
  - everything else

Episode

- When our agent interacts with our environment, an **episode** is generated
- Episode starts in an initial state (at $t = 0$) and ends in a terminal state (at $t = T$)
- Sequence: $S_0, A_0, R_1, S_1, A_1, ..., A_{T-1}, R_T, S_T$
- Initial state S_0 is represented by Σ
- Terminal state S_T is represented by $\Sigma' = \Sigma \cup \{c_1, ..., c_T\}$

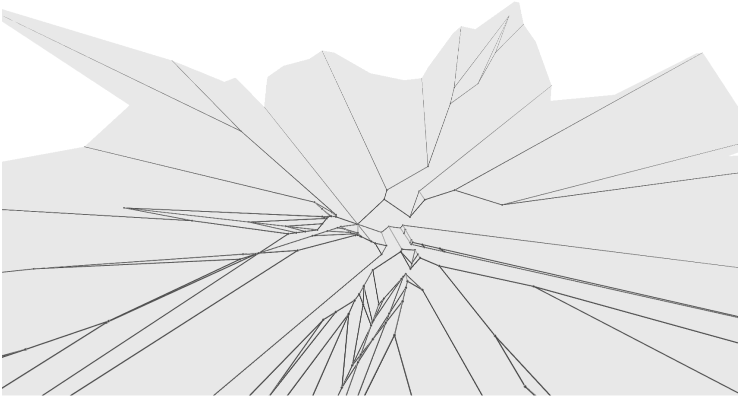
POMDP

- **Partially Observable Markov Decision Process**
- 6-tuple $(\mathcal{S}, \mathcal{A}, \Omega, T, O, R)$
  - $\mathcal{S}$: State space
  - $\mathcal{A}$: Action space
  - Ω: Observation space
  - $T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$: Transition function
  - $O : \mathcal{S} \times \mathcal{A} \times \Omega \to [0, 1]$: Observation function
  - $R : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}$: Reward function

State

- A **state** describes the current situation in the environment
- A state of our environment is represented by Σ
- Σ is described by its vertices, edges and facets and their connectivity information
- Note that state space $\mathcal{S}$ comprises all possible surfaces than can occur in the environment ($\mathcal{S}$ is infinite)

State Example



Σ

Observation

- For each state $s \in \mathcal{S}$ there is an **observation** $o \in \Omega$
- Typically, o does not provide all information of s, o only contains hints about how s looks like
- We already know s, so why are we interested in o?
  - s provides too much information
  - Scaling/Translation/Rotation of Σ is irrelevant in our problem
  - Exact position of each vertex on Σ is redundant

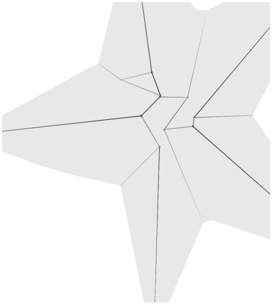
Observation

- What part of Σ might be important for encoding the problem structure?
  - Connectivity information between facets
  - Relation between facets in the space
  - Sharing planes of facets
- Formulating o as a **graph** G
- Think of G as a reduced special dual representation of Σ
- Let $\mathcal{N}$ be the nodes and $\mathcal{E}$ the edges of G

Graph Nodes

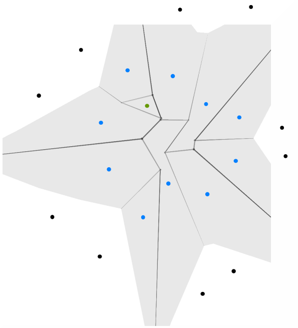
- Each facet $f \in \Sigma$ corresponds to a node $n \in \mathcal{N}$
    - If f is bounded and its associated c is bounded $\Rightarrow$ type $n_t = 1$
    - If f is bounded but c is unbounded $\Rightarrow n_t = 2$
    - If f is unbounded $\Rightarrow n_t = 3$
- Each plane on Σ corresponds to a node $n \in \mathcal{N}$ with $n_t = 4$

Graph Nodes Example



State

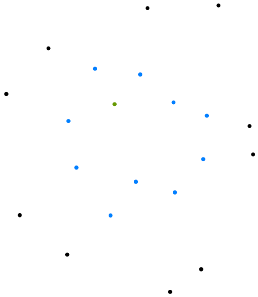
Graph Nodes Example



State together with $\mathcal{N}$

(green for $n_t = 1$, blue for $n_t = 3$, black for $n_t = 4$)

Graph Nodes Example

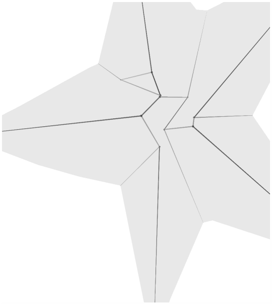


$\mathcal{N}$

Graph Edges

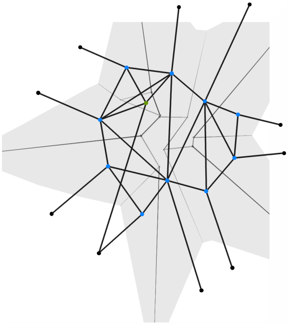
- Each edge $e' \in \Sigma$ corresponds to an edge $e \in \mathcal{E}$, connecting the nodes of neighbouring facets
- Each facet $f \in \Sigma$ corresponds to an edge $e \in \mathcal{E}$, connecting the facet's node with its supporting plane's node
- Each $e \in \mathcal{E}$ has weight e_w, corresponding to the interior angle between the two facets on Σ
    - If e is adjacent to a plane node $\Rightarrow e_w = 0$, since in the imagination the facet encloses no angle with its plane

Graph Edges Example



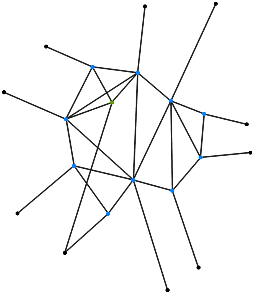
State

Graph Edges Example



State together with $\mathcal{N}$ and $\mathcal{E}$
(without weights)

Graph Edges Example

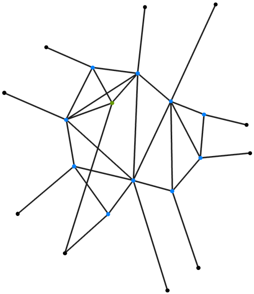


$\mathcal{N}$ and $\mathcal{E}$

Action

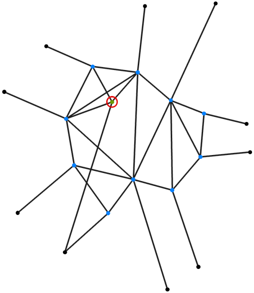
- Based on the current observation, the agent chooses an **action**
- An action expresses what the agent can do in the current observation of the environment
- An action of our agent is represented by selecting a node $n \in \mathcal{N}$ of type $n_t = 1$
- n is associated with a (bounded) facet $f \in \Sigma$
- f is associated with a (bounded) cell $c \in \mathcal{A}(v)$
- The action causes merging of c onto $\Sigma \Rightarrow$ state and observation transition

Action Example



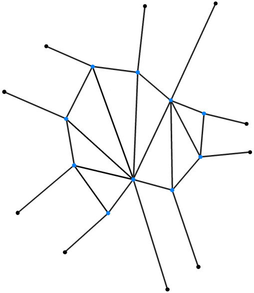
G

Action Example



Selecting a node of type $n_t = 1$

Action Example



Next G

Reward

- Based on the current state and selected action, the environment emits a **reward**
- The reward tells the agent how good its action was
- It is often represented by a scalar *r*
- The higher *r*, the better the action was
- **What does our reward look like?**
- **Can we give a reward after each action?**

Reward

- There might be cells $c_1, c_2, ... \in \mathcal{A}(v)$ that certainly belong to our solution set $\mathcal{C}$
- But instead of giving these actions a higher reward and learn from this, we can directly insert $c_1, c_2, ...$ into $\mathcal{C} \Rightarrow$ *Expert Policy* (discussed later)
- Actions that directly lead to the success/failure state, should get a high/low reward
- For all intermediate actions, we cannot state in general which ones should be prefered

Reward

- Let $s \in \mathcal{S}$ be the current state
- Let $a \in \mathcal{A}$ be the selected action
- Let $s' \in \mathcal{S}$ be the next state when performing a in s
- Then,
  - $r(s, a) = 0$ if s' is no terminal state
  - $r(s, a) = +1$ if s' is a success terminal state
  - $r(s, a) = -1$ if s' is a failure terminal state
- Reward signal is very sparse

Policy

- The **policy** π describes the behaviour of the agent
- π represents the strategy of the agent
- Using π, the agent selects an action $a \in \mathcal{A}$ for the given observation $o \in \Omega$
- Formally, π can be represented as $\pi(o) = a$
- What is our policy?

$\Rightarrow$ **A good policy is what we want to learn!**

Expert Policy

- If $f \in \mathcal{F}$ is not visible from $v \Rightarrow f$ cannot belong to Σ' $\Rightarrow$ add c to $\mathcal{C}$, for c upon f, to make f disappear
- If such f exists, select corresponding node $n \in \mathcal{N}$ as action (**expert policy**)
- The action suggested by the expert policy is always optimal
- We already have the expert policy, so we do not need to learn it
- But the expert policy cannot always suggest an action $\Rightarrow$ we need another policy

ϵ-greedy Policy

- We need to explore the environment well enough to learn about the problem structure and a good policy
- We have to find a suitable balance between exploration and exploitation (= applying what has already been learned)
- Introduce a small value ϵ between 0 and 1
- With a probability of ϵ take a random action, otherwise take the *greedy action* (**ϵ-greedy policy**)
- We have defined policies, but **how do we learn a good one?**

Q-Learning

- **Q-Learning** is an algorithm that learns a Q-value for each o, a pair
- This value is used to determine the quality of a in o
- **Q-function** $Q(o, a)$ is trained
  - $\Rightarrow$ Optimal Q-function $Q^*(o, a)$
  - $\Rightarrow$ Optimal policy $\pi^*(o) = \text{argmax}_a Q^*(o, a)$
  - $\Rightarrow$ Best a for given o
- Update Rule: $Q(O_t, A_t) \leftarrow Q(O_t, A_t) + \alpha \cdot$

$$[R_{t+1} + \gamma \max_A Q(O_{t+1}, A) - Q(O_t, A_t)]$$

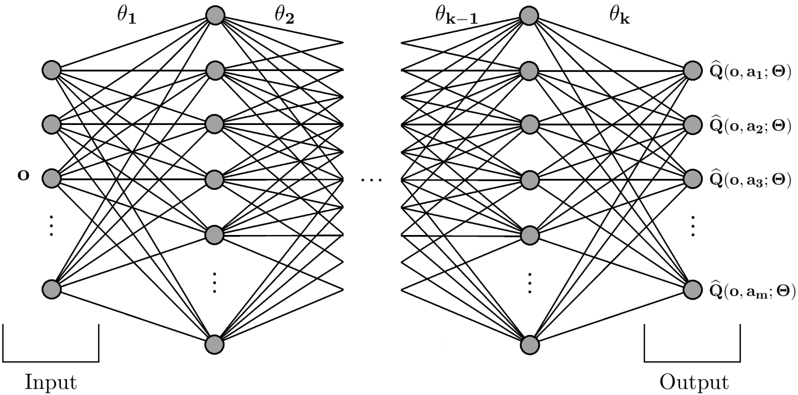
Q-Learning

- At the beginning of training, *Q*-function returns arbitrary values for all o, a pairs
- After sufficient exploration of the environment and application of the update rule $\Rightarrow Q(o, a)$ becomes more meaningful
- **Problem:** $\mathcal{S}, \mathcal{A}, \Omega$ are infinite, so we cannot store and update all o, a pairs
- **Solution:** Generalization over several observations and actions $\Rightarrow$ **Q-Function approximation**

Deep Q-Learning

- Creating a function approximation $\widehat{Q}(o, a; \Theta) \approx Q(o, a)$
- $\widehat{Q}(o, a; \Theta)$ can be computed with a Deep Neural Network with model parameters $\Theta \Rightarrow$ **Deep Q-Network** (DQN)
- DQN approximates the Q-value for each o, a pair
- Approximations are sufficient as long as the relative quality assessments are preserved ($\text{argmax}_a \widehat{Q}(o, a; \Theta)$)

DQN



A classical DQN model

Training DQN

- Training of DQN by optimizing its parameters $\Theta = \{\theta_1, ..., \theta_k\}$
- Optimization of Θ by minimizing quadratic loss $\mathcal{L}(\Theta) = \mathbb{E}[(y - \widehat{Q}(O_t, A_t; \Theta))^2]$ with
  $y = R_{t+1} + \gamma \max_A \widehat{Q}(O_{t+1}, A; \Theta)$
- $\mathcal{L}(\Theta)$ can be (locally) minimized using the Stochastic Gradient Descent (SGD) algorithm
- **Problem:** SGD needs *iid* data, but our data are sequentially dependent ⇒ unstable training
- **Solution: Experience Replay**

Experience Replay

- In each timestep t collect an experience $E_t = (O_t, A_t, R_{t+1}, O_{t+1})$
- E_t is stored in a **replay memory** $\mathcal{M}$
- During training, sample a random mini-batch $b \overset{iid}{\sim} \mathcal{M}$ and optimize Θ for b using SGD
- Experiences are reused in many updates $\Rightarrow$ higher data efficiency
- Data in b are uncorrelated $\Rightarrow$ more stable network training

Applying DQN

- When using DQN, we need to consider more issues

(1) DQN only accepts inputs of the same size

  - Input of our DQN is o, or its graph G
  - Different G have different sizes

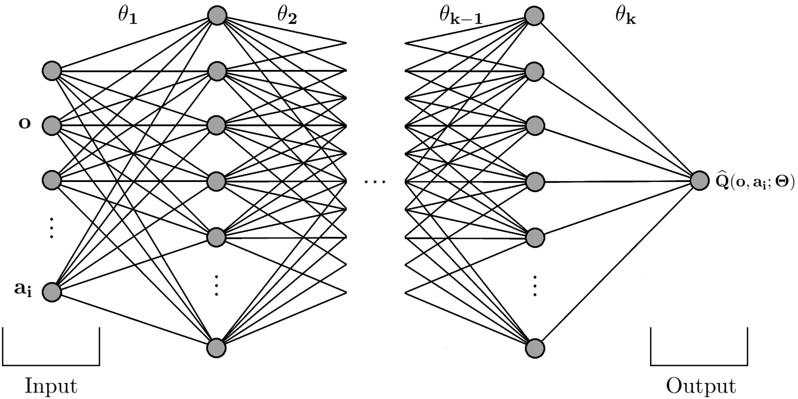
(2) DQN only produces outputs of the same size

  - Output of our DQN is $\widehat{Q}(o, a_i, \Theta), \forall a_i \in \{a_1, ..., a_m\}$
  - Different G have different number of actions m

Applying DQN: Issue (2)

- This matter can be solved quite easily
- Modify DQN
  - Input: o and a single a_i
  - Output: $\widehat{Q}(o, a_i, \Theta)$
  - Compute Q-value for each a_i separately
- DQN can now be applied to different numbers of actions
- *Drawback:* Instead of a single forward pass, we now need m forward passes $\Rightarrow$ takes more time

Updated DQN



The updated DQN model

Applying DQN: Issue (1)

- This matter is not so easy to solve
- Order of the nodes and edges in G should not matter
- Connectivity information between nodes and all features should be preserved
- *Idea:* **Learn a fixed-sized representation** of o and a, independent of the size of G
- Then, input of DQN is also of fixed size
- **Solution: Graph Neural Network**

Graph Neural Network

- A Graph Neural Network (GNN) processes graph-structured data
- GNN simulates an exchange of features between the nodes in the graph (**message passing**)
- For each node, and finally for the graph itself, a low-dimensional vector representation is learned (= **Node**/**Graph embedding**)
- Embeddings include important graph characteristics, e.g. node types, edge weights, neighbourhoods, etc.
- GNN can be seen as a Graph Embedding Network

GNN: Initial Node Embedding

- For each $n \in \mathcal{N}$ we learn a p-dimensional node embedding μ_n
- Initialize μ_n as follows:

$$\mu_n^{(0)} = \text{relu}(\theta_1 t_n), \text{ where}$$

  - t_n is the one-hot encoded type of n
  - $\theta_1 \in \mathbb{R}^{p \times 4}$ are trainable parameters
  - $\text{relu}(x) = \max(0, x)$ is the rectified linear unit
- Initially, global node type information is collected

GNN: Message Passing

- Each n computes a message and transmits it to its neighbours
- Each n aggregates the messages of its neighbours
- Each μ_n is updated synchronously up to T iterations:

  $\mu_n^{(t)} \leftarrow F_{\theta_2}(\mu_n^{(t-1)}, \{\mu_u^{(t-1)}\}_{u \in \mathcal{N}(n)}, \{e_w\}_{e \in \mathcal{E}(n)})$, where

  - $\mathcal{N}(n)$ are the neighbouring nodes of n
  - $\mathcal{E}(n)$ are the adjacent edges of n
  - $\{\cdot\}$ is some aggregation function (e.g. summation)
  - F is a neural network with θ_2 combining them

GNN: Message Passing

- Messages are propagated recursively according to the topology of G
- Only when $\mu_n^{(t)}$ has been computed for all n, the next iteration $t + 1$ starts
- The larger T is, the further the messages are propagated through G
- $\mu_n^{(T)}$ includes information of its T-hop neighbourhood
- Typically a small T is sufficient for a good node representation

Parameterizing $\widehat{Q}(o, a; \Theta)$

- Let $\mu_G = \sum_{n \in \mathcal{N}} \mu_n^{(T)}$ be the graph embedding of G
- Combining the GNN with our DQN, we can parametrize $\widehat{Q}(o, a; \Theta)$ with $\Theta = \{\theta_1, \theta_2, \theta_3\}$:

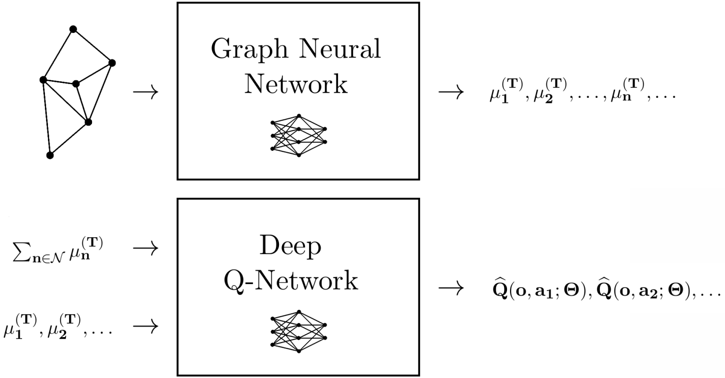
  $$\widehat{Q}(o, a; \Theta) = H_{\theta_3}(\mu_G, \mu_n^{(T)}), \text{ where}$$

  - $\mu_G \in \mathbb{R}^p$ is a representation of o
  - $\mu_n^{(T)} \in \mathbb{R}^p$ is a representation of a
  - H is a neural network with θ_3 combining them
- All parameters in Θ are trainable

Final Model

- Our final deep reinforcement learning model has a GNN and DQN component
- GNN takes a graph and produces node embeddings of fixed size
- Aggregating all node embeddings to a graph embedding of fixed size that represents o
- DQN takes the graph embedding and one node embedding each, which represents a
- DQN produces $\widehat{Q}(o, a; \Theta)$

Final Model



$\rightarrow$ Graph Neural Network $\rightarrow$ $\boldsymbol{\mu}_{\mathbf{1}}^{(\mathbf{T})}, \boldsymbol{\mu}_{\mathbf{2}}^{(\mathbf{T})}, \ldots, \boldsymbol{\mu}_{\mathbf{n}}^{(\mathbf{T})}, \ldots$

$\sum_{\mathbf{n} \in \mathcal{N}} \boldsymbol{\mu}_{\mathbf{n}}^{(\mathbf{T})}$ $\rightarrow$

$\boldsymbol{\mu}_{\mathbf{1}}^{(\mathbf{T})}, \boldsymbol{\mu}_{\mathbf{2}}^{(\mathbf{T})}, \ldots$ $\rightarrow$

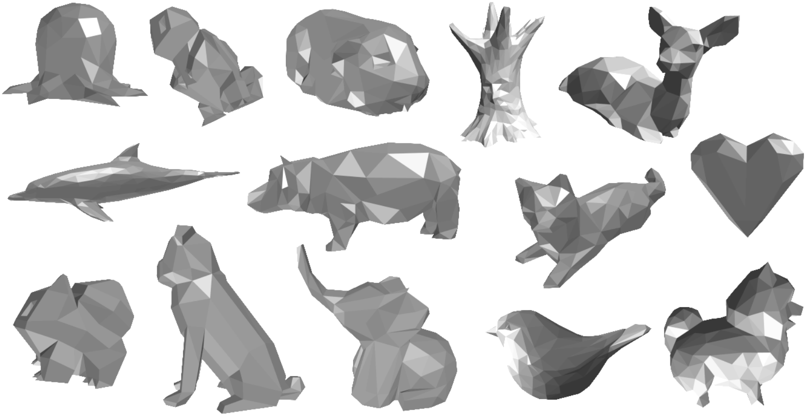
Deep Q-Network $\rightarrow$ $\widehat{\mathbf{Q}}(\mathbf{o}, \mathbf{a_1}; \boldsymbol{\Theta}), \widehat{\mathbf{Q}}(\mathbf{o}, \mathbf{a_2}; \boldsymbol{\Theta}), \ldots$

The final GNN-DQN model

Training

- All parameters in the GNN and DQN can be learned end-to-end using reinforcement learning
- Training is done by SGD using experience replay
- In order for our model to understand the underlying problem structure, we need a **lot of training data**
- Initial task was to compute offset surfaces of polyhedra in $\mathbb{R}^3$
- Therefore, take various real world data on which our model should be trained

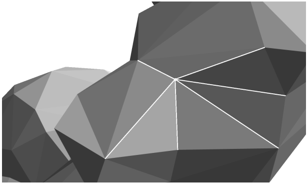
Training Data



Some polyhedra

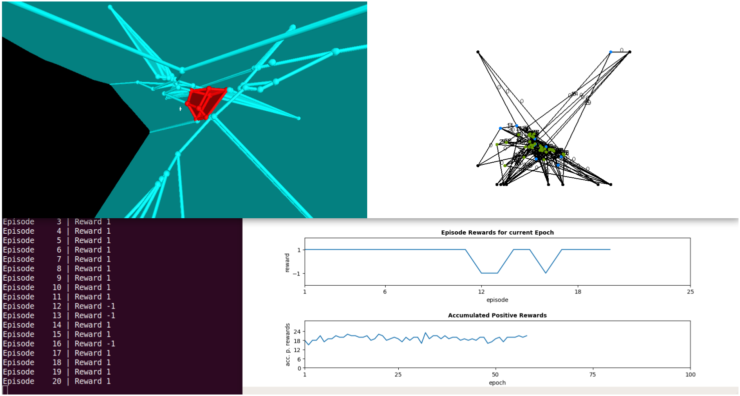
Training Sample

- We need many polyhedra in $\mathbb{R}^3$
- Each *higher-degree* vertex with its local neighbourhood is a **training sample** *d*
- For each *d* inital Σ is computed and CAP is applied
- CAP generates all experiences used in training



A training sample

Time for demo!



```
Episode     3 | Reward 1
Episode     4 | Reward 1
Episode     5 | Reward 1
Episode     6 | Reward 1
Episode     7 | Reward 1
Episode     8 | Reward 1
Episode     9 | Reward 1
Episode    10 | Reward 1
Episode    11 | Reward 1
Episode    12 | Reward -1
Episode    13 | Reward -1
Episode    14 | Reward 1
Episode    15 | Reward 1
Episode    16 | Reward -1
Episode    17 | Reward 1
Episode    18 | Reward 1
Episode    19 | Reward 1
Episode    20 | Reward 1
```

Summary

- Σ is required for constructing a straight skeleton of a polyhedron in $\mathbb{R}^3$
- Σ may contain bounded facets
- CAP algorithm does not always find a suitable Σ
- Subroutine based on reinforcement learning should learn a good policy for CAP
- Dual representation of Σ as G for learning model
- Q-learning with GNN and DQN
- Training requires a lot of data

Outlook

- Clear setup of the training dataset
  - Which and how many polyhedron vertices and of what degree?
- Implementation of the Deep Q-Learning Model with the Graph Embedding Network
- Hyperparameter tuning (ϵ, γ, p, T, epochs, network architectures, etc.)
- Testing and evaluation
  - How well does the learning model perform on previously unseen data?