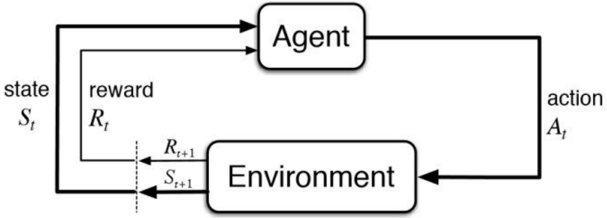
Multi-Agent Path Planning with Evolutionary Graph Reinforcement Learning

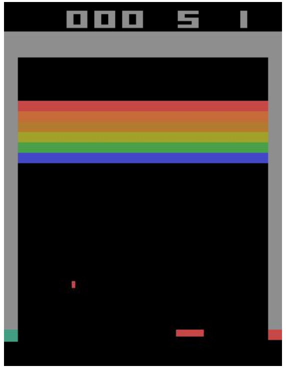
Introduction

- Objective of multi-robot deployments: Avoiding collisions, reaching the goal positions
- Multi-agent path planning
  - Centralized
    - Inefficient, inflexible
  - Decentralized
    - More efficient and flexible, able to plan online
- This project implements a multi-agent path planning system that uses evolutionary reinforcement learning to train a policy that can be used by each agent to plan its own path
- Uses a graph representation of the environment map for training

Preliminaries: Reinforcement Learning Overview

- Problems involving an agent interacting with an environment, which provides numeric reward signals
- Based on Markov Decision Processes (MDPs)
  - Action + State → Reward and new state
- Goal of RL: Learn the optimal mapping from states to actions that leads to the maximum cumulative reward





Preliminaries: Evolutionary Algorithms

- Search-based optimization technique
- Step 1: Initialization
    - Randomly generate a population over the search space
- Step 2: Selection
    - A portion of the existing population is selected to create a new population
    - "Fitter" solutions are more likely to be selected, as measured by some fitness function that measures the quality of each solution
- Step 3: Crossover and/or mutation
    - A way of combining previous solutions to generate a new (better) solution
    - Mutation will generate new variations from the previous solution
- MAPPER simply takes the best model among all the agents and propagates it to all the other agents

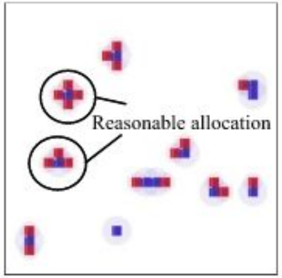
Multi-Agent Evolutionary Reinforcement Learning from MAPPER

- **Initialization:** Initialize N agents with random weights for their own model
  - $\Theta = \{\Theta_1, ..., \Theta_N\}$
- **Train each agent's model separately** using A2C algorithm
  - After k training episodes, agent i will accumulate rewards over the last k episodes, denoted by $R^{(k)}_i$.
  - Assuming agent j has the maximum normalized reward, start crossover and selection phases
- **Selection:** Find the agent that has the greatest normalized reward after k training episodes
  - $\frac{R_i^{(k)}}{R_{max}^{(k)} - R_{min}^{(k)}}$
-
- **Crossover:** agent i keeps its original weights with probability p_i and replaces its weights with agent j's weights with probability $1-p_i$. In this equation, η is the evolution rate
  - Larger η means agents with lower rewards are more likely to be updated
  - *m* is sampled from U~[0,1]. If m < p_i, then agent i gets agent j's weights
- **Repeat** until convergence

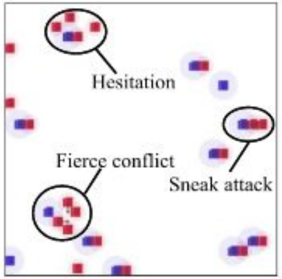
$$p_i = 1 - \frac{\exp(\eta \bar{R}_i^{(k)})}{\exp(\eta \bar{R}_j^{(k)})}$$

Graph Convolutional Reinforcement Learning (DGN)

- Tackles the multi-agent path planning problem using graphs
- Agents can be represented as nodes in a graph
- Relations between agents help achieve greater performance



(c) DGN in jungle



(d) DQN in jungle

DGN Architecture

- Observation encoder
  - Local observation is encoded into a feature vector by MLP (for low dimensional input) or CNN (for visual input)
- Convolutional layer
  - Integrates features in the local region (a given node + its neighbors) and generates a latent feature vector
  - Convolutional layers are stacked to increase the receptive field of each agent, increasing the scope of cooperation
- Q-Network
  - Feature vectors from the convolutional layers are fed into the Q-network (explained on next slide with the loss function?)

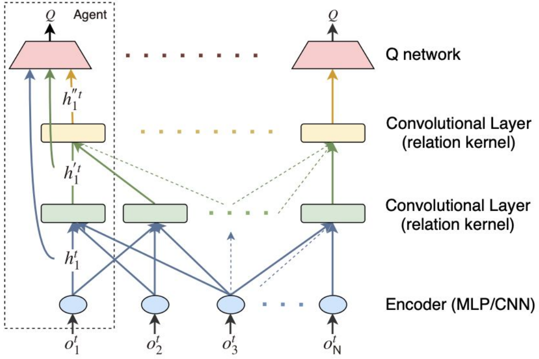
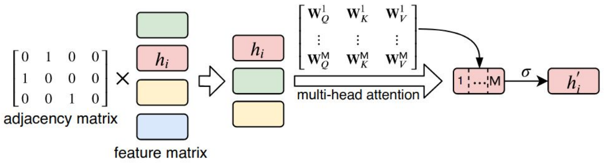


Figure 1: DGN consists of three modules: encoder, convolutional layer, and Q network. All agents share weights and gradients are accumulated to update the weights.

DGN Relation Kernel

- Relationships between agents are extracted using a relation kernel
  - More formally known as multi-head dot-product attention
- For attention head *m*, the relation between *i* and *j* is computed as

$$\alpha_{ij}^m = \frac{\exp\left(\tau \cdot \mathbf{W}_Q^m h_i \cdot (\mathbf{W}_K^m h_j)^\mathsf{T}\right)}{\sum_{k \in \mathbb{B}_{+i}} \exp\left(\tau \cdot \mathbf{W}_Q^m h_i \cdot (\mathbf{W}_K^m h_k)^\mathsf{T}\right)},$$



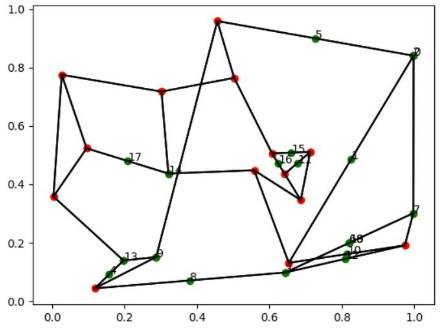
$$h_i' = \sigma(\text{concatenate}[\sum_{j \in \mathbb{B}_{+i}} \alpha_{ij}^m \mathbf{W}_V^m h_j, \forall m \in \mathsf{M}]).$$

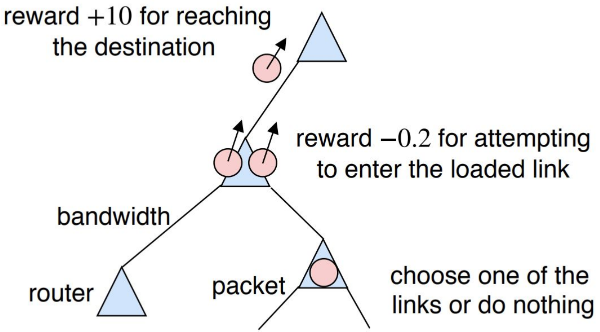
Evolutionary DGN

- Train multiple DGN models simultaneously
- Calculate evolution probability based on max accumulated reward
  - Normalize accumulated reward for all models and select largest
  - Update individual model based on evolutionary probability $p_i = 1 - \frac{\exp(\eta \bar{R}_i^{(k)})}{\exp(\eta \bar{R}_j^{(k)})}$
- Repeat until models converge

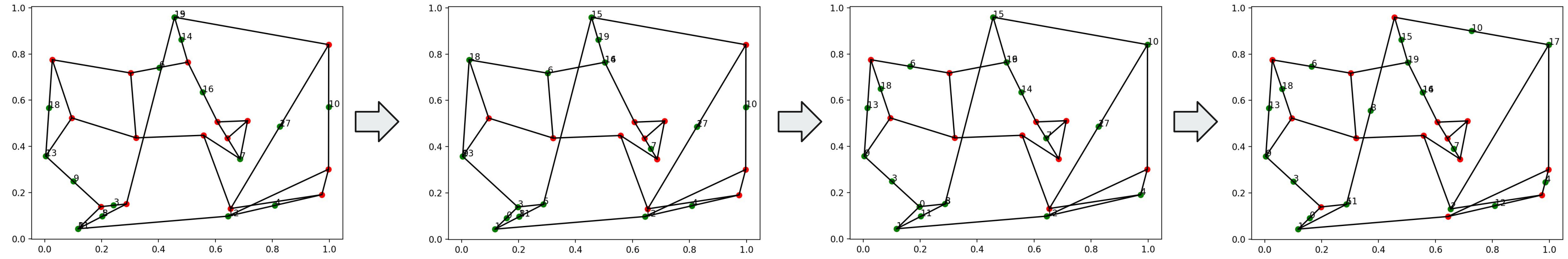
Routing

- Environment
  - Graph where nodes are routers and edges are connections between nodes
  - Each node connected to three other nodes
- Agents
  - Data packet with attributes
    - Current node
    - Destination node
    - Packet size
- Actions
  - Travel to any neighboring node
- Goal
  - Route all packets to destination node
  - Decrease congestion





Code Demo



Training with evolution:

- Training is tricky!
  - DGN: One "agent" controls all, so singular model
  - Evolutionary: Each agent has its own model
- Solution:
  - Still use DGN's singular "agent"
  - Have multiple of these "agents"
  - Swap order of loops
    - Similar to changing order of integration
  - This means we train K times and then change agent

Algorithm 1 Multi-Agent Evolutionary Training Approach

Require: Agents number N; discount factor γ; evolution interval K; evolution rate η;

Initialize agents' model weights $\Theta = \{\Theta_1, ..., \Theta_N\}$
repeat
  Set accumulated reward $R_1^{(k)}, ..., R_N^{(k)} = 0$
  // update model parameters via A2C algorithm
  for $k = 1, ..., K$ **do**
    for each agent i **do**
      Executing the current policy π_{Θ_i} for T timesteps, collecting action, observation and reward $\{a_i^t, \boldsymbol{o}_i^t, r_i^t\}$, where $t \in [0, T]$
      Compute return $R_i = \sum_{t=0}^{T} \gamma^t r_i^t$
      Estimate advantage $\hat{A}_i = R - V^{\pi_{\Theta_i}}(\boldsymbol{o}_i)$
      Compute gradients $\nabla_{\Theta_i} J = \mathbb{E}[\nabla_{\Theta_i} \log \boldsymbol{\pi}_{\Theta_i} \hat{A}_i]$
      Update Θ_i based on gradients $\nabla_{\Theta_i} J$
    end for
    $R_i^{(k)} = R_i^{(k)} + R_i$
  end for
  Normalize accumulated reward to get $\bar{R}_1^{(k)}, ..., \bar{R}_N^{(k)}$
  Find maximum reward $\bar{R}_j^{(k)}$ with agent index j
  // Evolutionary selection
  for each agent i **do**
    Sample m from uniform distribution between $[0, 1]$
    Compute evolution probability $p_i = 1 - \frac{\exp(\eta \bar{R}_i^{(k)})}{\exp(\eta \bar{R}_j^{(k)})}$
    if $m < p_i$ **then**
      $\Theta_i \leftarrow \Theta_j$
    end if
  end for
until converged

Training with evolutionary algorithm:

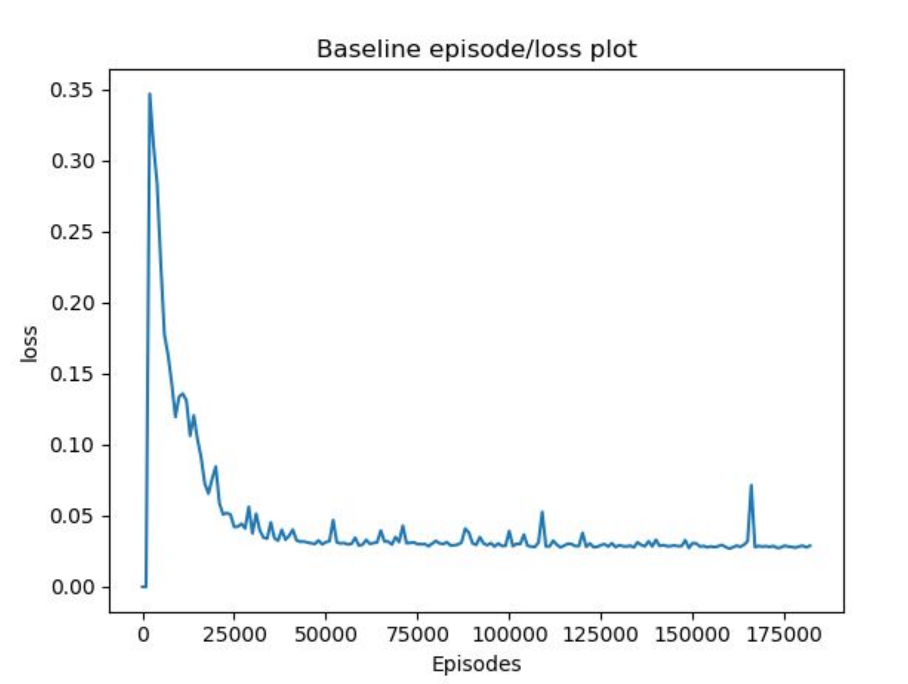
- ★ Still use single “agent” / model to control whole routing process
  - However, now train many “agents” + do evolution globally
  - Optional mutation parameter to treat whole “agent” as multiple separate parts
    - Analogy: an agent is a car. Take a wheel or engine from best car and put it in ours (mutation)
      - Or replace the whole car and rely on randomness
- ★ Replay Buffer in DGN: After 2000 episodes, begin training. Reset loss/score/etc every 100 episodes
- ★ Solution :
  - Ensure that (evolution interval) % (reset every) == 0
  - Adds another hyper parameter
    - We must choose number of agents, evolution interval, evolution rate, reset-every, ...

Results - baseline

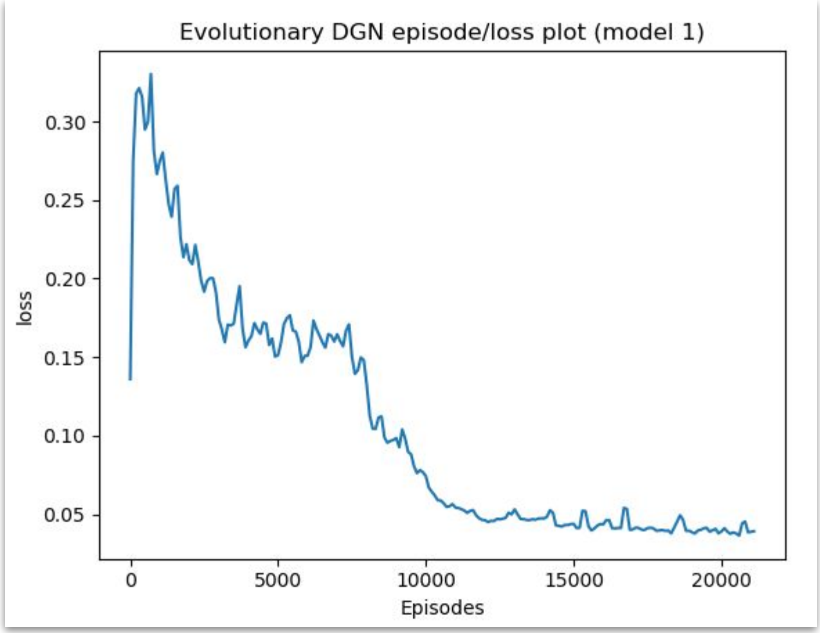
Note that at first “loss is 0” because we only
Start training after the first 2000 eps

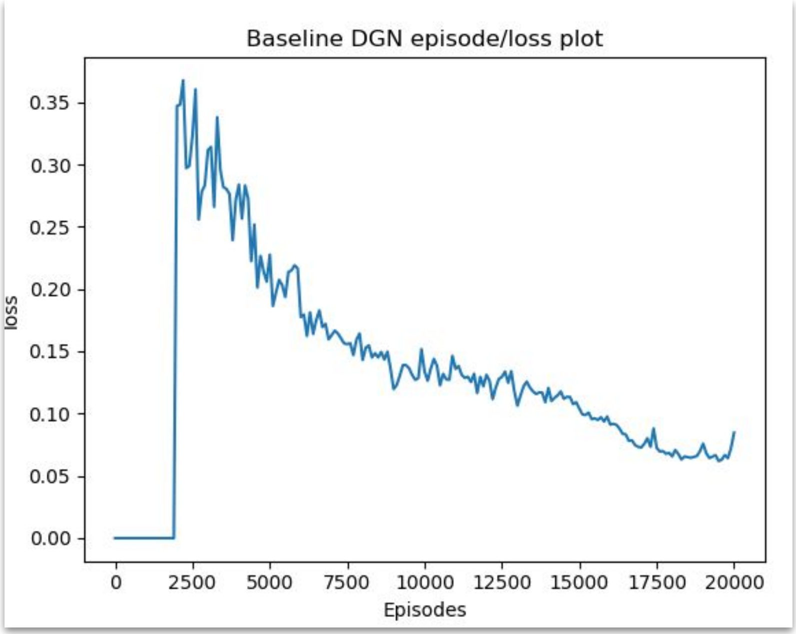
Baseline DGN converges fast; after 20-50k eps

“Loss” is actually average of prev. 100 loss terms



Results - evolutionary vs baseline





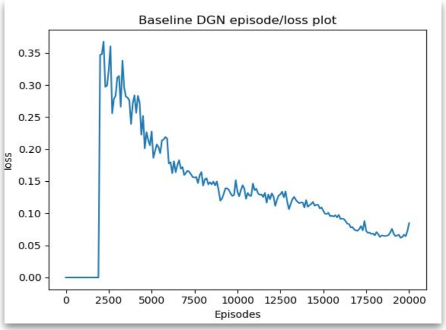
Results - evolutionary vs baseline

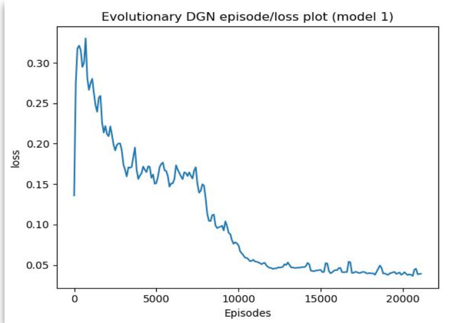
We do achieve lower loss for the same episodes

However, note that we are training 4 models at the same time (without mutation)

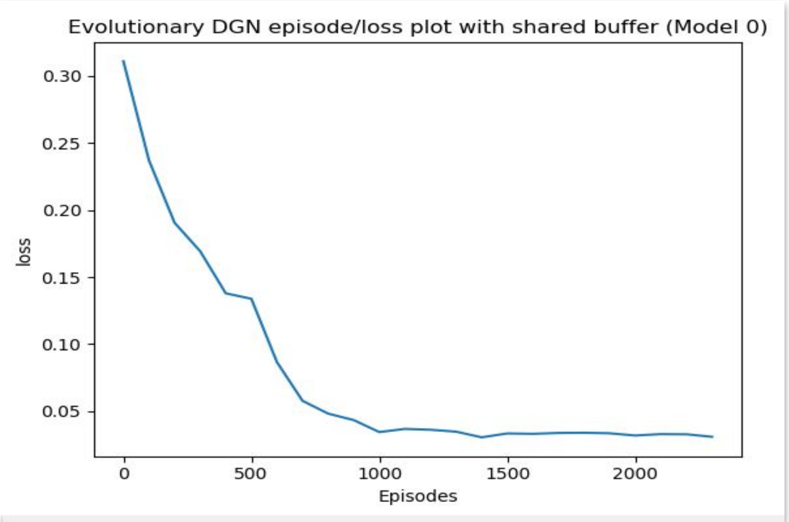
Loss change is less gradual- This is because of evolution

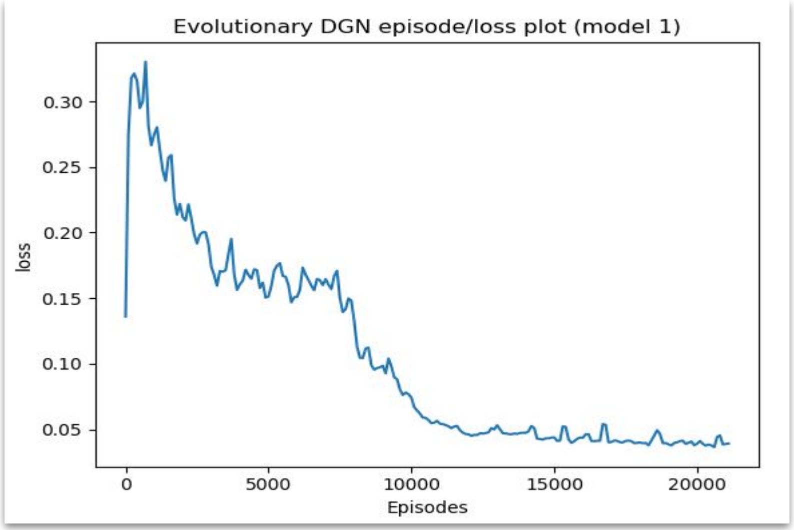
With “mutation,” we have no guarantees and so it stops learning around ~0.15 (result not shown)



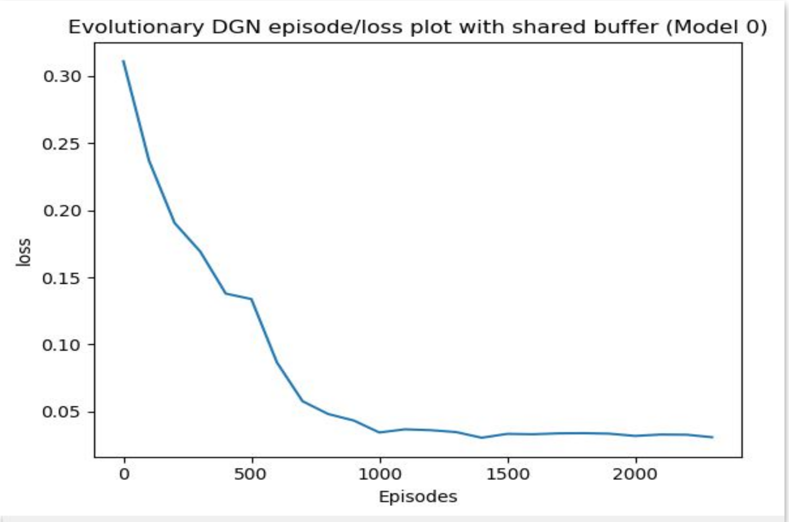


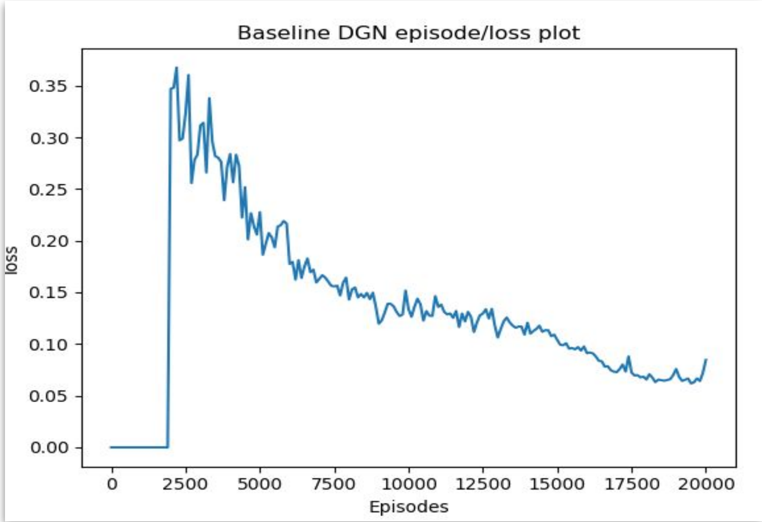
Results - evolutionary (improved) vs old





Results - evolutionary (improved) vs baseline





Conclusion and Future Work

- ★ Pro:
  - Faster 2000 eps * 5 agents vs 20,000 eps
- ★ Cons:
  - More memory: 5 agents -> 5 models stored
  - “Mutation” does not yield best results
- ★ More experimentation necessary
  - Training time is long, but possibly reaches more optimal solutions
- ★ Explore different hyperparameters
  - Or different variations of the algorithm
    - (here we only show best results of: no mutation with/without shared buffer)
- ★ Explore different number of nodes/routers
  - Or application to other areas (only to routers)