Förderstelle G-F-ORA Flatland LSTM

Diese Confluence-Seite enthält die Dokumentation und Arbeitsunterlagen zu meiner Förderstelle bei G-F-ORA September 2023-Februar 2024. Hauptziel war der Wissensaufbau im Bereich Reinforcement Learning anhand des Papers Multi-Agent Path Finding via Tree LSTM.

Unterseite	Was	Anschauen, wenn du
Verständnis des Papers	Notizen zu den theoretischen Grundlagen des Papers selber	▪ dich dafür interessiert, wie die Autoren des Papers Tree LSTM genau umgesetzt haben ▪ Genaue Beispiele für die Observations willst
Technische Überlegungen /Implementationsdetails	Notizen zu wie ich das Training zum laufen gebracht habe	▪ Tipps zum umsetzen eines RL Algorithmus suchst ▪ aus meinen Fehlern bei RL lernen möchtest ▪ mehr Infos zu verwendeten Packages möchtest
Flatland Training Experiments	Notizen zum eigentlichen Training	▪ sehen möchtest, wie ich das Training zum Laufen gebracht habe ▪ was ich alles bereits probiert habe (und was nicht geklappt hat) ▪ meine Learnings zum Training selber sehen möchtest
Transformer Tree Architektur	Notizen zu einer neuen Architektur für die Tree Embeddings	▪ einen alternativen Ansatz zu der im Paper verwendeten LSTM Architektur sehen möchtest

Experimente für Training des LSTM Netzwerkes

Generelle Learnings

Frameworks	▪ wenn immer möglich bereits bestehendes RL Framework verwenden (z.B. TorchRL). Auch wenn man das Environment noch etwas dafür anpassen muss, ist das gut investierte Zeit, da sonstige Fehler im Code unwahrscheinlicher werden.
Vorgehen	▪ http://karpathy.github.io/2019/04/25/recipe/ ein super Start ▪ Fokus auf minimalen Beispielen, die konvergieren sollten (bei Flatland: Gerade Strecke zum Bahnhof und ein Agent, der sich vorwärts bewegen soll)

Flatland Training Experiments

Eigenschaften der Experimente

- TensorFlow Experiment Name
- run command + curriculum
- kurze Beschreibung

Context: Log for experiments with goal to train the TreeLSTM used in Flatland in paper: 2210.12933.pdf (arxiv.org)

Fehler beim Ausführen des Makros 'toc'

null

Initialization

Linear layers are initialized with a mean of 0 and standard deviation of 0.01 (higher standard deviations lead to numerically unctractable outputs).

Single Track Setting

We just have a single track and one agent. As the agent will not stop once it has started moving, the only decision it faces is when to start moving. We incentivize moving by giving a reward of 1 when the train departs, and none when the train does not depart. Each game only has a max duration of 2.

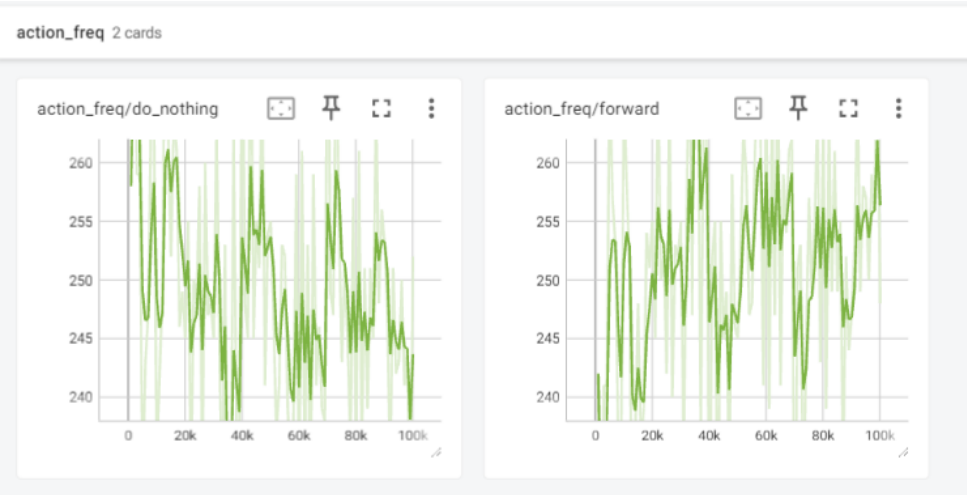
Learning Rate set to constant 2e-4

Results

Initialization works, both actions (forward and do nothing) occur with similar frequencies

rl__attempt_learn_to_start_initalized_std_01_add_loss_shares__1__170

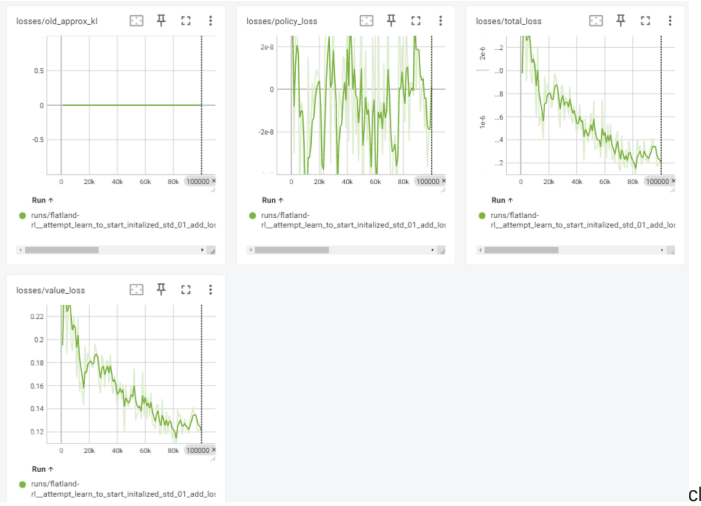
0127215



no clear change in the

frequency of either action

run_commands/1_20_fifty_agents_no_deadlocks_regularized_batch_size_2_000_clip_coef_0_2.txt



clear decrease in the value loss, but seemed to stagnate towards the end

Conclusion

- Make sure that the whole network can actually train, and the backpropagation is never interrupted, e.g. by using .numpy() on a tensor
- then the network converges very quickly and learns that it should go forward/not forward, depending on how the rewards are set

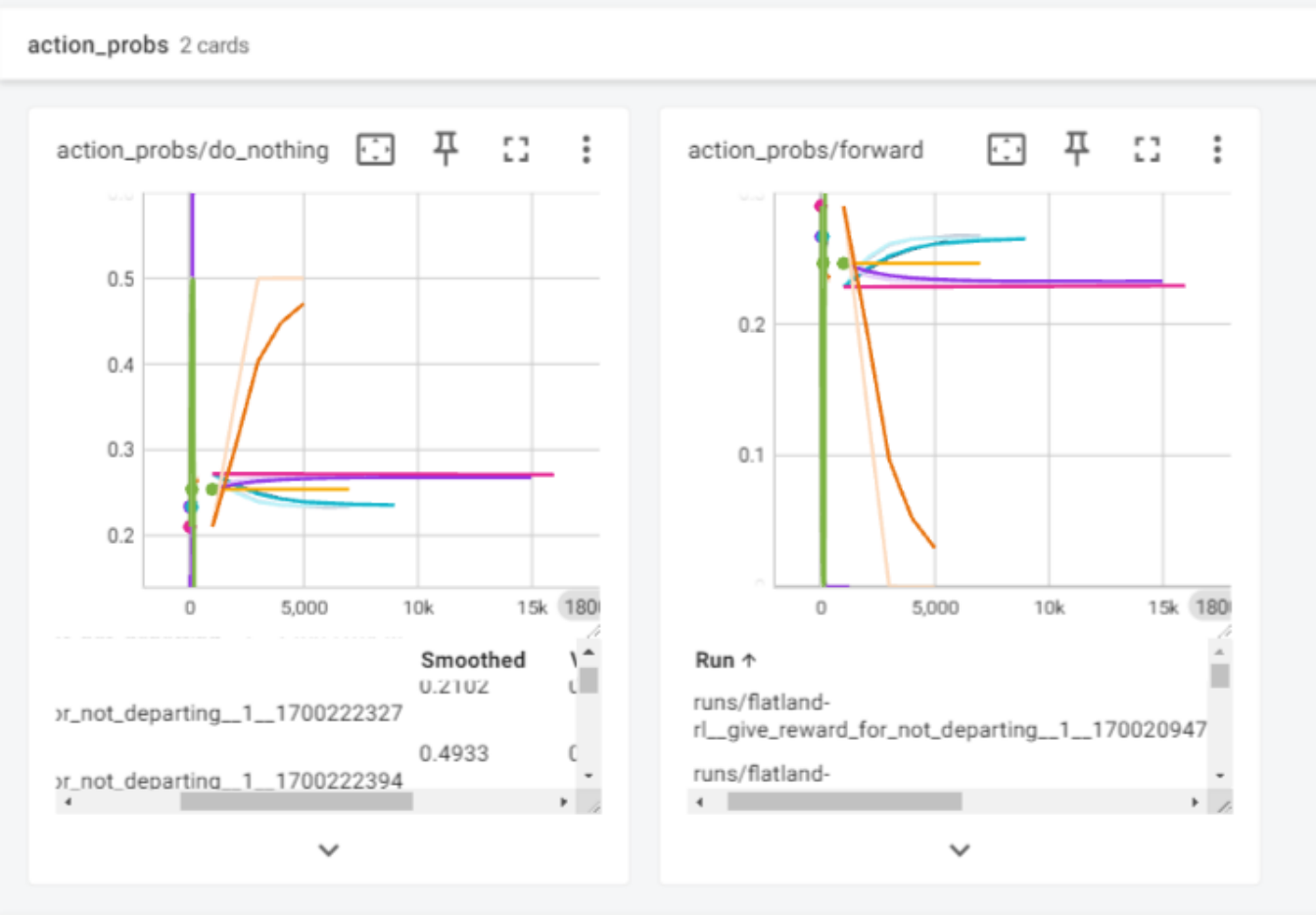


Figure of Eight

Setting

We give the agent a figure of eight, containing two stations and switches in both directions. The maximal game duration is set to 10, ensuring that the agent can reach the stations, but only if it choses correctly the first time it passes a switch. This way we avoid issues with actions in a dead end and a randomized map, while still forcing the agent to learn to chose to go left or right.

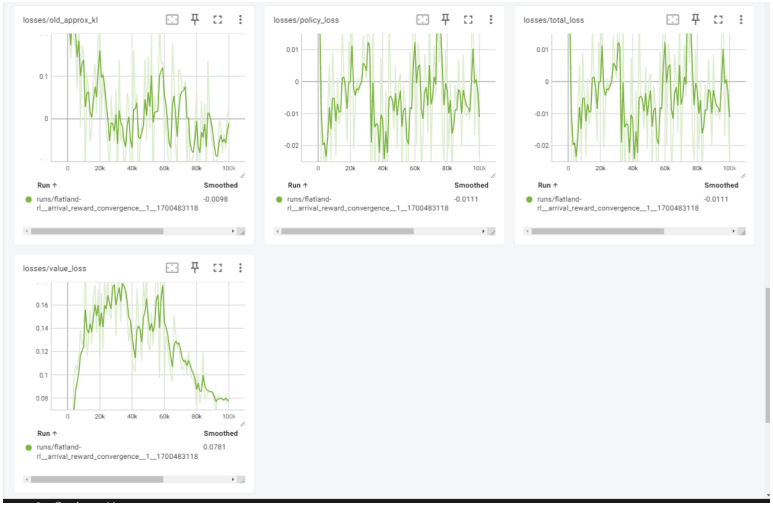
Initial Attempt

In an initial attempt, the agent learns very quickly to move forward (i.e. not stay still once it could start), and it consequently reaches the destination 50% of the time. Afterwards, the agent would have to correctly identify if it should go left or right based on the input features of the tree, which currently does not seem to work.

Idea: use the pretrained network on this example, see how it performs

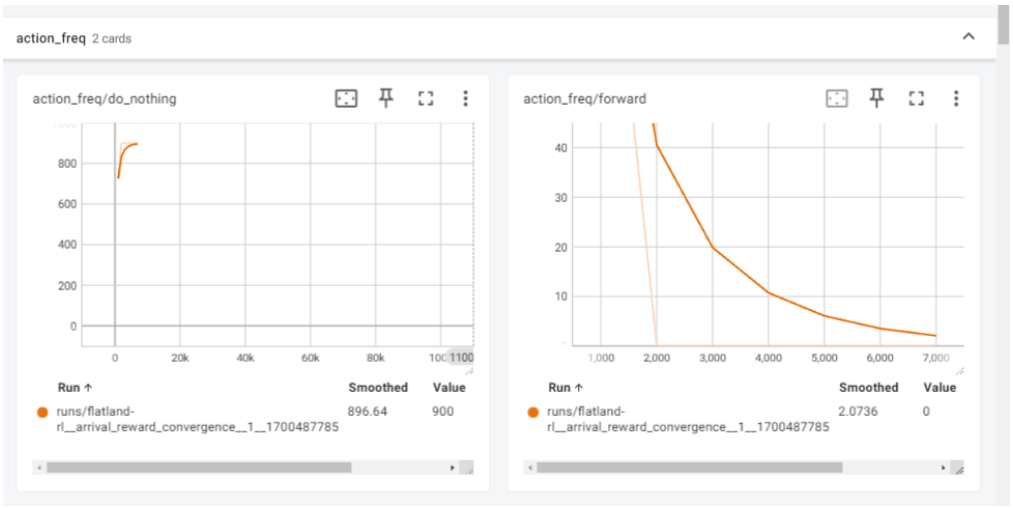
Different Options for the reward

Initially it was attempted to just give a reward of one when the train reached the target. Paired with the limited time for each run, the actor quickly learned to depart once the game starts (and not chose the option "do nothing") (runs/flatland-rl__arrival_reward_convergence__1__1700483118)



However the rewards quickly plateaud, suggesting that the model is learning to depart, but does not learn whether to turn left or right. Similar settings with this reward were tried with the pretrained network.

The pretrained network however is very likely to do nothing, and quickly converges towards just not departing: (runs/flatland-rl__arrival_reward_convergence__1__1700487785)

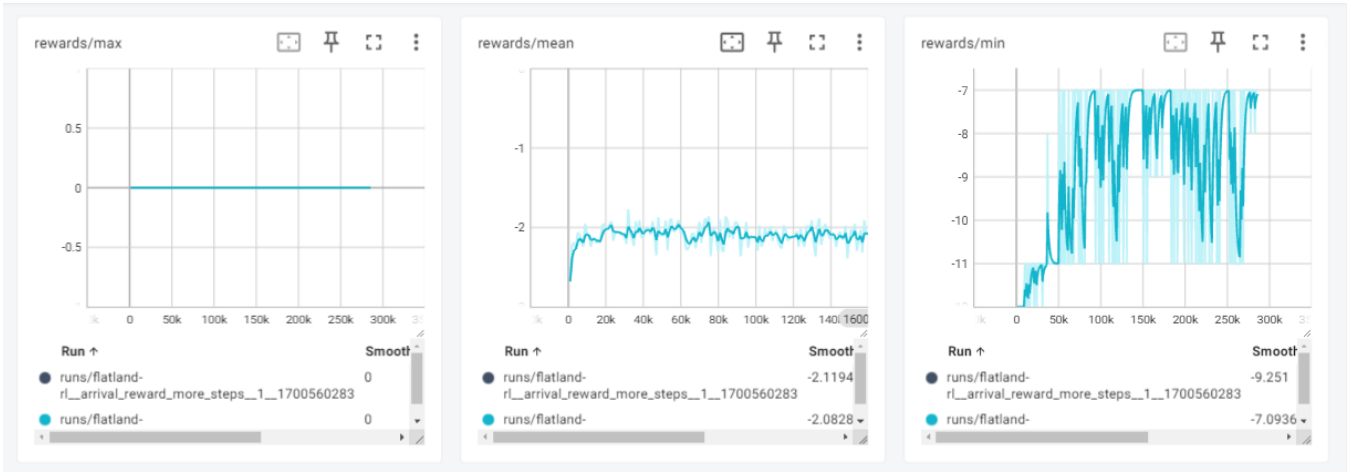


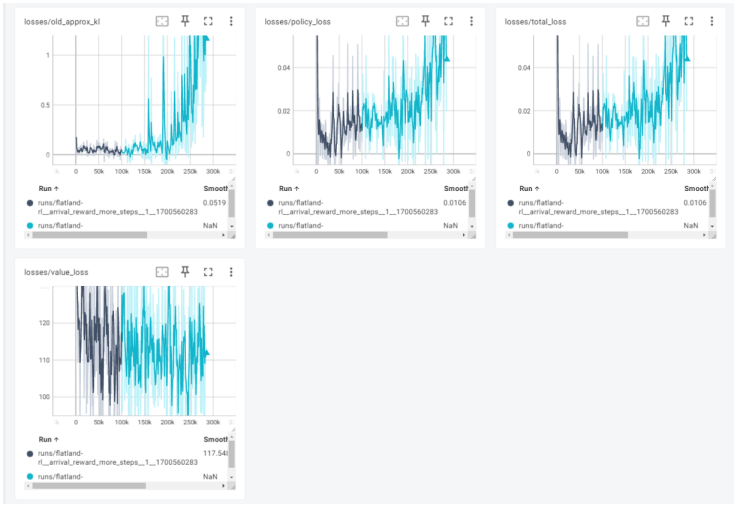
As a consequence, the delay reward (as used in the environmental reward in flatland) was used in the following examples. The motivation is that the delay has a jump-increase when the train makes the wrong choice at a switch (because it will have to take the substantially longer way round). This should provide more immediate feedback as to whether the choice at the switch was a good one.

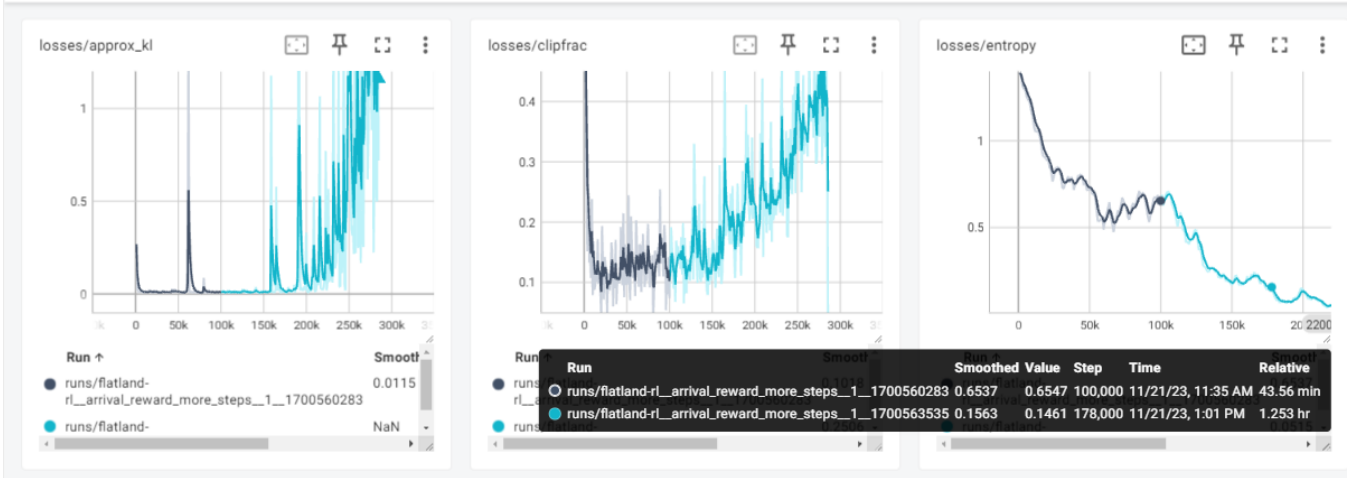
Using the embeddings of the pretrained network

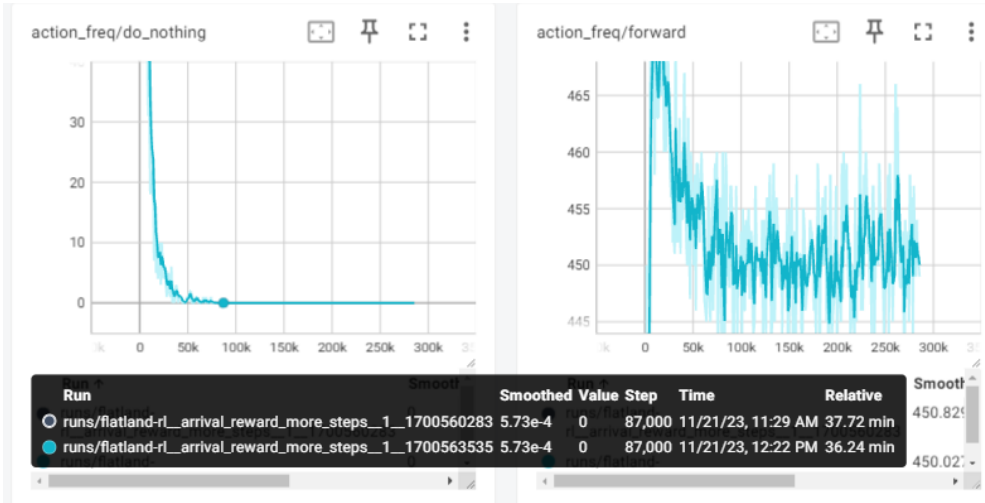
In order to quicken training, it was attempted to load the pretrained network and freeze the parameters, except for the actor and critic net. The hope was that the pretrained network provide useful embeddings, all while reducing the number of parameters that have to be trained. (runs/flatland-rl__arrival_reward_more_steps__1__1700563535)

- model ran for approximately 250'000 steps
- used frozen pretrained network for embedding and attention layers, actor and critic net were normally initialized and trained
- delay reward was given at each step
- lr 2.5e-4









Result

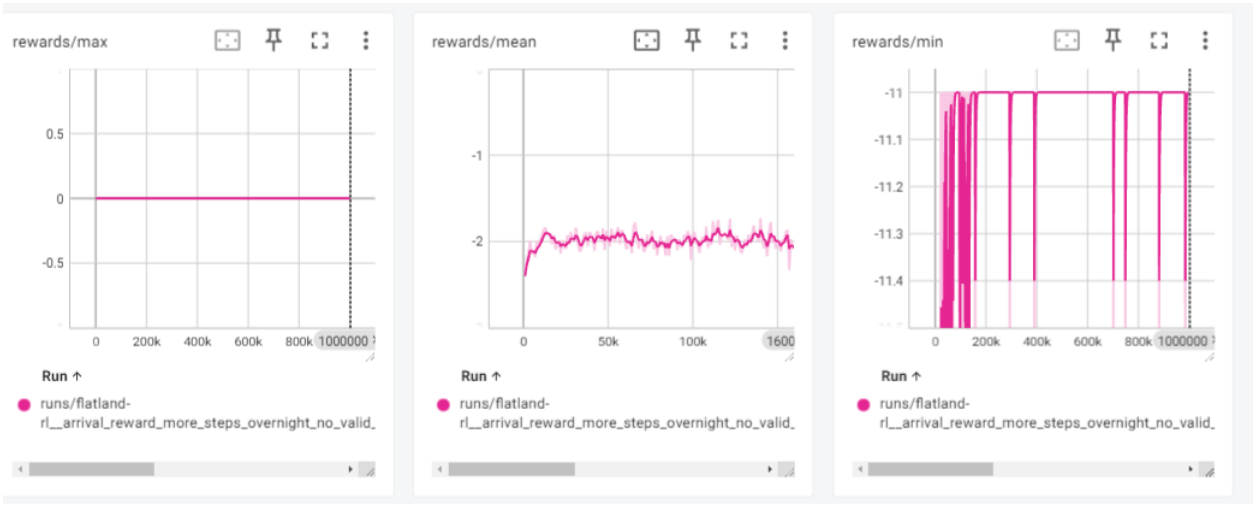
As before, the model learned quickly to not stand still at the beginning. There is however no evidence of the model learning when it should turn left right or go ahead. What strikes is the increasing policy loss towards the end, combined with the increasing KL divergence. It is unclear what this means.

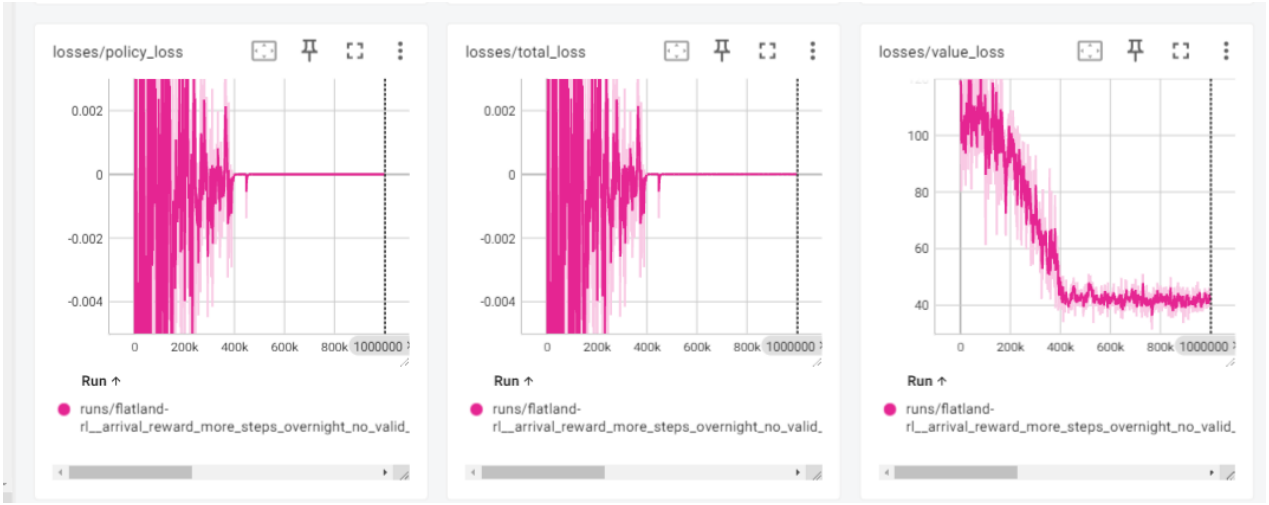
The training also stopped because of the all-zero-probabilities issue. investigate if we can find a way to avoid that

Longer runtime, no forcing to valid actions

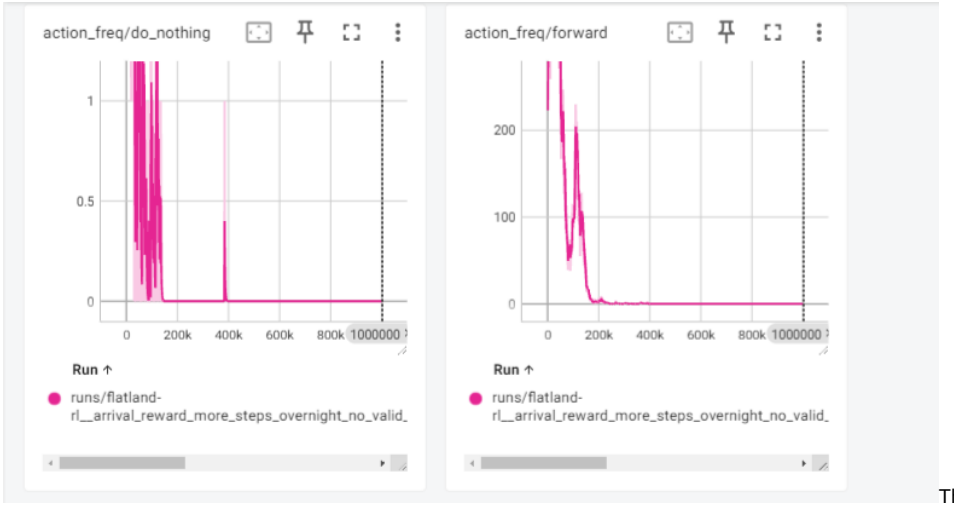
In order to see whether the training converges when running for a longer time, the goal was to train overnight for 1'000'000 timesteps. The following setting was used (runs/flatland-rl__arrival_reward_more_steps_overnight_no_valid_action_forcing__1__1700599483):

- don't force a returned action to be a valid action this avoid issues when the probability of all actions is forced to zero
- learning rate of 2.5e-4
- use delay reward at each step
- use figure of eight map
- initialize network from scratch (no pretrained parts)





The model seems to use the way flatland deals with invalid actions in a "smart" way. It starts to not depart at all:

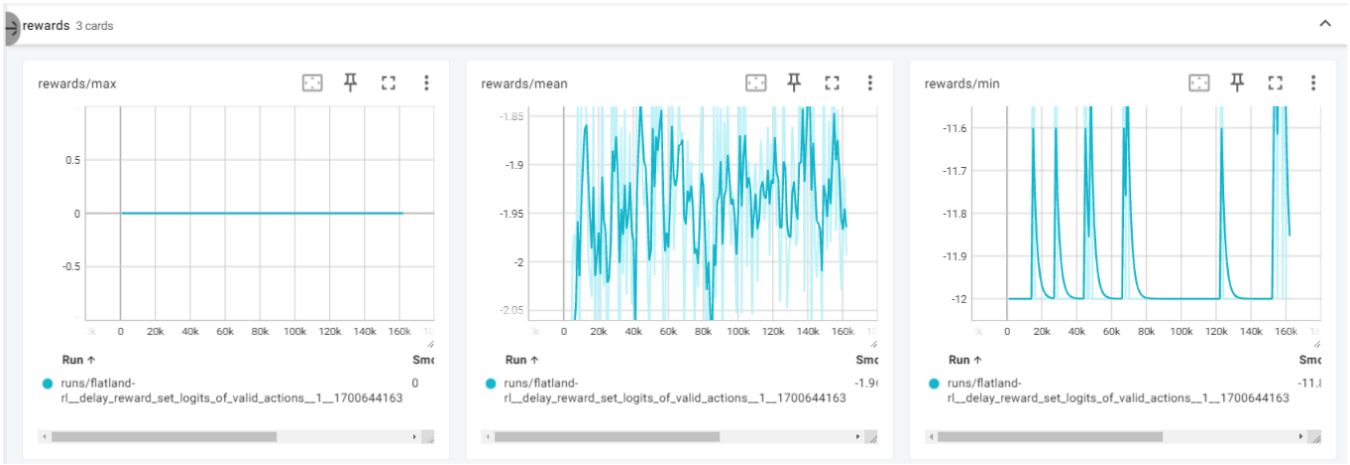


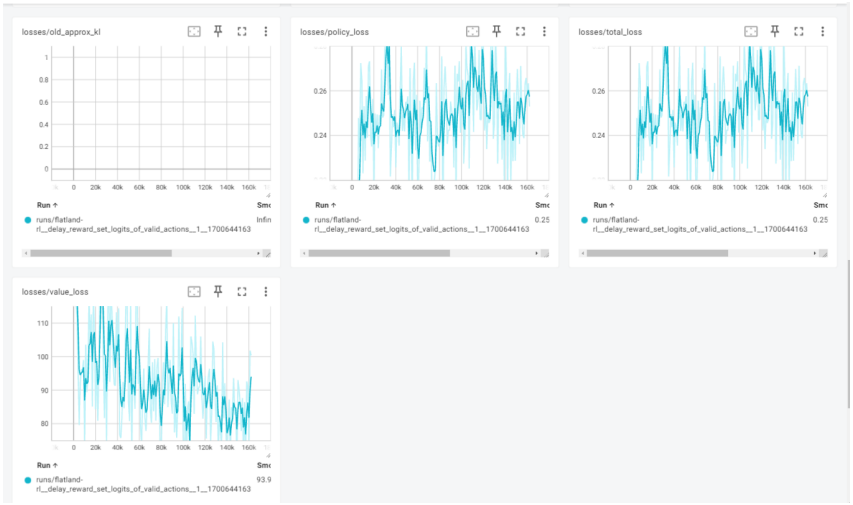
This apparently leads to much smaller variance in the loss.

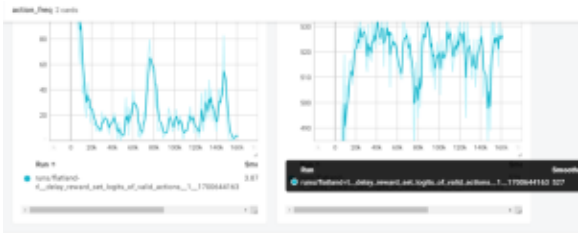
learnings:

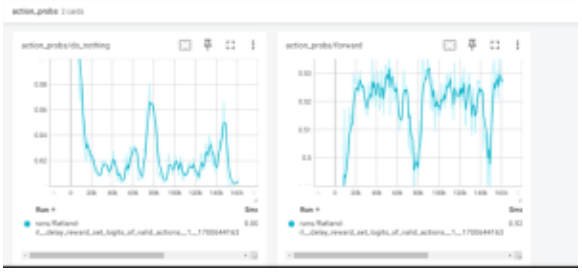
- log more information
- find better way to deal with valid actions check with how it was done in original solution

Changing the logit valid actions







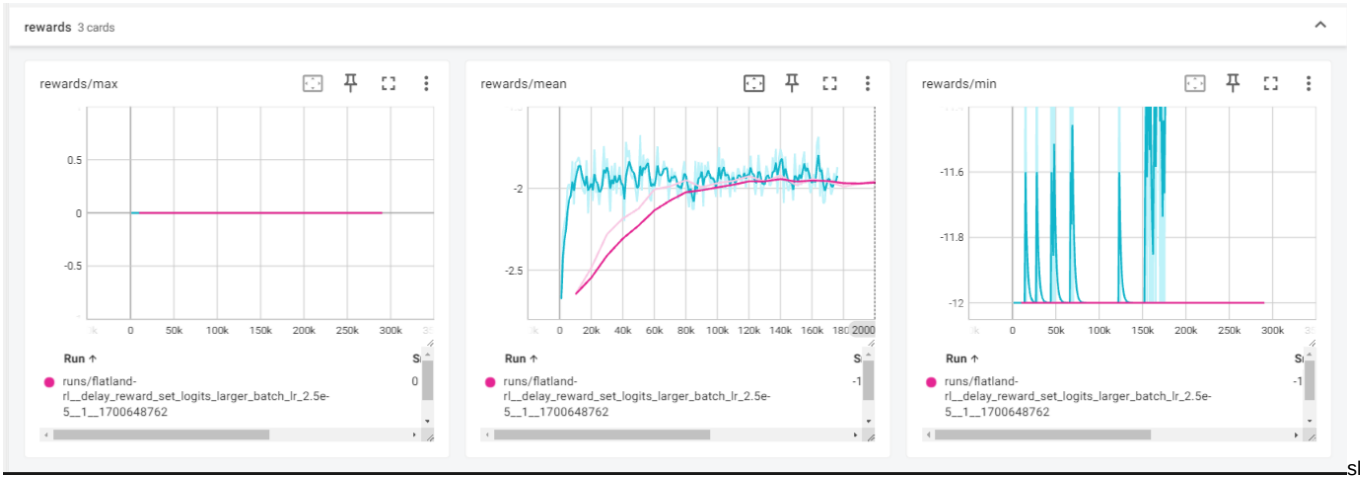


no clear change in behavior :/

Try larger batch sizes

runs/flatland-rl__delay_reward_set_logits_larger_batch_lr_2.5e-5__1__1700648762

- set the batch size to 10'000



slower and less jittery reward, but still the same general pattern

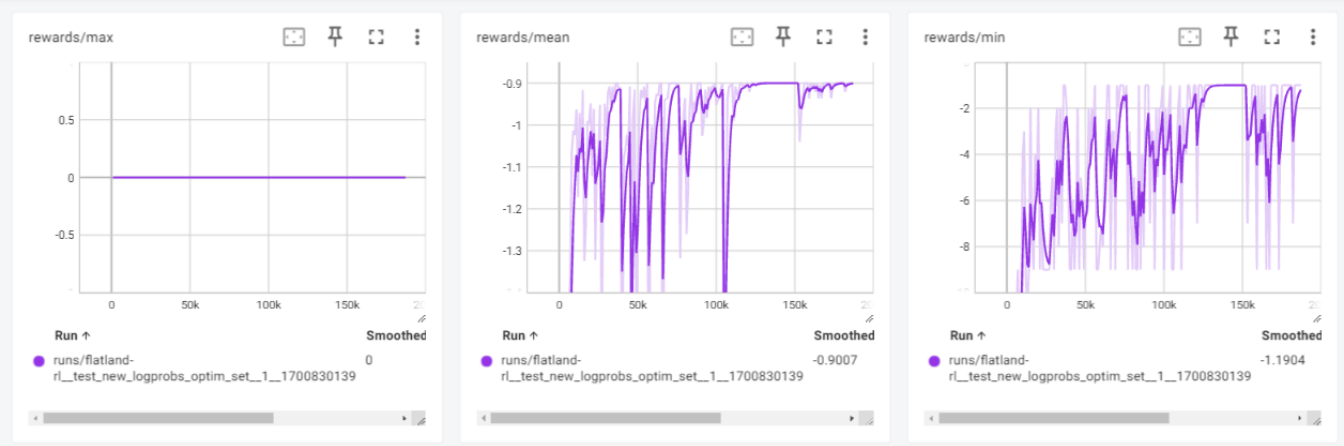
slower convergence, but similar patterns as before

why does the policy loss increase? Can we even achieve better policy loss than this?

After fixing log probs

An issue with the formatting of the chosen actions that were used in probs.log_prob(actions) caused log probs of invalid actions to be returned, which were consequently -inf and messed up the training. After fixing this issue, training converged in the following setting (runs/flatland-rl__test_new_logprobs_optim_set__1__1700830139)

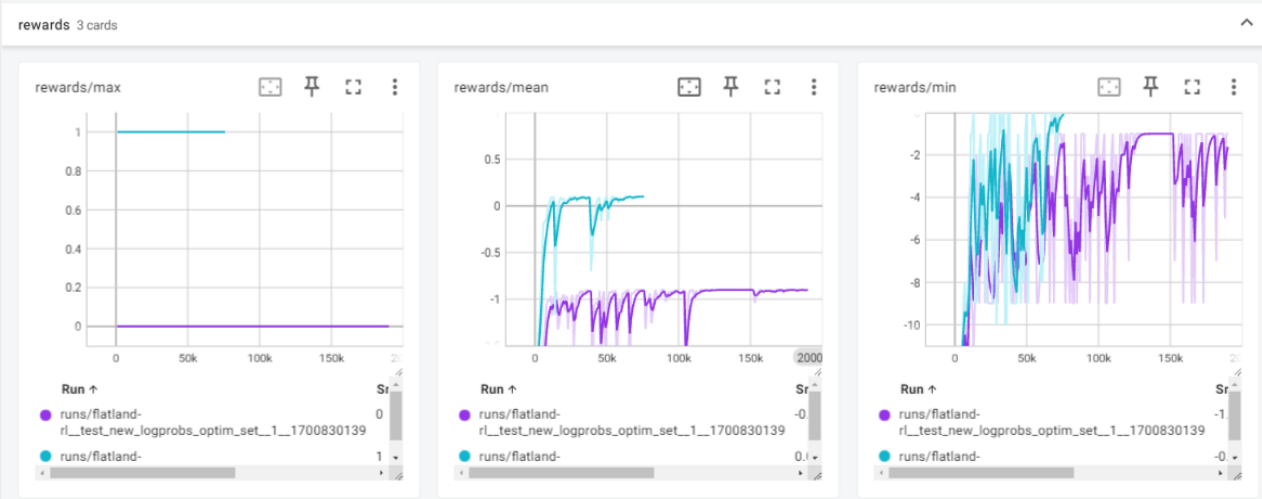
- using delay reward
- 1000 batch size
- 10 minibatches
- using pre-trained network for embedding and attention layers, frozen
- only training actor and critic net



Question: why does the reward stagnate at -0.9?

artifact of the definitions of delay with the first start, if we set time to 11 we get value slightly above 1, faster training convergence.

runs/flatland-rl__test_new_logprobs_optim_set__1__1700834531

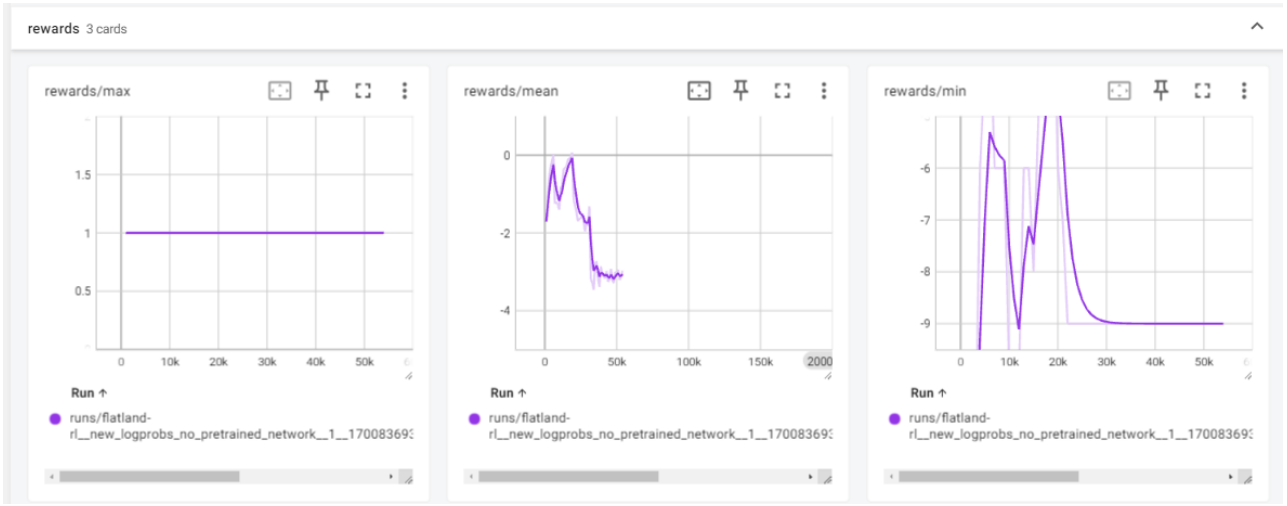


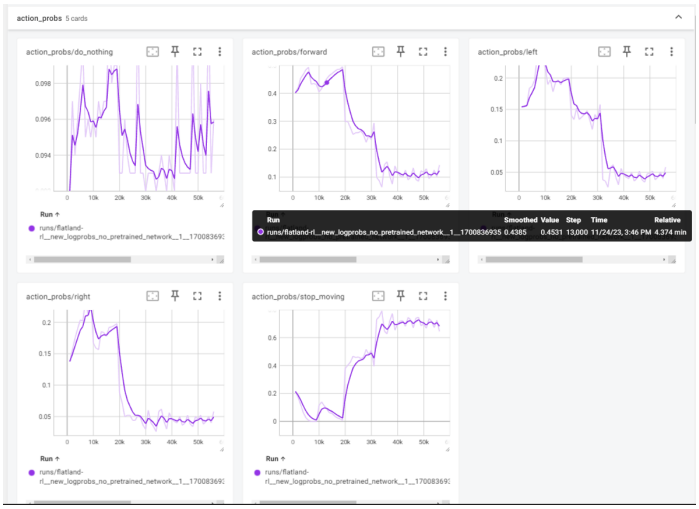
converges pretty quickly 😃

try different seed (so far seed1)

works for different seeds

Try training the whole network in same setting for the pretrained network



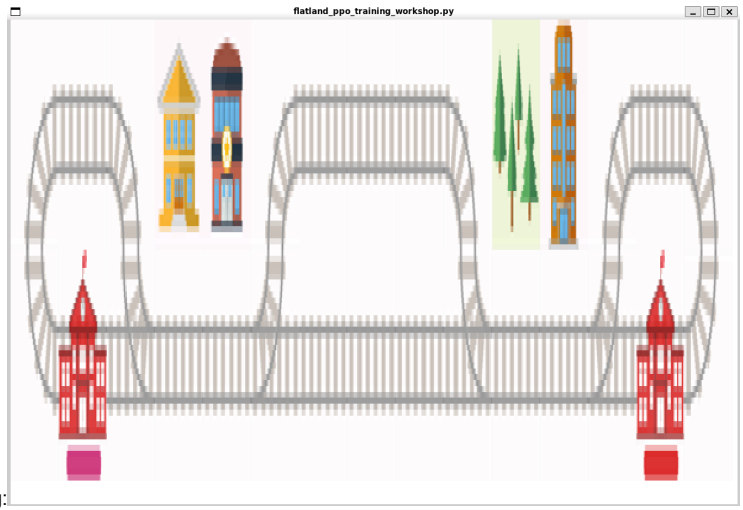


Model likes to start to not move

It might be much harder to learn to embed the trees in a smart when the variance in the data is very small due to the map being constant good candidate to try randomized map

Training a Train Meet

In order to start training for the multi-agent case, we use the following setting for getting trains to use a double track correctly for a meet.

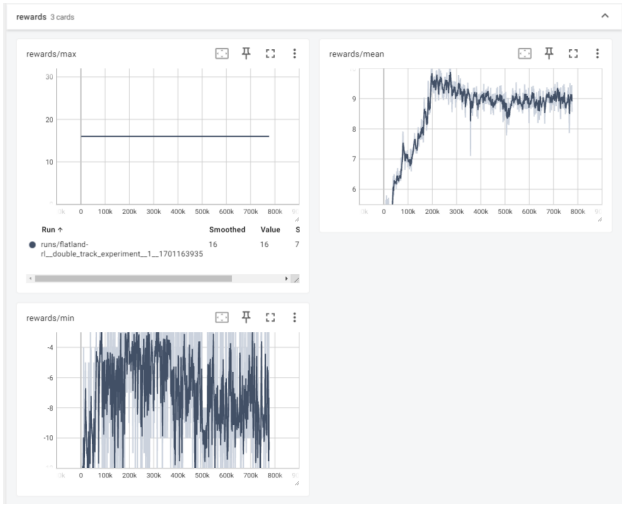
The layout used was the following: 

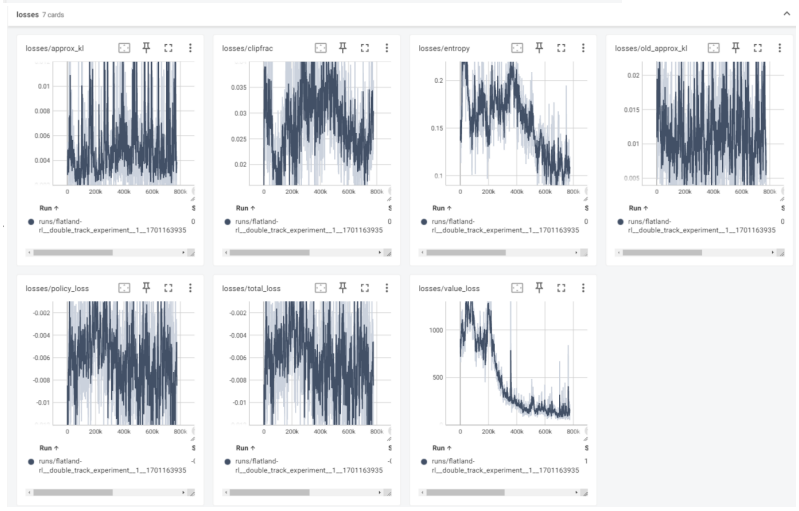
In an initial trial, the following setting was used (

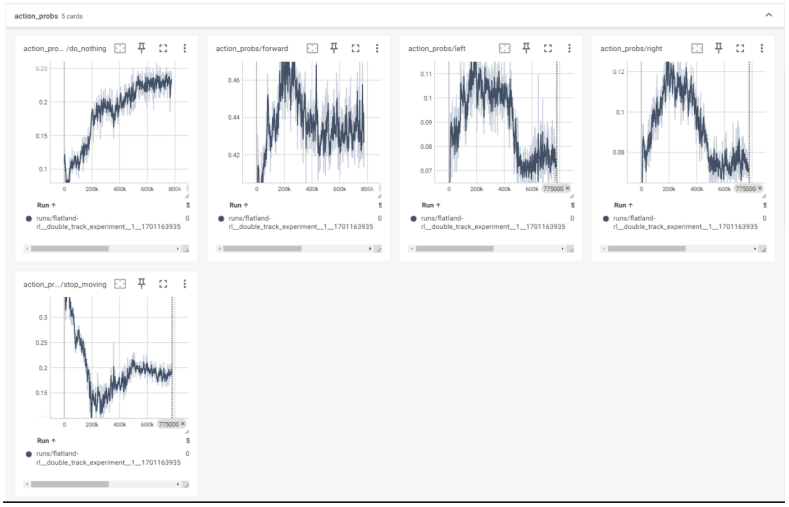
runs/flatland-rl__double_track_experiment__1__1701163935):

- delay reward
- 2 agents
- pretrained network with frozen embeddings
- meet maps
- max episode timestep 30
- 1000 batch size
- 10 minibatches
- lr 2.5e-4

Results







- the reward does improve quite a bit initially, which shows that the training loop works fundamentally
- the model does however not find an optimal solution. we would expect the minimal reward to be positive, because the trains should be able to reach their goals in time, given that they use the passing loop and wait before the single track sections are cleared
    - possible explanations: with the loops at the end of the track, the trains could also wait until the other train has cleared the single track sections, which means one train will have a positive reward, and one a negative reward, but the average reward is better than if both trains venture out on the single track and neither arrives on time
- possible future ideas: extend the lengths of the double track passing loop, while not increasing the episode duration that much. This would incentivize the traisn to both depart.
- We don't know if the suggestions for the different agents are based on their order, i.e. agent 0 always gets "move forward" and agent 1 always gets "do nothing"

Training a single Agent on a Random Map ("shortest path alg")

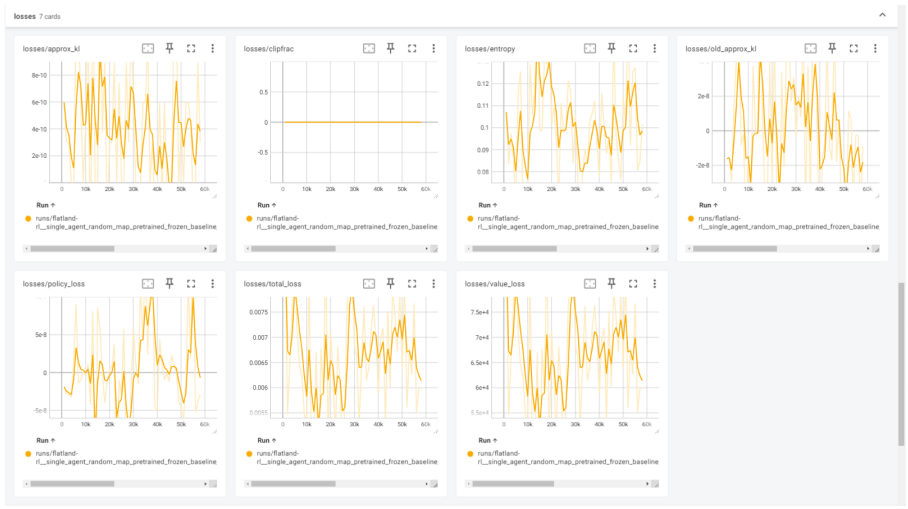
Reason for focusing on this setting: We have managed to train just the actor and critic network if we keep the embedding and attention layers fixed. Now the next step would be to try and train the whole network in a simple setting. For this, the working assumption is that the model needs to see a variety of data in order to train correctly we need a randomized flatland map.

Baseline with frozen pretrained model

Settings used (runs/flatland-rl__single_agent_random_map_pretrained_frozen_baseline__1__1701181112):

- random maps generated by marl paper default settings
- delay reward
- one agent
- pretrained network
- no training

Results



Average reward seems to be around 22

Training Attempts

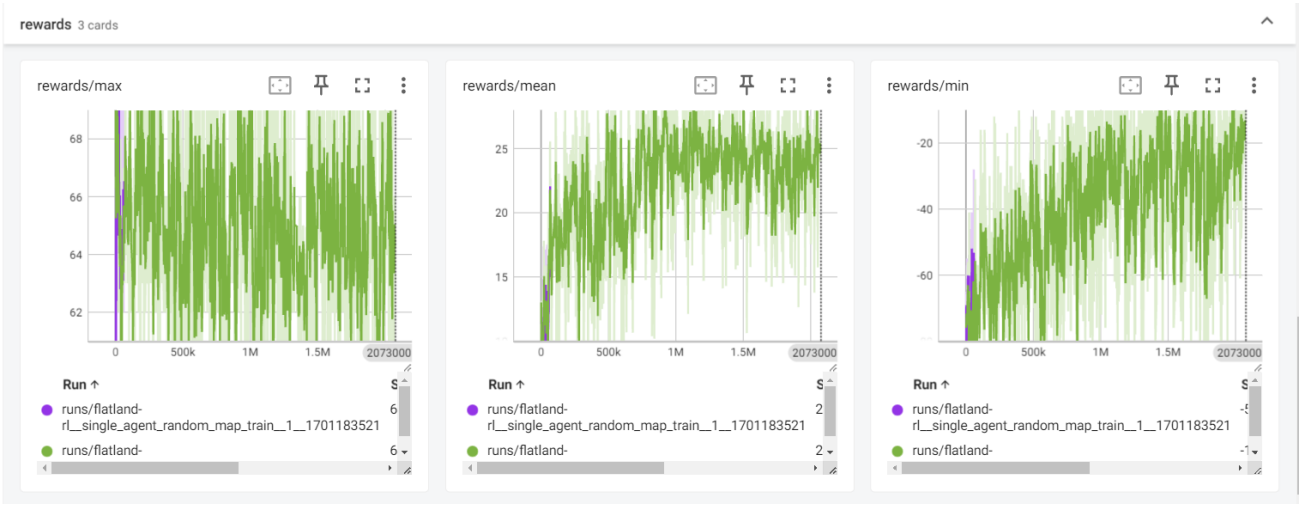
First attempt

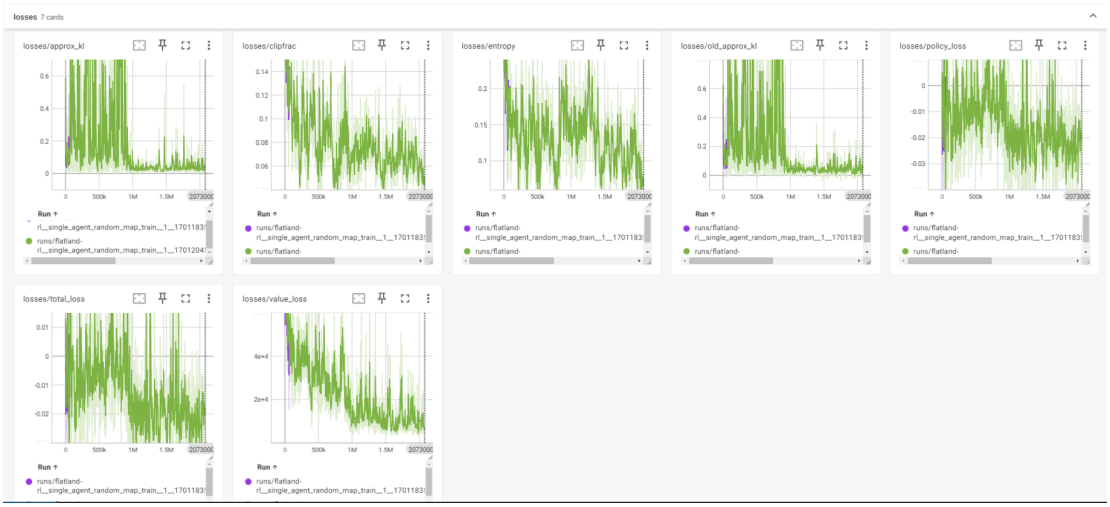
Settings used (runs/flatland-rl__single_agent_random_map_train__1__1701204519):

- random map as used in the demo example
- delay reward
- one agent

- lr 2.5e-4
- max_episodes_step of 80
- batch size 1000
- default random seed of 1

Preliminary Results





- the results look very promising, the mean reward ends up in an area comparable to the one received by the original model
- try adding the arrival rate as a logged
- will need to confirm with additional runs with different seeds

Multi-Agent Attempt with 5 Agents

Settings used (Settings used (runs/flatland-rl__single_agent_random_map_train__1__1701204519):

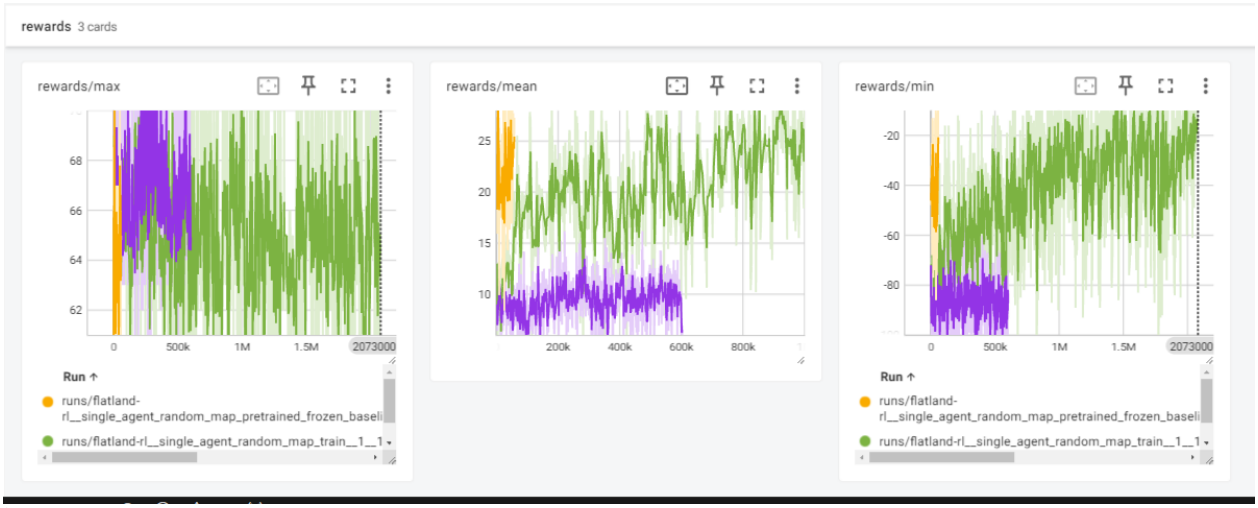
- random map as used in the demo example
- delay reward
- **five agents**
- lr 2.5e-4
- max_episodes_step of 80
- batch size 1000
- default random seed of 1

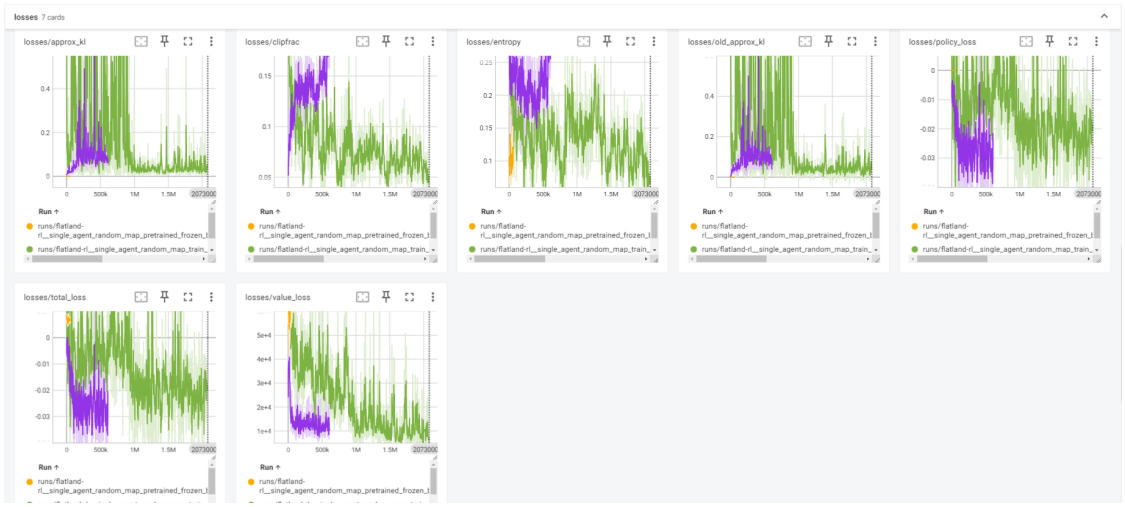
Results:

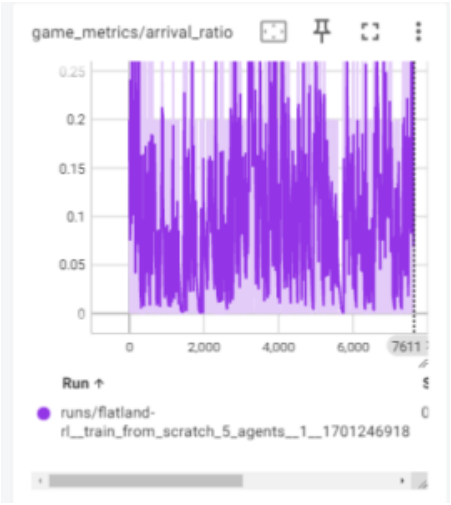
purple this experiment

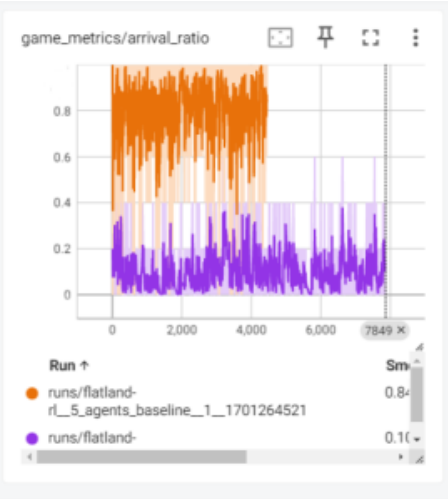
green single agent setting

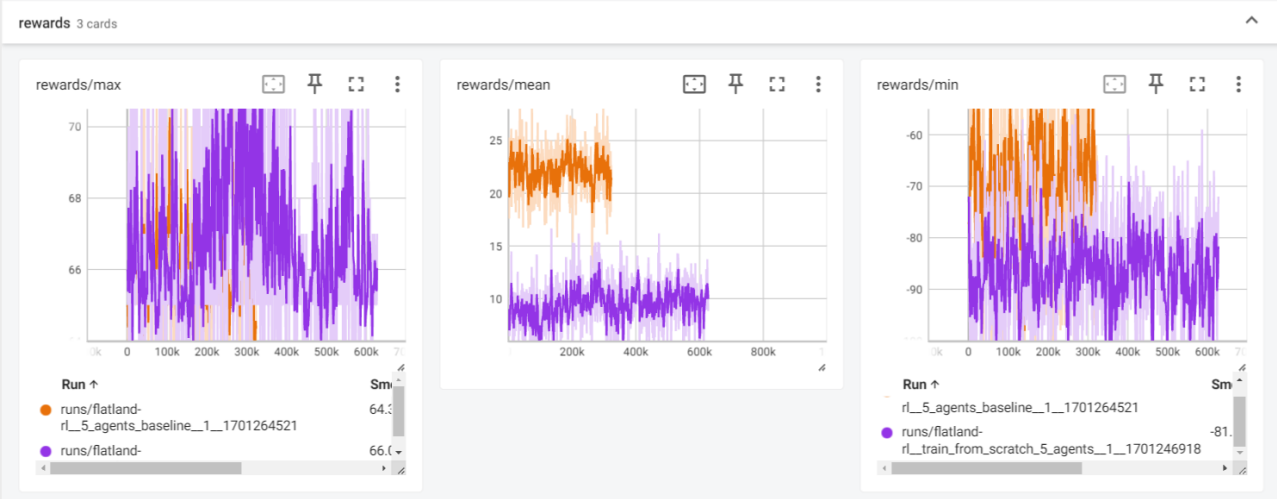
yellow singleagent baseline with pretrained network


rewards 3 cards
rewards/max
rewards/mean
rewards/min
Run
runs/flatland-rl__single_agent_random_map_pretrained_frozen_baseli
runs/flatland-rl__single_agent_random_map_train__1__1


losses 7 cards
losses/approx_kl
losses/clipfrac
losses/entropy
losses/old_approx_kl
losses/policy_loss
losses/total_loss
losses/value_loss


game_metrics/arrival_ratio
Run
runs/flatland-rl__train_from_scratch_5_agents__1__1701246918





Even after some training, the model does not have start closing in on the baseline (runs/flatland-rl__5_agents_baseline__1__1701264521)

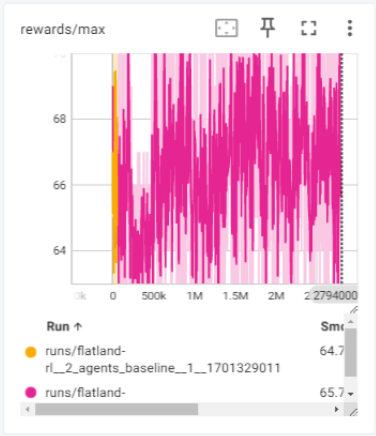
Multi-Agent attempt with two Agents

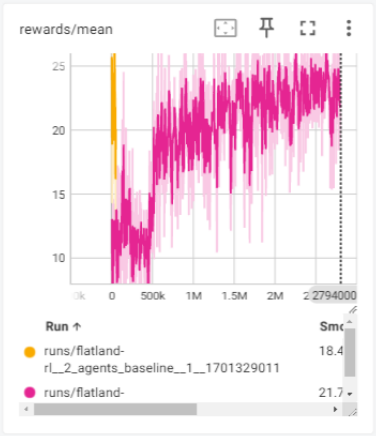
In order to determine whether there were issues with the correctness of the algorithm, or if the 5-agent setting just didn't manage to converge due to insufficient time or reaching a local optimum. The following setting was used (runs/flatland-rl__2_agents_from_scratch__1__1701274527):

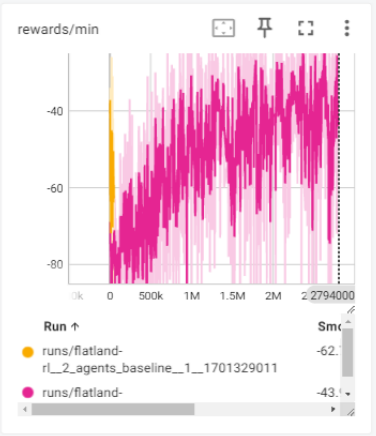
- random map as used in the demo example
- delay reward
- two agent
- lr 2.5e-4
- max_episodes_step of 80
- batch size 1000
- minibatch size 100
- default random seed of 1

runs/flatland-rl__2_agents_from_scratch__1__1701274527

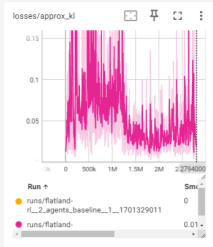
rewards 3 cards

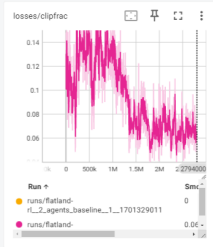


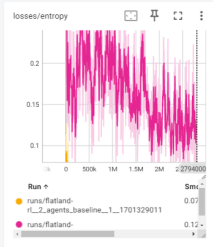


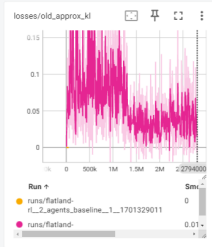


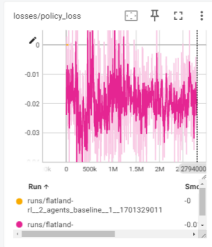
losses 7 cards

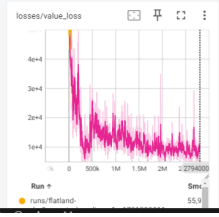




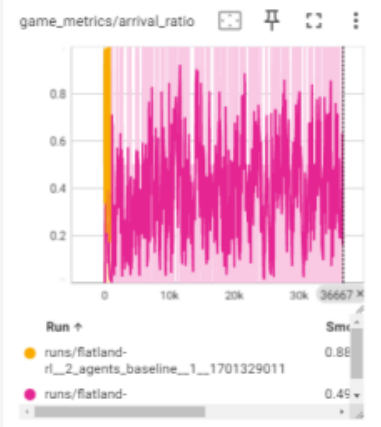








game_metrics



Results

- The reward clearly increases and reaches a level comparable with the pretrained network. This shows that the model does train and increase the reward.
- The arrival ratio is small compared to the pretrained model. This can be the result of different rewards, as the pretrained model used explicit rewards for arriving, while in our model, trains might just go close to the destination and consider that good enough.
- Generally, the pretrained model is more likely to wait and do nothing after the game starts, which could be a strategy for dealing with multiple agents, where we don't want all agents to depart at the same time.

Comparison of TorchRL and CleanRL implementations

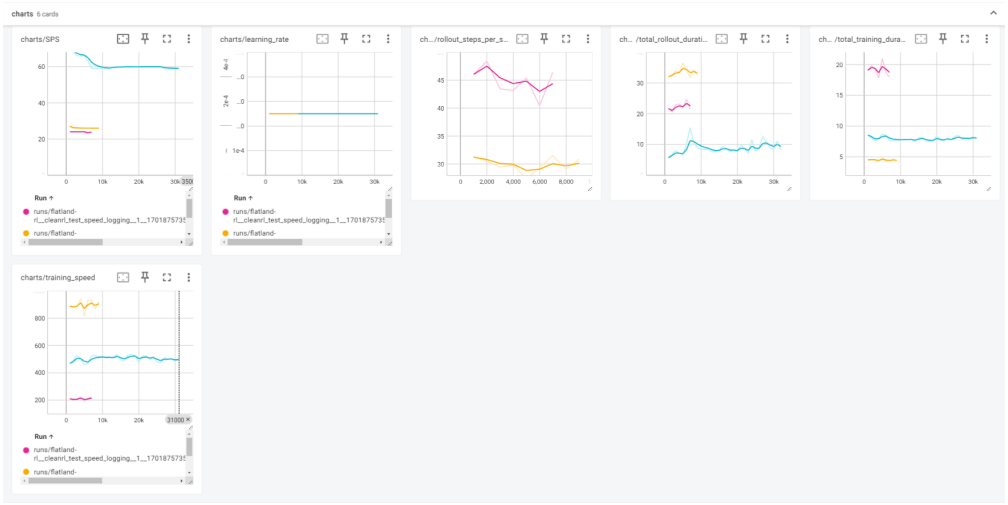
In an effort to speed up rollouts, the use of parallel environments was explored. The framework used was TorchRL, which offers a parallel env wrapper ParallelEnv — torchrl main documentation (pytorch.org) for its environment. After adapting Flatland to suit TorchRL, this makes the parallelization of environments easy. The GAE calculations could not be done using TorchRL's tools, because it does not allow for dynamically masked tensors - these however are an integral part of the Flatland TreeeLSTM implementation. Therefore, the GAE calculation of CleanRL was adapted (vectorized over the different batches for the envs at the same time).

For some unknown reason, it was not possible to have the policy network on the gpu during the rollout phase, so the optimal TorchRL training emerged as the following: Conduct the rollout of the data in parallel in multiple environments on the CPU, including the passes through the network. Afterwards, send the collected data to GPU and run the training loop there, including the passes through the network. Then update the policy network on the CPU with these changes, and start a new rollout.

Results of speed comparison between TorchRL and CleanRL versions

Settings used:

- 10 agents
- batch size of 1000 (i.e. 1000 steps are played in flatland in total, either consecutively, or in parallel in 8 environments)
- n_agents 10
- 4 epochs
- 10 minibatches



We can see that the blue TorchRL version is more than twice as fast in the overall than the CleanRL versions. This is largely due to the faster rollout speed. Between the two CleanRL version, one can observe a trade-off between training speed and the rollout speed, with a run on CPU being faster for the rollout, but slower for the training, and vice versa.

Overall, the TorchRL version allows similar training speeds for 10 agents as are achieved for 2 agents in the CleanRL setting. This is significant enough to merit further exploration.

runs/flatland-rl__cleanrl_test_speed_logging__1__1701875735

runs/flatland-rl__cleanrl_test_speed_logging__1__1701875735

runs/flatland-rl__torchrl_test_speed_logging_2_agents__1__1701878452

runs/flatland-rl__torchrl_test_speed_logging__1__1701875055

TorchRL debugging

There were several issues with the initial TorchRL version, that prevented it from performing comparably to CleanRL. Among them were:

- masking of unavailable actions: The original model from the paper only gets to decide among the actually available actions, i.e. the unavailable actions have to be masked. This had to be implemented in TorchRL. At the moment, this is implemented by manually setting the logits of unavailable actions to -inf in the network, but MaskedCategorical — torchrl main documentation (pytorch.org) would be another option.
- InteractionType of probabilistic actor: The probabilistic actor, which controls the decisions made in the rollout, has an argument "default_interaction_type". The default setting for this is "mode", i.e. the argmax for a categorical distribution. By setting it to InteractionType.Random, the model actually gets to randomly explore the map.
- Advantage normalization: The advantages were of a completely different order of magnitude in TorchRL compared to CleanRL by setting the normalize_advantage parameter to True in the loss_module, this was remedied.
- Advantage shape: For the losses to be computed correctly, the advantages need to have the same shape as the actions (as advantages are multiplied element-wise with the logprobs of actions), and with a trailing dimension of 1. This was fixed with the line `tensordict_data[("agents", "advantage")] = tensordict_data["advantage"].repeat_interleave(args.num_agents).reshape(tensordict_data[("agents", "action")].shape).unsqueeze(-1)`

Results of Trials after these changes

With the above modifications, the model showed consisten convergence in the single-agent setting with several random seeds.

Settings used:

- learning rate 2.5e-4
- 8 envs, 125 steps per env
- 10 minibatches
- 4 epochs
- normalized advantages, 0.2 clip coef
- vf coef of 0.01
- max grad norm of 0.2 still converged pretty well, but also still got the NaN issue
- max episode steps 80

Runs:

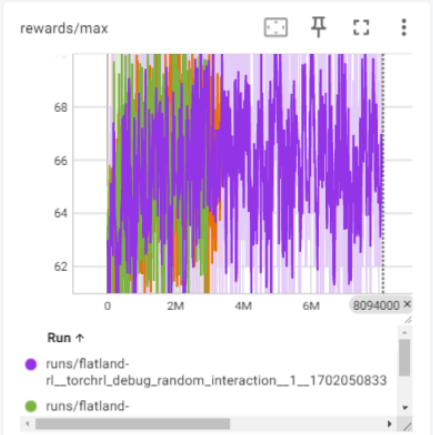
runs/flatland-rl__torchrl_debug_random_interaction__1__1702050833

runs/flatland-rl__torchrl_debug_random_interaction__2__1702129352

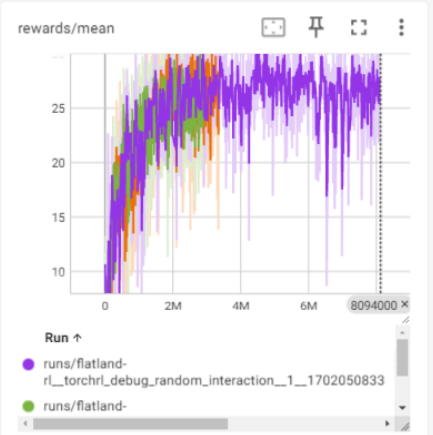
runs/flatland-rl__torchrl_debug_random_interaction__3__1702194741

rewards 3 cards

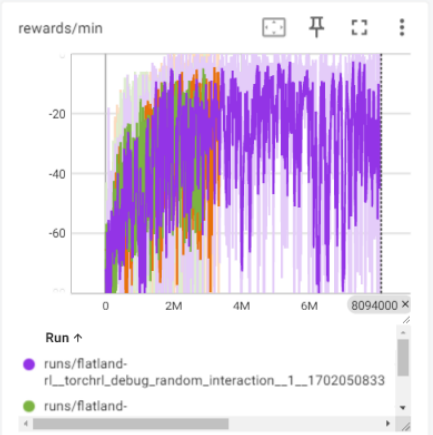
rewards/max



rewards/mean

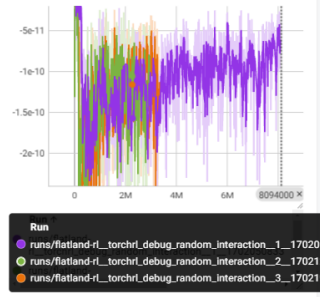


rewards/min

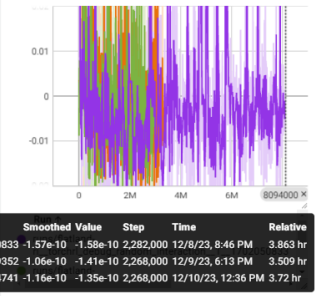


losses 4 cards

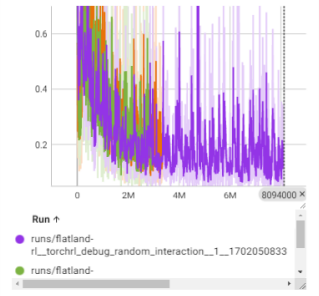
losses/entropy



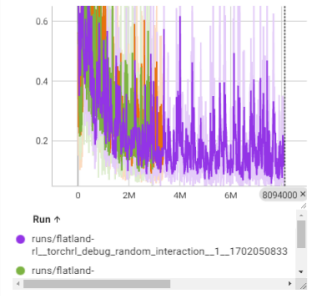
losses/policy_loss



losses/total_loss



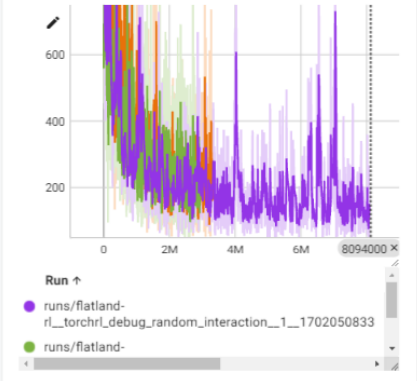
losses/value_loss



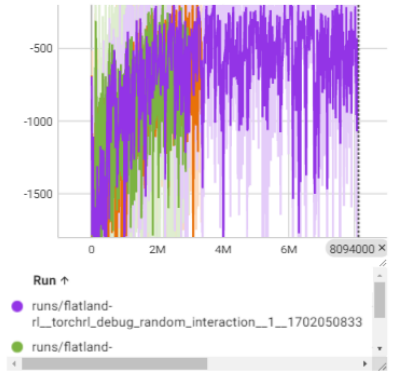
Run	Smoothed	Value	Step	Time	Relative
runs/flatland-rl__torchrl_debug_random_interaction__1__1702050833	-1.57e-10	-1.58e-10	2,282,000	12/8/23, 8:46 PM	3.863 hr
runs/flatland-rl__torchrl_debug_random_interaction__2__1702129352	-1.06e-10	-1.41e-10	2,268,000	12/9/23, 6:13 PM	3.509 hr
runs/flatland-rl__torchrl_debug_random_interaction__3__1702194741	-1.16e-10	-1.35e-10	2,268,000	12/10/23, 12:36 PM	3.72 hr

internal_values 2 cards

internal_values/advantages_max

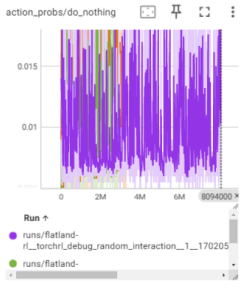


internal_values/advantages_min

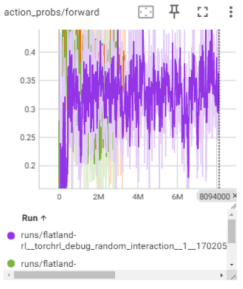


action_probs 5 cards

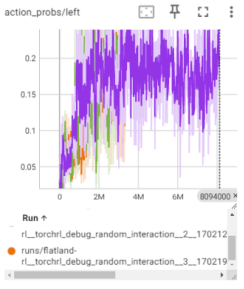
action_probs/do_nothing



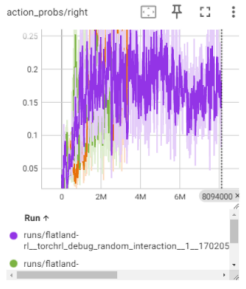
action_probs/forward



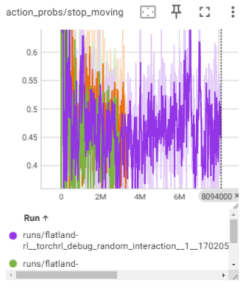
action_probs/left



action_probs/right



action_probs/stop_moving

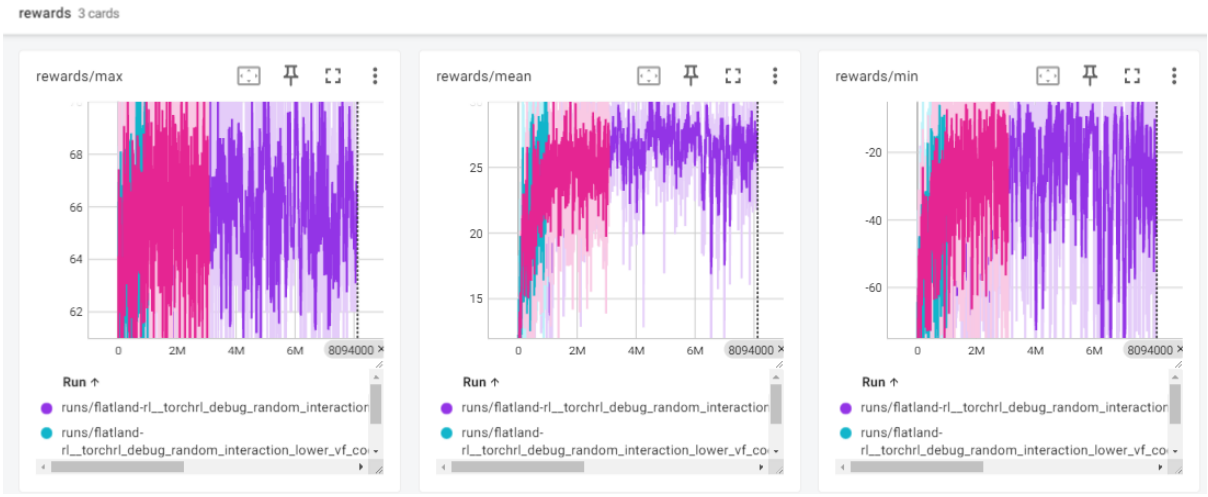


Conclusion

- TorchRL now converges
- High Value Loss is not an issue, model can still converge, even if it choses action "do nothing" exclusively for a brief period in beginning

Variant: Lower VF Coefficient

A version of the above setting with a vf coefficient of 0.0001 instead of 0.01 was tried and converged at similar speeds for two different seeds. Hence, the training does not seem to react very sensitively to the vf coefficient value.



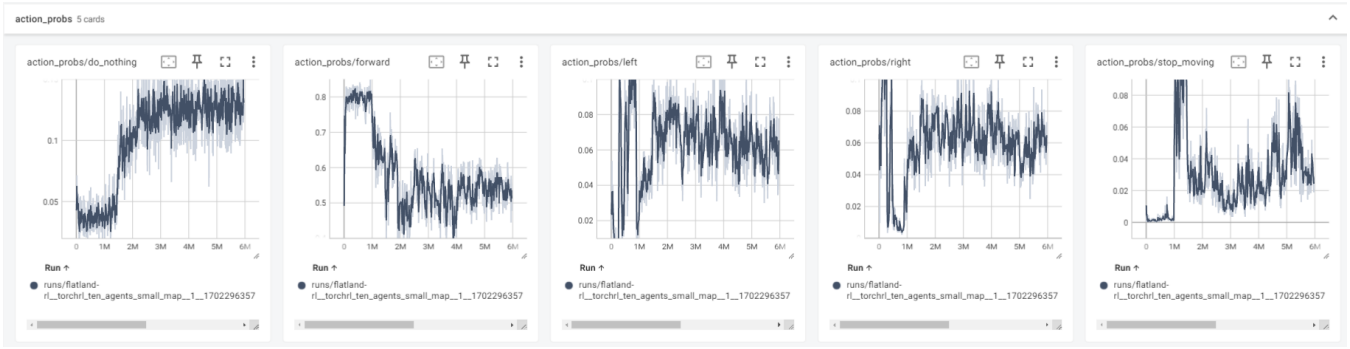
Experiment with ten agents

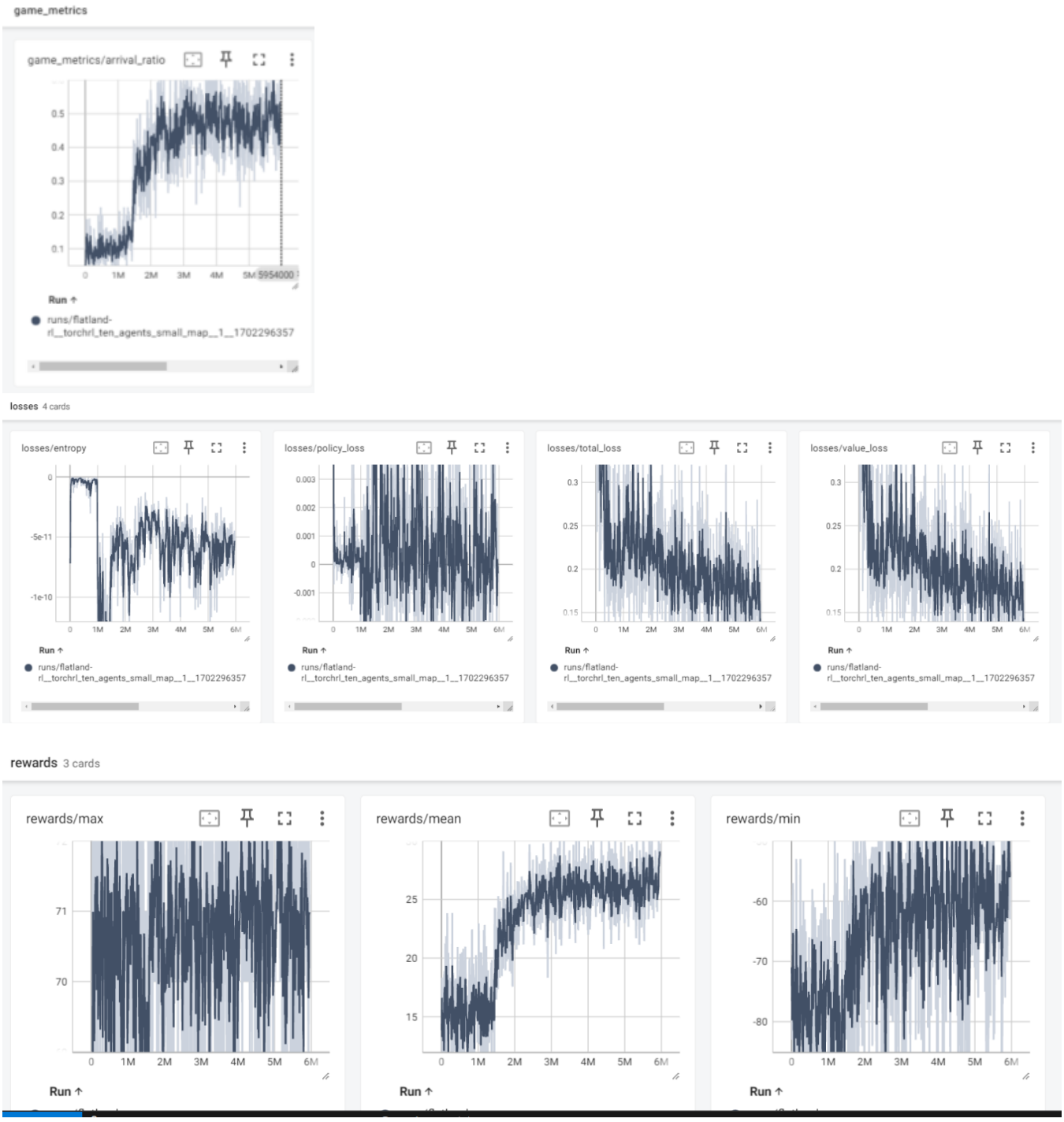
After the successfull run with 2 agents, a run with ten agents was tried in the following setting (runs/flatland-rl__torchrl_ten_agents_small_map__1__1702296357):

- capped max episode length to 80 (possibly lowering arrival ratio)
- 8 envs, 125 steps per env
- map height and width of 25
- vf coef 0.01
- max grad norm 0.3

Thanks to the parallelized rollouts, it was still possible to train about 80 game steps per second.

Results





Conclusion

What is interesting is the relatively long period in the beginning where nothing visibly happens, and it remains unclear if the model actually learns something. It turns out, it does, and the performance can increase quite drastically even after nothing happens for quite some time. Also the NaN issue did not pop up, even though training ran for almost six million steps.

Open Questions

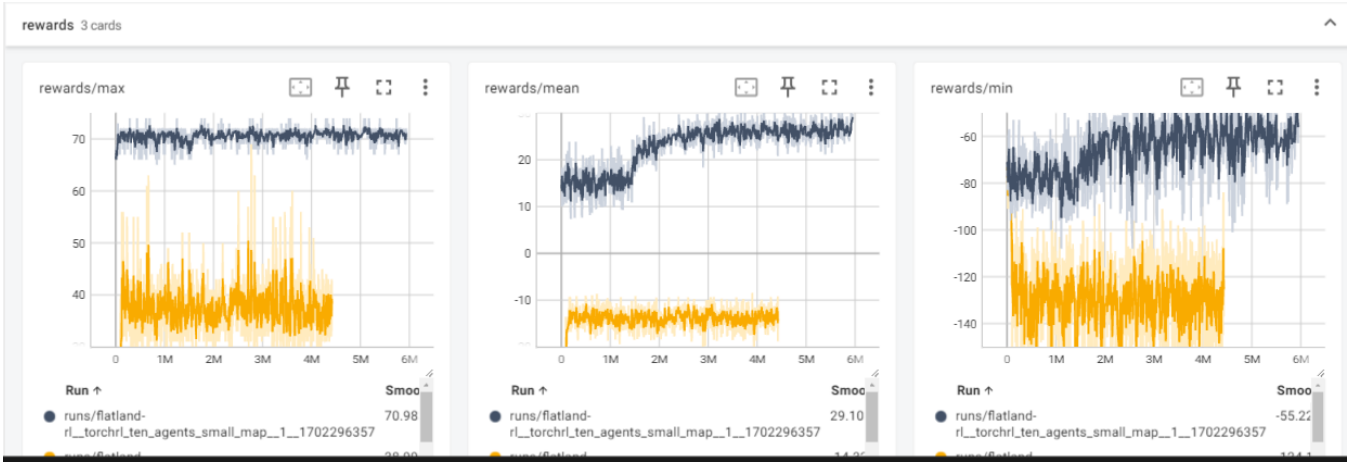
- What does this sudden jump in rewards correspond to in actual behavior of the trains? What do they learn? How can we investigate that?
- Are there any settings that allow us to reach this tipping point faster? as the training in the single agent setting shows a very similar behavior, it could be an idea to do hyperparameter tuning on single agent setting, see if anything works
- What kind of behavior are we promoting with the current reward structure?
    - The general flatland reward does not give positive rewards for early arrival (arriving earlier isn't better)
    - giving the env reward after each step will heavily promote shortest path learning, whereas we don't care about agents taking the shortest path, but only arriving on time (maybe a detour is beneficial globally)

	#agents	Reward Weights				Initialized by
		c_e	c_a	c_d	c_l	
Phase-I	50	1	5	0	2.5	N/A
Phase-II	50	0	5	1	2.5	Phase-I
Phase-III-50	50	0	5	1	2.5	Phase-II
Phase-III-80	80	1	5	0.1	2.5	Phase-III-50
Phase-III-100	100	1	5	0.1	2.5	Phase-III-50
Phase-III-200	200	1	5	0.1	2.5	Phase-III-100

-
- Indeed, env reward is discarded at some point in paper

Changes in Rewards

A second run with 10 agents was conducted, with the rewards being set exactly like in Phase-I in the paper and longer episode runs. However, in this case, no convergence was observed runs/flatland-rl__torchrl_ten_agents_small_map__2__1702397159.



try another run with delay reward at all times

Another reward structure

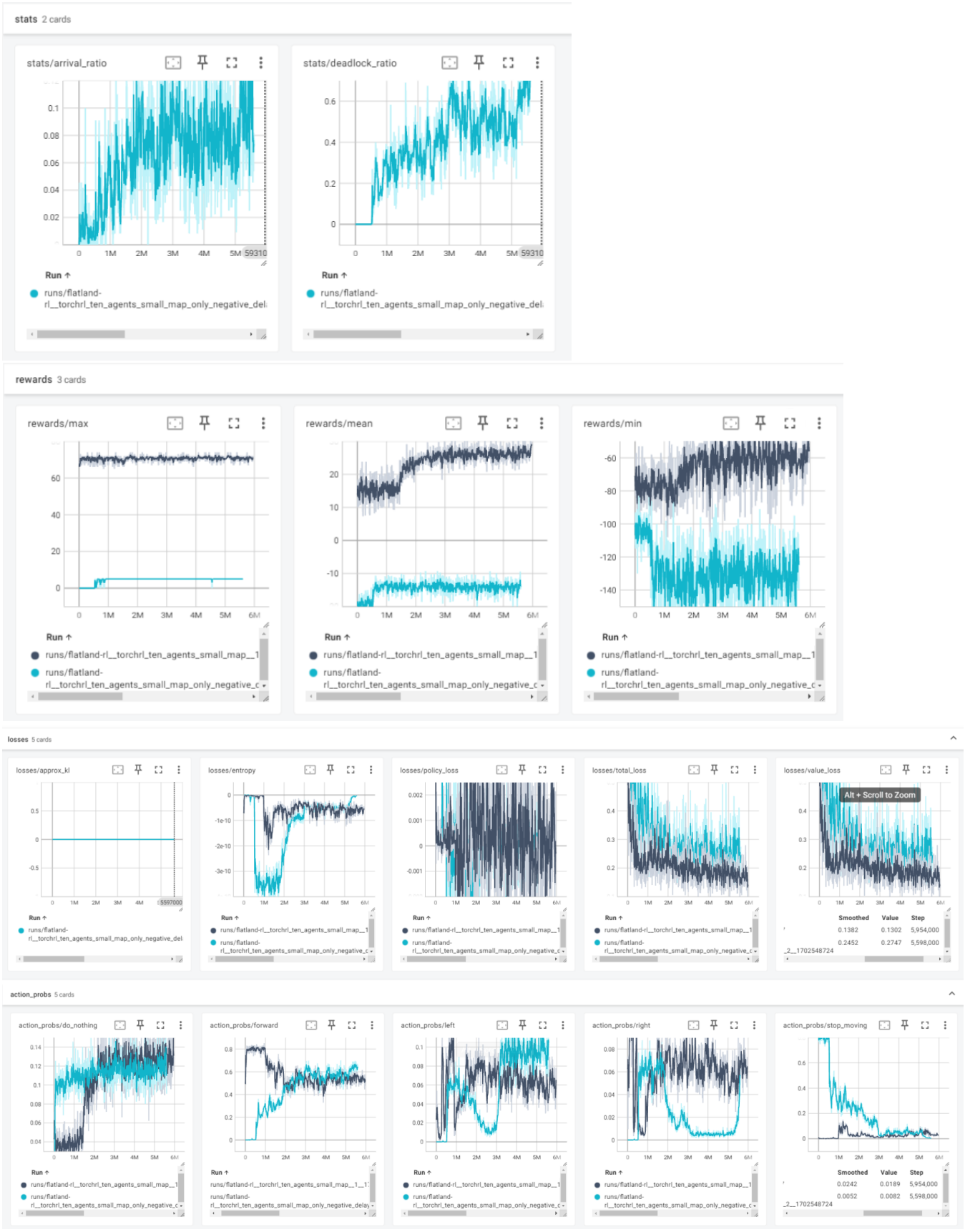
runs/flatland-rl__torchrl_ten_agents_small_map_fixed_rewards__2__1702459236

- env reward
  - if agent is done
    - give minimum of delay, arrival time
  - if agent is in prgress
    - give delay, but only if on map state (should change this to: if agent could already depart)
  - if agent never departed, give reward 0

reward structure like this allows model to just not depart, which is not good, as it gives a reward of 0, rather than a penalty for delay

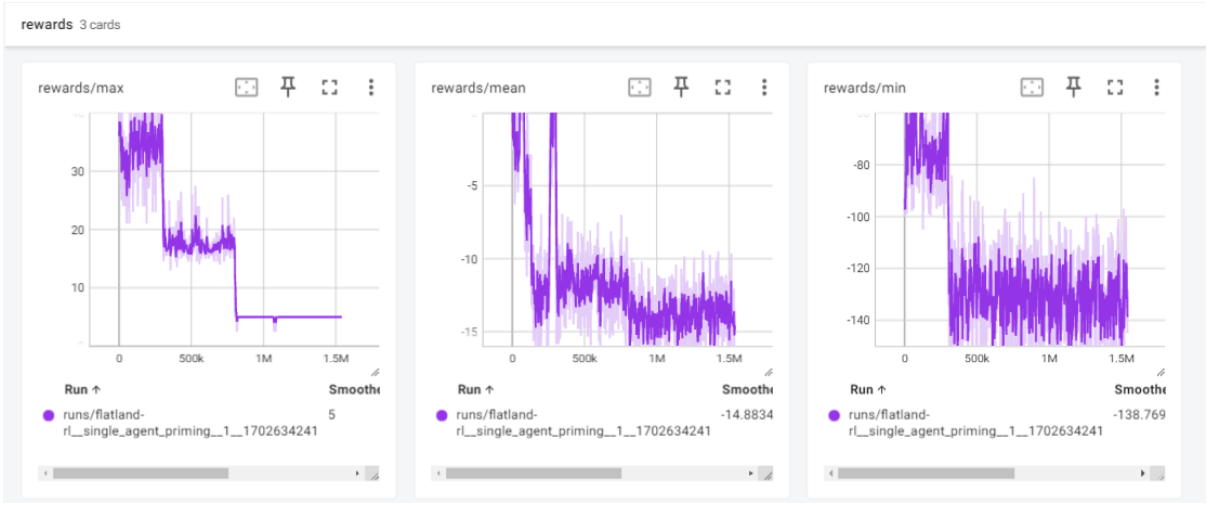
Run: runs/flatland-rl__torchrl_ten_agents_small_map_only_negative_delay_rewards__2__1702548724

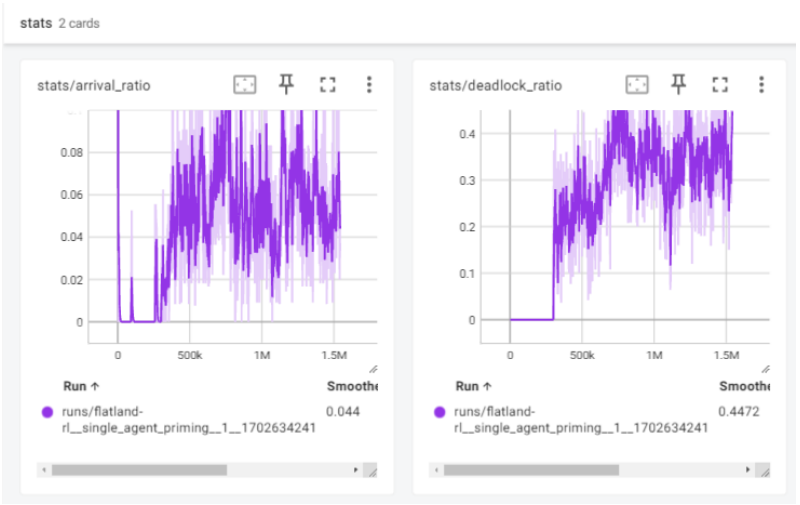
- only give the delay reward (i.e. 0 if the agent is "ahead of schedule")
- this apparently makes it harder to find the shortest paths, no convergence after 5mio steps

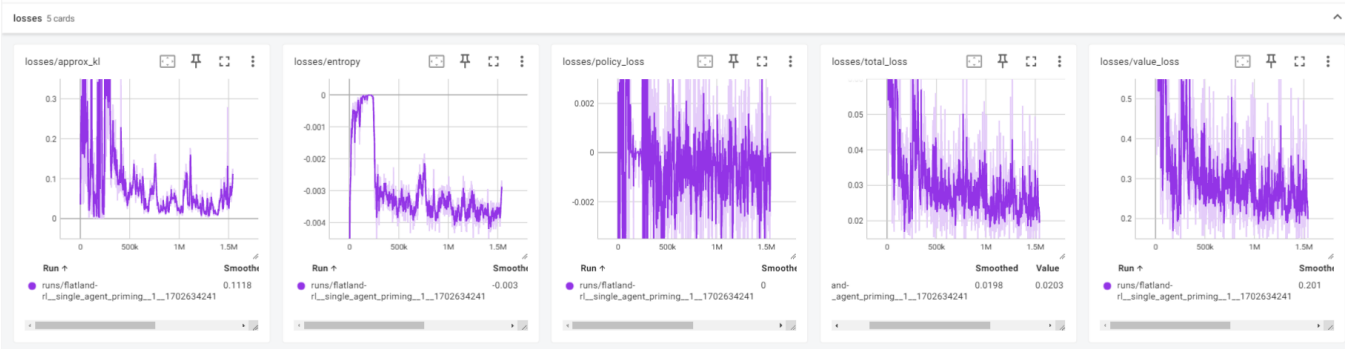


run: runs/flatland-rl__single_agent_priming__1__1702634241

- using curriculum single_agent_shortest_path.json
  - give a start with 300'000 steps single agent, then switch to 10 agents
  - ent coef of 0.01
  - vf coef of 0.01
  - max grad norm of 0.5
  - lr 2.5e-3







No convergence after 1.2 Mio steps in 10-agent setting

- single-agent setting did nothing for some time in the beginning, did not have enough time to learn optimal strategy (low arrival rate) longer time for first curriculum
- higher entropy and learning rate did not seem to make good or bad change
- might try even larger learning rate for next setting

run: runs/flatland-rl__single_agent_priming_longer_start__1__1702653020, single_agent_shortest_path_longer_start.txt

- do 600'000 steps of single agent learning, with departure and arrival reward of 5, shortest path reward
    - then 1 million 50:50 shortest path and delay reward
    - then end in only delay, arrival and deadlock penalty
- lr 2.5e-2, max grad norm 0.7

Results

- agent learned an almost optimal policy in 600'000 steps, about 1h of training
- there was a Vorzeichenfehler, the deadlock_penalty was positive instead of negative

Run: runs/flatland-rl__single_agent_priming_adding_agents_sequentially__1__1702802977

- 1 agent, shortest path, arrival and departure reward
- 2 agents ,shortes path, arrival, departure and deadlock penalty
- 3 agents, arrival, shortest, departure and deadlock
- 10 agents, arrival, deadlock (still wrong sign on deadlock penalty)

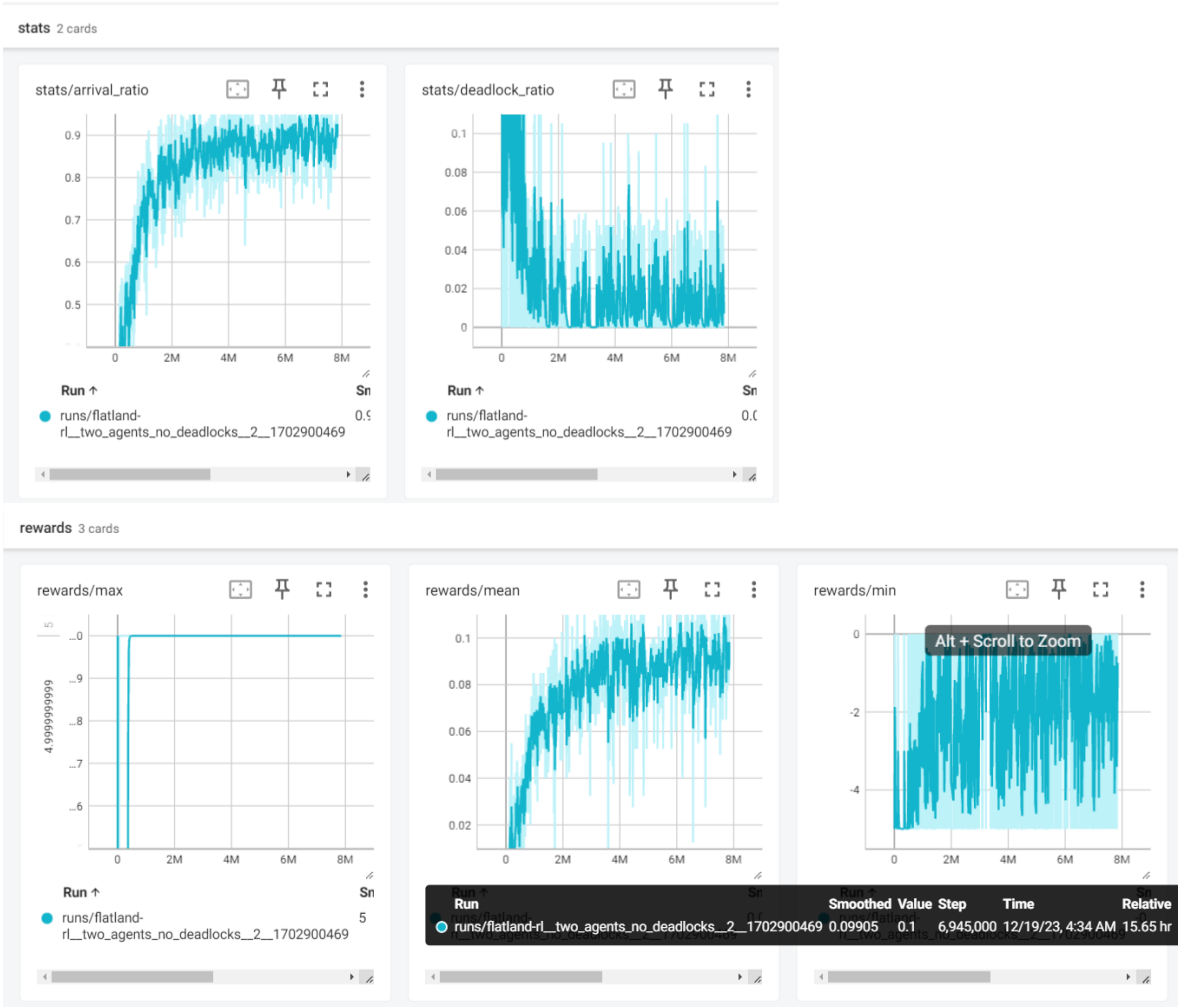
- interestingly. deadlock ratio is still falling, and arrival ratio still rising over time, even though rewards were long
- learning rates did not seem to have large impact

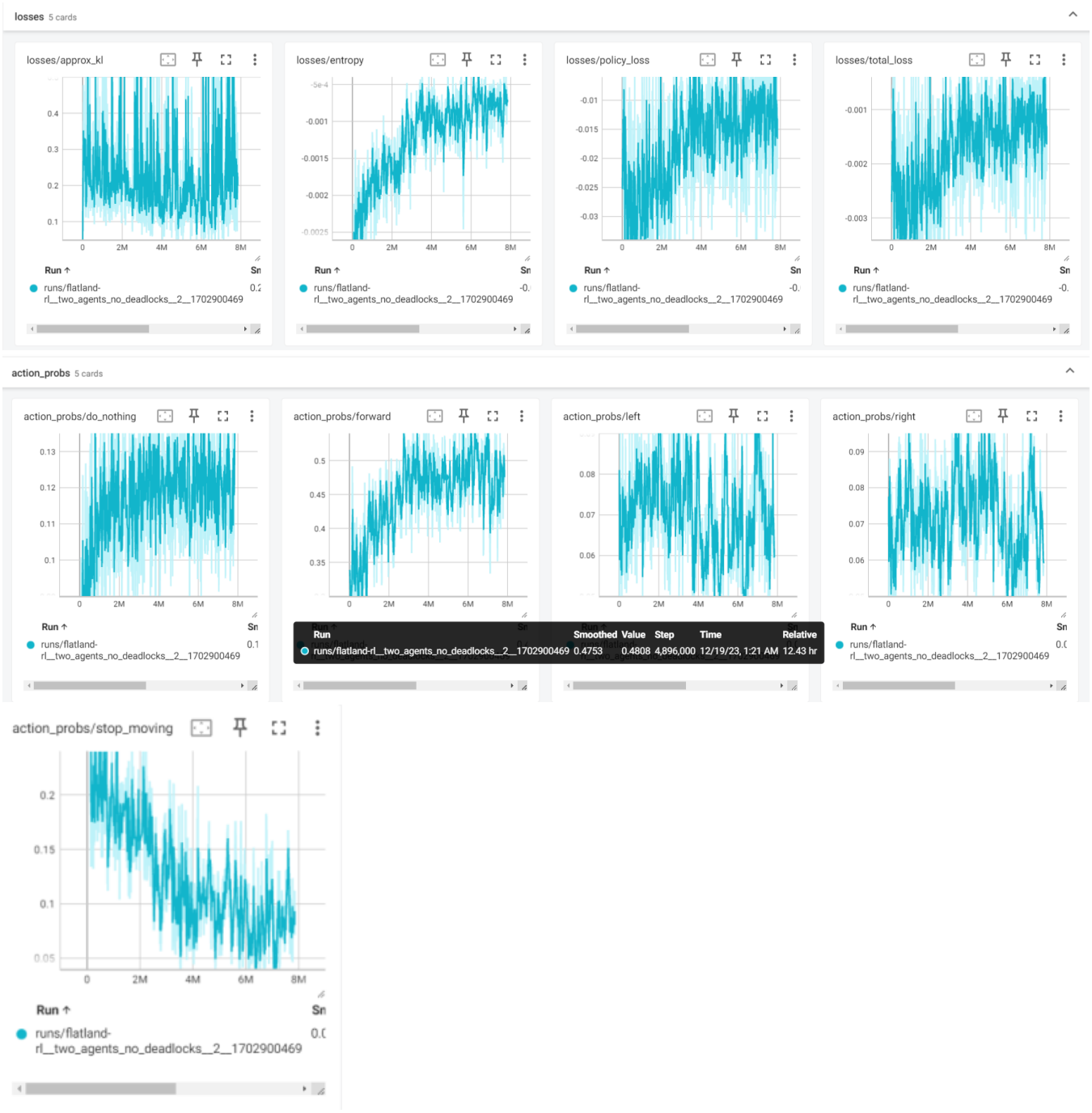
Run with fixed deadlock rewards

- Goal: try and show that we can reach 0 deadlocks with 2 agents
  - try sparser rewards, only arrival and deadlock rewards (we don't care about the shortest path too much for the moment, or possible delays at the end)

- doing 2 runs, see how it goes
- generally looking very promising so far

runs/flatland-rl__two_agents_no_deadlocks__2__1702900469





- Converges nicely in the beginning, somehow less pronounced jumps than in the setting where we give continuous, delay-based rewards
- but the arrival rate doesn't go to 1, nor does the deadlock ratio go to 0. While we don't know what the theoretical upper and lower bounds for these values are, the period around step 3mio for the deadlock ratio suggests that it can be 0 for longer periods. This raises the question if the learning rate is too large and we are oscillating around an optimal solution.

Hypothesis: The learning rate is too large to allow for a stable optimal solution

Test: Hot-start with current optimal model and a lower learning rate, see whether the model converges.

- accidentally have the new version where we give delay penalty at end of game for agents that have not arrived yet

changed arrival reward to only reward punctual arrivals

Two-Agent regularized setting

In order to try and gain a more stable training in the two-agent setting (and hopefully get arrival rate to 1 and deadlock rate to 0), we regularize the training in the following way

- do 500 rollout steps in each env, giving us a batch size of 4000
- max grad norm of 0.1
- learning rate of 2.5e-4
- clip coef of 0.1

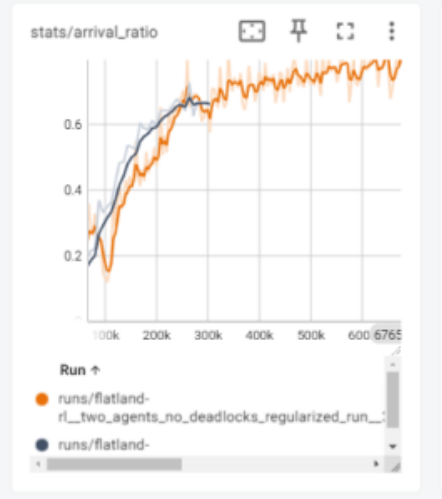
We hope this will give

- less jumps in KL value
- less zig-zagging of action probabilities

runs/flatland-rl__two_agents_no_deadlocks_regularized_run__3__1703162346

converges very nicely

comparison with run_commands/1_13_two_agents_no_deadlocks_regularized_batch_size_8000.txt, that has same setting but larger batch size



if anything learning is even a bit faster with larger batch size, and significantly smoother with smaller KL values we might even get to clip a bit less or increase the LR a bit?

batch size of 16_000 seems too big, training takes longer

- increase minibatch nr

runs/flatland-rl__tten_agents_no_deadlocks_regularized_run_bs_16000_clip_coef_0_2__1__1703191518 suggests that doing 16'000 steps with 10 agents become prohibitively expensive, probably problems with ram

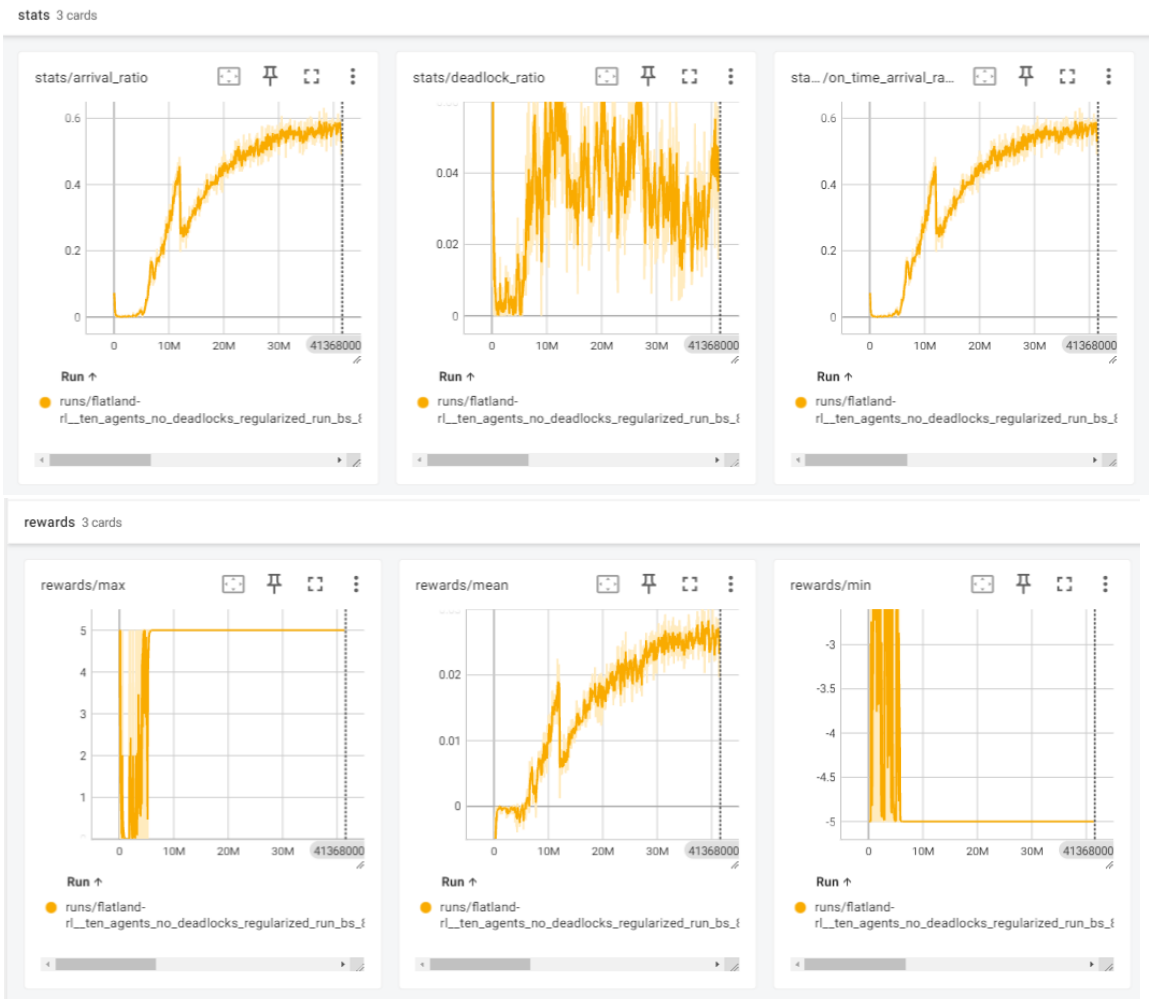
- do a run with two agents, 8'000 steps and a clip coef of 0.2, see if it becomes too unstable or if it's fine
  - indeed seems to have higher KL and lower clip rate values
  - doesn't seem to make a big difference in terms of learning speed to change the clip coef

- do a run and check if using a higher vf coef is good
  - so far a bit slower training, but good otherwise

- doing a hyperparameter sweep for 2 agent setting
  - using the discrete rewards enables quick convergence, and hence reasonably fast results about hyperparameter impact
  - made a mistake with the clip coef, it is not being considered in the sweep already fixed it, run again

Long run with 10 Agents

ten_agents_no_deadlocks_regularized_run_bs_8000_clip_coef_0_1_long_run



Conclusions:

- steps per seconds becomes very low for 10 agents for a batch size of 8'000
  - might have to balance batch size and agent number more agents maybe require less steps to be sufficiently regularized, as we have enough observations
  - the current learning speed was prohibitively low, took about 1 week to reach a plateau
  - the learning rate was set to be quite low (2.5e-6), maybe an increased learning rate would lead to faster convergence
- learning still happens nicely for 10 agents, even with the binary reward
  - but we notice that the arrival rate in the 10 agent case does not drop below 0.2 there are still quite a few positive examples for the model to go on
  - it is not clear if the model could have learned as well, had it started directly with 10 agents

Train 50 agents

- run_commands/1_20_fifty_agents_no_deadlocks_regularized_batch_size_2_000_clip_coef_0_2.txt try and see if arrival rate increases if we train with 50 agents without any hot-starting with smaller nr of agents (does the start with 2 agents actually help?)

- The model has very few examples of trains arriving, and hence prefers to not engage in any deadlocks it could be that this would resolve itself over time, but it is hard to say how long that would take, so the warm-up training with fewer agents seems to be sensible

Benchmarks in evaluation script

model	2 agents	5 agents
solution /policy/phase-III-50.pt	arrival ratio: 0.7767295837402344 deadlock ratio: 0.0	arrival ratio: 0.88064515590667 deadlock ratio: 0.0

model_checkpoints/flatland-rl__two_agents_no_deadlocks__2__1702900469/flatland-rl__two_agents_no_deadlocks__2__1702900469_3200000.tar	arrival ratio: 0.925000011920929 deadlock ratio: 0.02222222276031971	
model_checkpoints/flatland-rl__ten_agents_no_deadlocks_regularized_run_bs_8000_clip_coef_0_1_long_run__1__1703410006/flatland-rl__ten_agents_no_deadlocks_regularized_run_bs_8000_clip_coef_0_1_long_run__1__1703410006_41000000.tar	arrival ratio: 0.772455096244812 deadlock ratio: 0.0419161692261695 86	arrival ratio: 0.7680000066757 deadlock ratio: 0.086400002241

Comparison of MutliSyncDataCollector and ParallelEnv

As mentioned in the note in torchrl.collectors package — torchrl main documentation (pytorch.org), there are two strategies to parallelize rollouts: Make parallel environments that take each step synchronously and calculate the actions in a batch, or run separate data collectors.

Here we aim to compare the two options

- multisyncdatacollector:
  - still very clear cpu/gpu phases

Model Size Comparisons

default LSTM

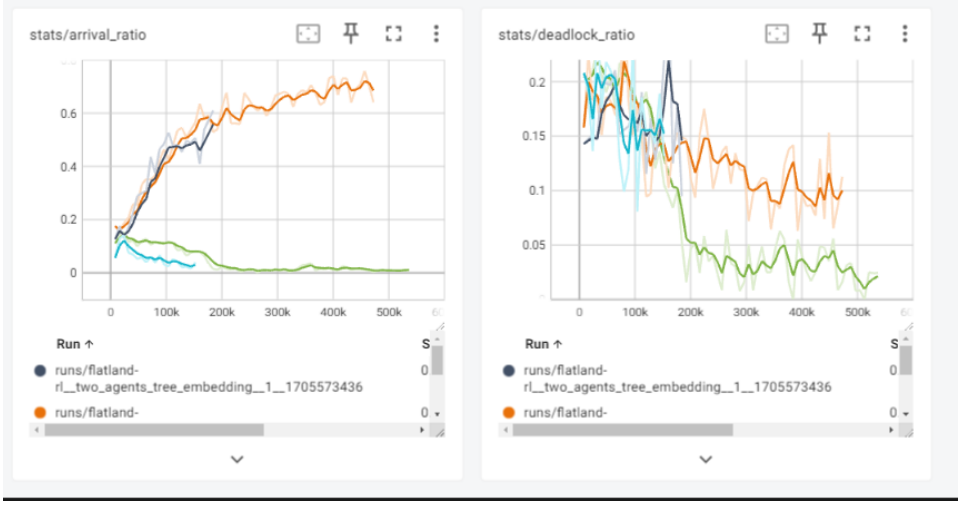
default lstm	**transformer tree embedding**

```
Number of Parameters:

attr embedding:     185,984
tree embedding:     219,776
transformer:      1,183,488
policy net:         164,869
value net:          164,353
total:            1,918,470
```

```
Number of Parameters:

attr embedding:     186,240
tree embedding:     709,120
transformer:      1,183,488
policy net:         164,869
value net:          164,353
total:            2,408,070
```

Experiment without LSTM

In runs two_agents_tree_embedding, it was investigated what happens if one freezes the weights of the tree lstm network to 0, effectively eliminating any information passing through it (run commands 1_33 and 1_34).



It can be seen that the model learns quickly when it has access to the LSTM information, and fails to do so when it does not. While this does not offer insight into what exactly it is that the model is learning about the features, it does show that the model needs the tree observations to learn

In this case, the model can't learn from the order of the data in the tensor directly, as the node attribute tensor is "sliced up" during the run of the

Value Loss Function

According to [2006.05990] What Matters In On-Policy Reinforcement Learning? A Large-Scale Empirical Study (arxiv.org), the loss function used for the value estimation can have an influence and l2 is optimal, and in these experiments so far the default used was the default of TorchRL, "smooth_l1". Hence these experiments investigate whether using l2 loss is advantageous.

Minimale Beispiele/Debugging

Insbesondere während der Entwicklung des Training-Codes hat sich die Verwendung von minimalen Beispielen als vorteilhaft erwiesen, da dadurch sehr schnell bestimmt werden konnte, ob der Algorithmus grundsätzlich lernen kann. Die Settings und Resultate werden hier kurz beschrieben. Die Resultate an sich sind dabei nicht interessant, jedoch können sie als Benchmark dienen, um einen neuen Ansatz schnell zu debuggen.

Line Map

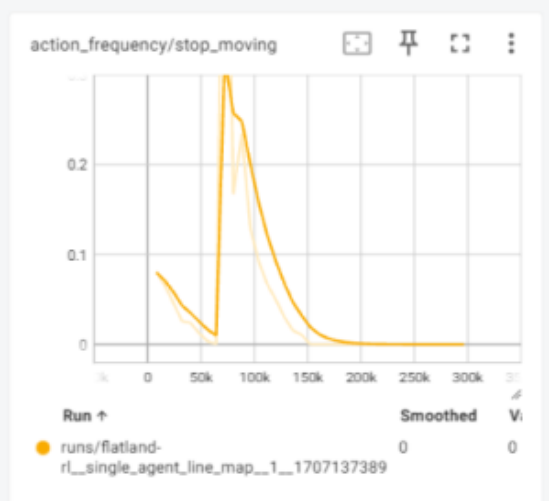
Beschreibung

Nur eine runde Strecke mit zwei Bahnhöfen. Es gibt einen Reward von 1, falls der Agent departed. Daher sollte der Agent möglichst schnell lernen, immer zu departen.

Run command

run_commands/minimal_debugging_examples/0_1_single_agent_line_map.txt

Resultate



bester Test: schauen ob Action Frequency von stop_moving auf 0 sinkt (hier gab es noch einen komischen Spike, schliesslich konvergiert es aber zu null)

Fazit/Take Away

- in diesem Setting sollte in wenigen tausend Schritten gelernt werden, dass Action "forward" besser ist (falls das nicht geschieht, gibt es höchstwahrscheinlich einen Bug im Code)
- Mögliche Fehlerquelle:
  - Interruption in der Backpropagation
    - sicherstellen, dass immer Tensoren und nie np Arrays verwendet werden
    - requires_grad=True auf den relevanten Tensoren

Figure of Eight

Beschreibung

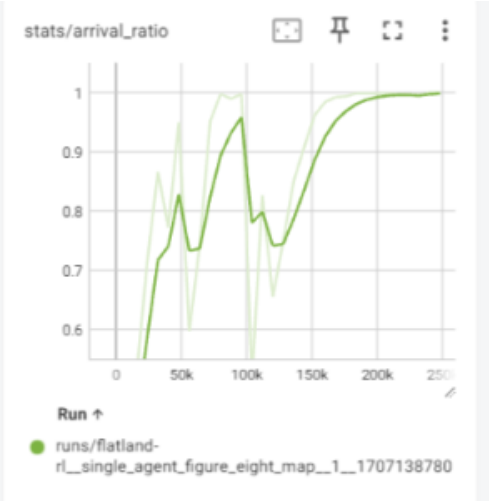
Eine Acht mit zwei Bahnhöfen (einer pro Schlaufe). Da Flatland die Startpositionen und Richtungen zufällig wählt, muss der Agent abwechslungsweise links oder rechts abbiegen, um ohne Extrarunde direkt zu seinem Ziel zu kommen. Somit kann getestet werden, ob Informationen aus den Observations erfolgreich berücksichtigt werden können

Run command

run_commands/minimal_debugging_examples/0_2_single_agent_figure_eight_map.txt

Resultate

runs/flatland-rl__single_agent_figure_eight_map__1__1707138780



Fazit

- Auch hier sollte das Modell in ca. 200'000 Schritten eine optimale Lösung finden, und falls es das nicht tut, ist ein Bug wahrscheinlich.
- Fehlerquellen in meinem Fall:
  - Kontrollieren, dass die valid Actions sinnvoll gehandhabt werden
  - Kontrollieren, dass in den Trainingsloops die logprobs der korrekten (gewählten) Handlung verwendet werden, und nicht die logprobs einer nicht-gewählten Handlung (die dann beispielsweise -inf sein könnte)

Single Agent Random Map

Beschreibung

Eine zufallsgenerierte 25x25 Flatland-Karte mit einem einzigen Agenten. Läuft schnell und lässt auch schnell sehen, ob das Modell grundsätzlich konvergiert.

Run command

run_commands/minimal_debugging_examples/0_3_single_agent_random_map.txt

Resultate

Fazit

- Auch hier sollte das Modell in ca. 200'000 Schritten eine optimale Lösung finden, und falls es das nicht tut, ist ein Bug wahrscheinlich.

Reward Struktur

Delay Reward

Beschreibung

Eine Serie von Experimenten, die jeweils für einen Agenten auf zufallsgenerierten 25x25 Karten die verschiedenen Rewards ausprobieren.

Run command

run_commands/reward_structure/1_1_single_agent_delay_reward.txt

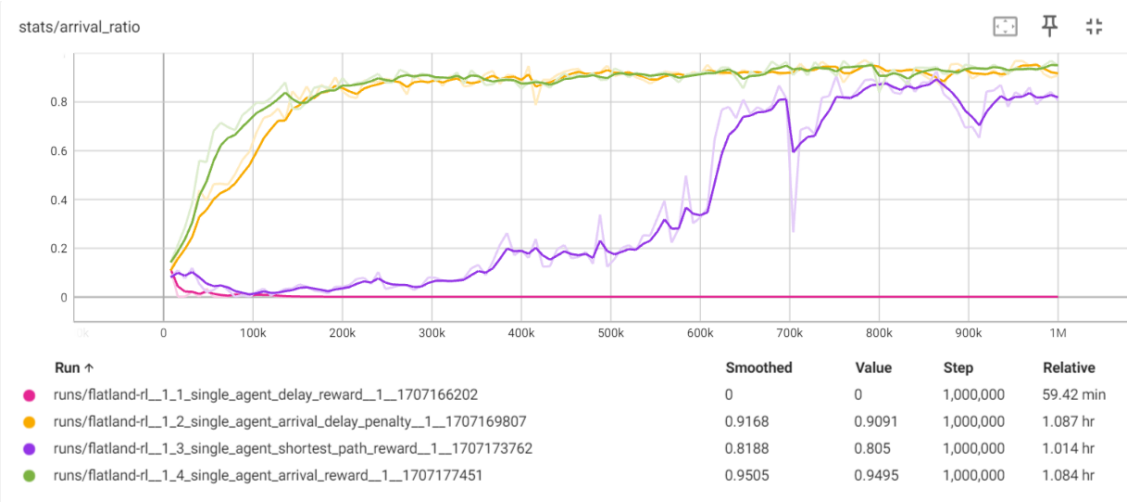
run_commands/reward_structure/1_2_single_agent_arrival_delay_penalty.txt

run_commands/reward_structure/1_3_single_agent_shortest_path_reward.txt

run_commands/reward_structure/1_4_single_agent_arrival_reward.txt

run_commands/reward_structure/1_5_two_agents_arrival_reward_deadlock_penalty.txt

Resultate



Run	Smoothed	Value	Step	Relative
runs/flatland-rl__1_1_single_agent_delay_reward__1__1707166202	0	0	1,000,000	59.42 min
runs/flatland-rl__1_2_single_agent_arrival_delay_penalty__1__1707169807	0.9168	0.9091	1,000,000	1.087 hr
runs/flatland-rl__1_3_single_agent_shortest_path_reward__1__1707173762	0.8188	0.805	1,000,000	1.014 hr
runs/flatland-rl__1_4_single_agent_arrival_reward__1__1707177451	0.9505	0.9495	1,000,000	1.084 hr

Fazit/Take Away

- Rewards nur am bei Ankunft auszugeben ist besser als nach jedem Schritt
- Kein klarer Unterschied zwischen arrival_delay_penalty und arrival_reward
  - Habe in folgenden Experimenten jedoch arrival_reward verwendet, da dieser prominenter verwendet wird im Paper und einfacher ist
  - Dies deckt sich mit folgender Erkenntnis aus Sutton & Barto Book: Reinforcement Learning: An Introduction (incompleteideas.net) (Seite 76 im pdf): "The reward signal is your way of communicating to the robot what you want it to achieve, not how you want it achieved."
  - Mit Rewards, die unter anderem die Distanz auf dem kürzesten Pfad berücksichtigen, zwingen wir den Agenten bereits auf den kürzesten Pfad. Dabei ist dieser in unserem Setting vielleicht gar nicht optimal.

Investigating Various Hyperparameters

3 Batch Size

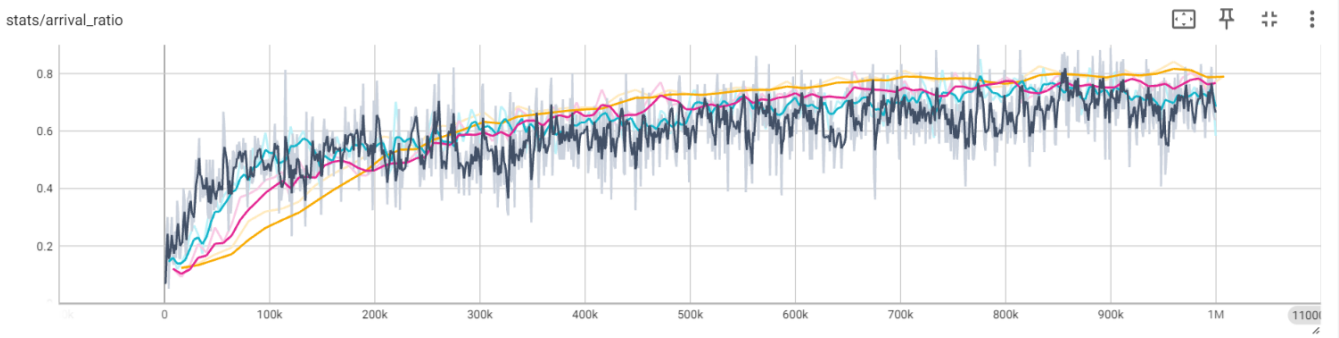
Description

Investigate if different batch sizes have an impact on training

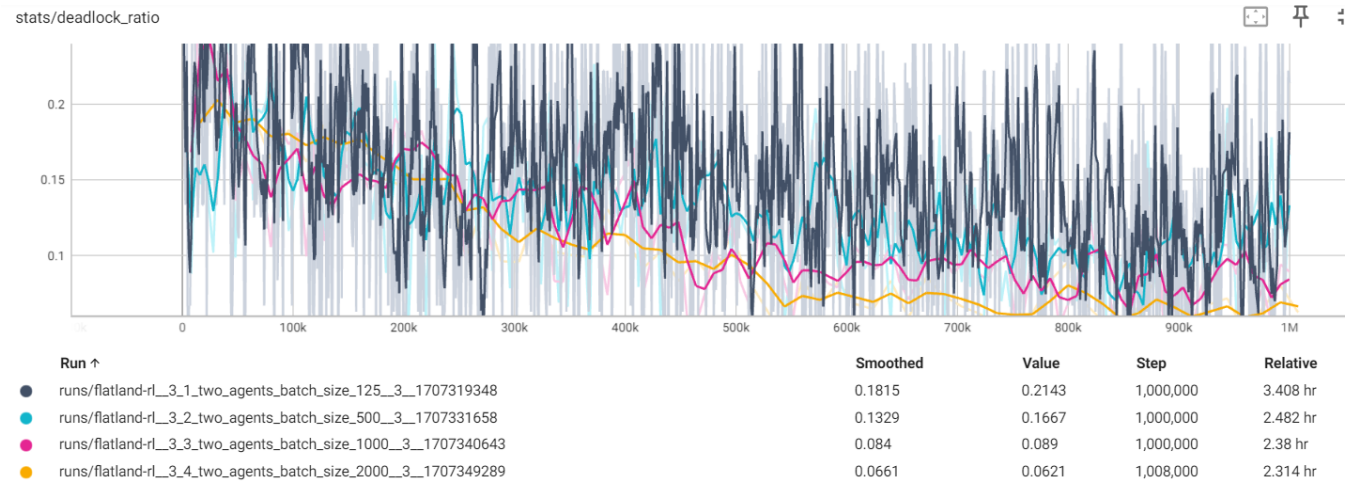
Run Commands

run_commands/3_batch_size

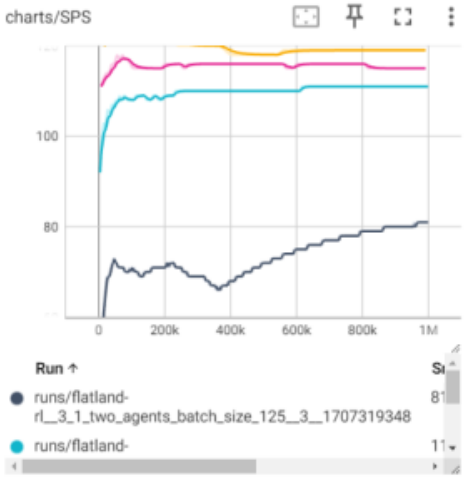
Results



Run	Smoothed	Value	Step	Relative
runs/flatland-rl__3_1_two_agents_batch_size_125__3__1707319348	0.6637	0.6429	1,000,000	3.408 hr
runs/flatland-rl__3_2_two_agents_batch_size_500__3__1707331658	0.6846	0.5833	1,000,000	2.482 hr
runs/flatland-rl__3_3_two_agents_batch_size_1000__3__1707340643	0.767	0.774	1,000,000	2.38 hr
runs/flatland-rl__3_4_two_agents_batch_size_2000__3__1707349289	0.7892	0.7925	1,008,000	2.314 hr



Run	Smoothed	Value	Step	Relative
runs/flatland-rl__3_1_two_agents_batch_size_125__3__1707319348	0.1815	0.2143	1,000,000	3.408 hr
runs/flatland-rl__3_2_two_agents_batch_size_500__3__1707331658	0.1329	0.1667	1,000,000	2.482 hr
runs/flatland-rl__3_3_two_agents_batch_size_1000__3__1707340643	0.084	0.089	1,000,000	2.38 hr
runs/flatland-rl__3_4_two_agents_batch_size_2000__3__1707349289	0.0661	0.0621	1,008,000	2.314 hr



Conclusion

Larger batch sizes take a bit longer to start learning in the beginning, but pay off quickly (as an added bonus, the plots become much cleaner). In terms of the tunning speed, large batch sizes also perform better than smaller ones (provided there is enough storage).

It seems recommendable to use the largest batch size that the infrastructure allows. Keep in mind that this will depend on the number of agents used, as more agents mean larger observations.

4 Clip Coef

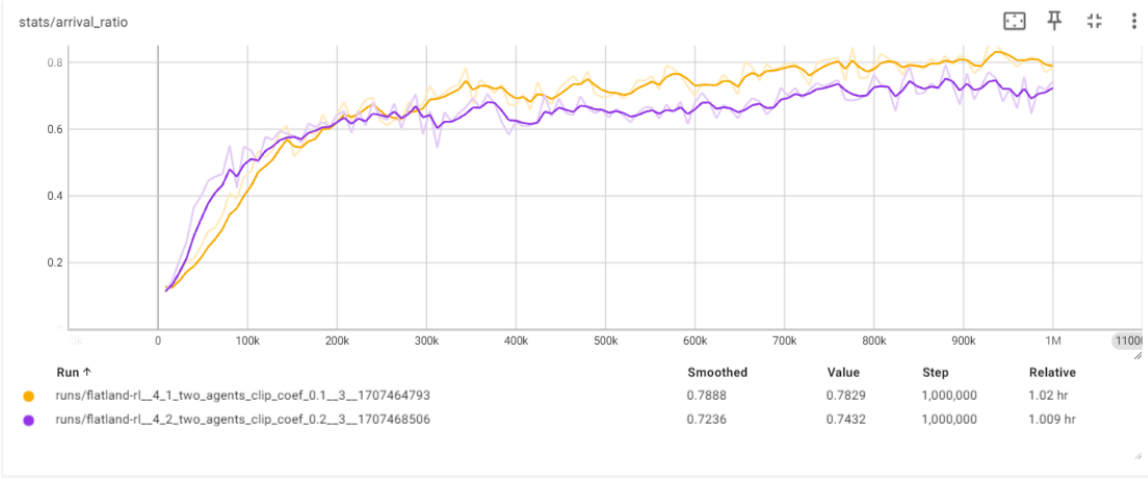
Description

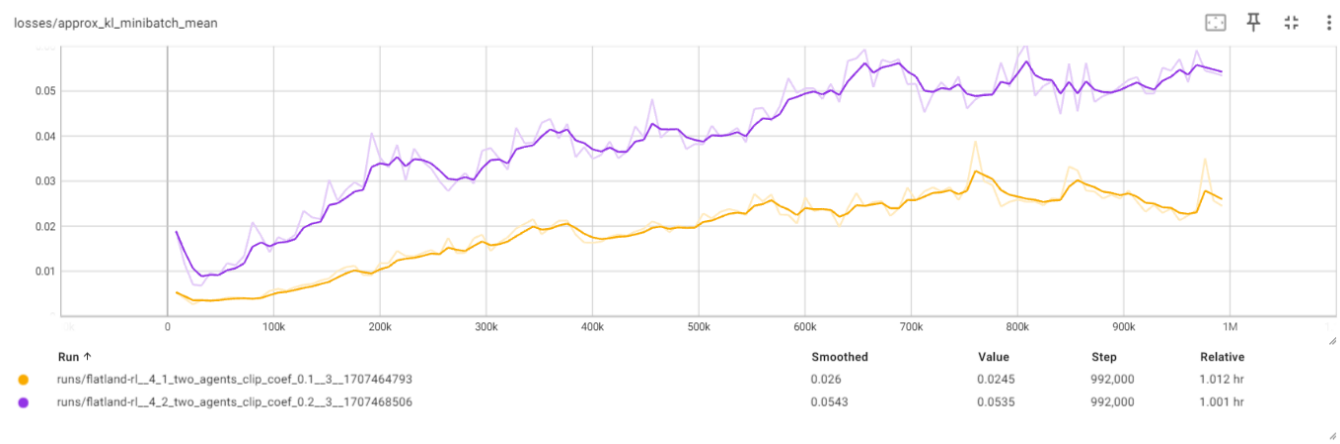
Comparison of two different values of the PPO clip loss coefficient (0.1 and 0.2)

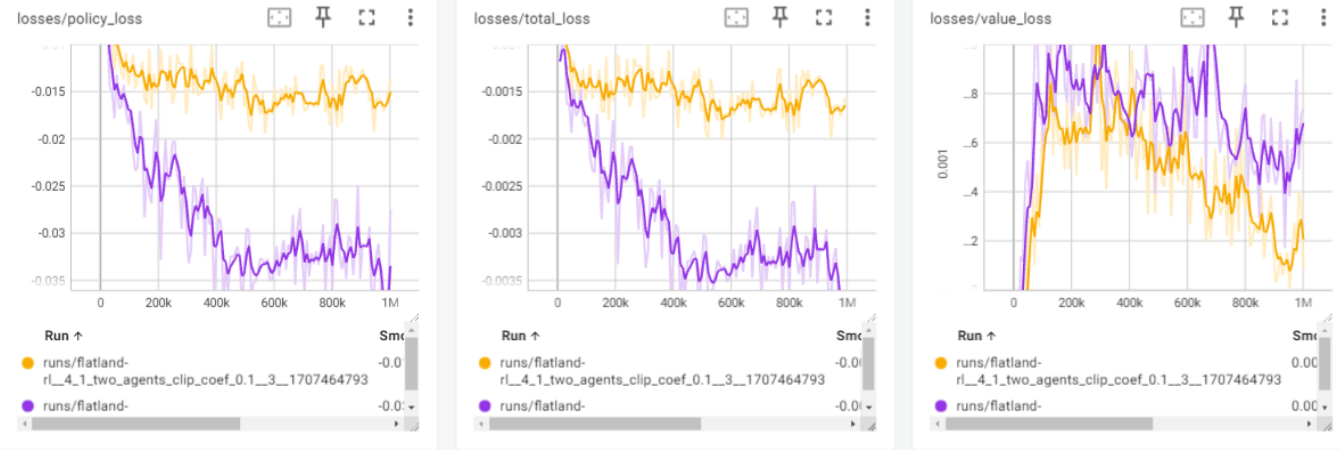
Run Commands

run_commands/4_clip_coef

Results







Conclusion

While the KL divergence is larger for the higher clip coefficient, and the policy loss changes more rapidly, there is no clear evidence that this translate to better performance in terms of the arrival ratio. So based on the available data, there is no clear evidence for either value to be prefered. It would be an interesting path of further investigation to see how different clip coef values compare in the long run for a higher number of agents, as at least in this small setting, the higher clip coef increases more rapidly, but then plateaus at a lower level.

5 Learning Rate

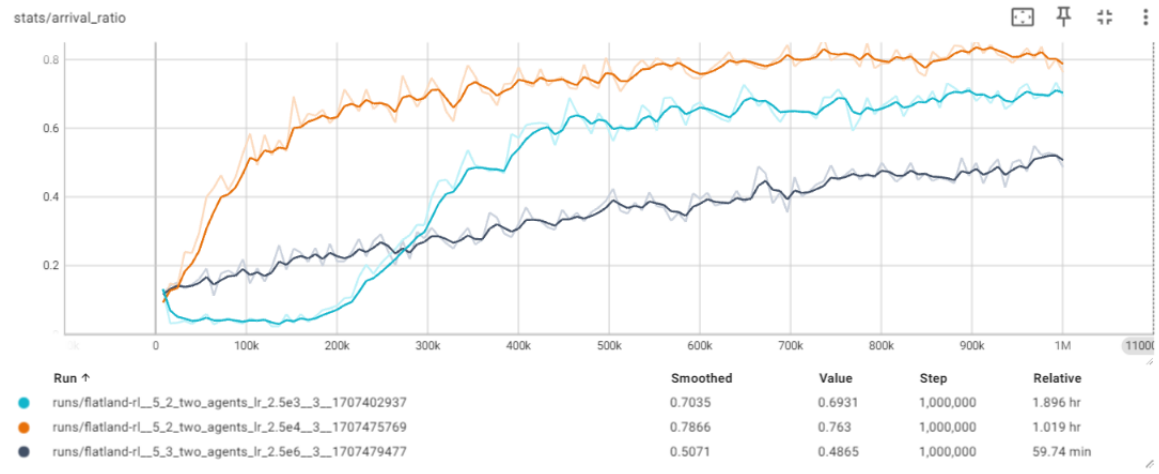
Description

Comparison of three different learning rates (2.5e-3, 2.5e-4, 2.5e-6).

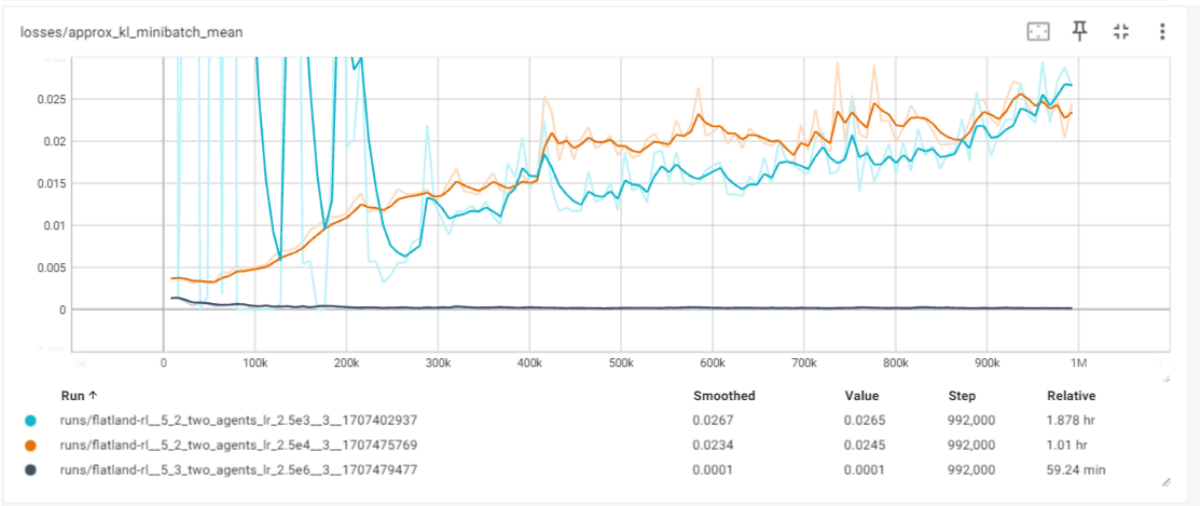
Run Commands

run_commands/5_learning_rate

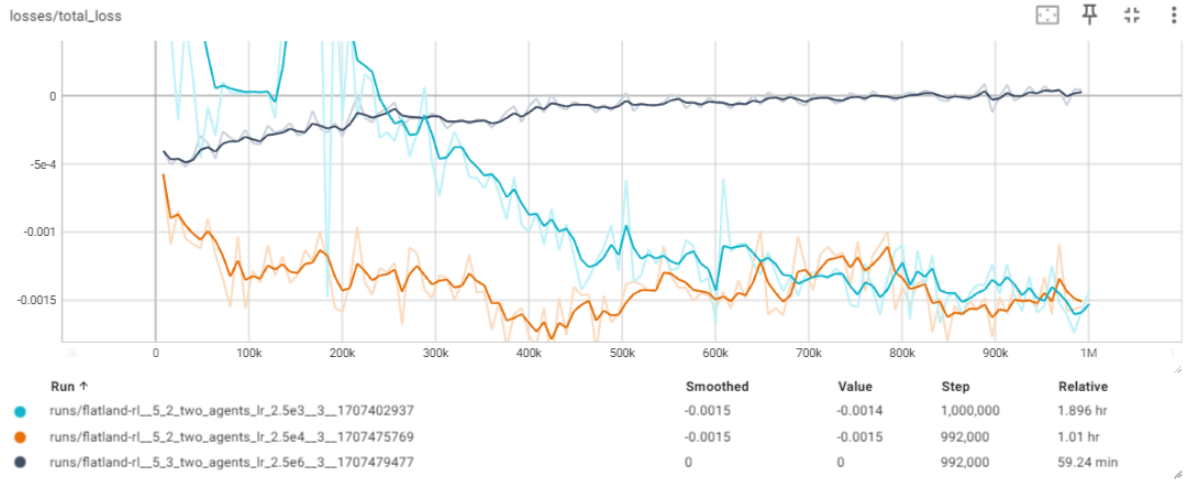
Results



Run	Smoothed	Value	Step	Relative
runs/flatland-rl__5_2_two_agents_lr_2.5e3__3__1707402937	0.7035	0.6931	1,000,000	1.896 hr
runs/flatland-rl__5_2_two_agents_lr_2.5e4__3__1707475769	0.7866	0.763	1,000,000	1.019 hr
runs/flatland-rl__5_3_two_agents_lr_2.5e6__3__1707479477	0.5071	0.4865	1,000,000	59.74 min



Run	Smoothed	Value	Step	Relative
runs/flatland-rl__5_2_two_agents_lr_2.5e3__3__1707402937	0.0267	0.0265	992,000	1.878 hr
runs/flatland-rl__5_2_two_agents_lr_2.5e4__3__1707475769	0.0234	0.0245	992,000	1.01 hr
runs/flatland-rl__5_3_two_agents_lr_2.5e6__3__1707479477	0.0001	0.0001	992,000	59.24 min



Run	Smoothed	Value	Step	Relative
runs/flatland-rl__5_2_two_agents_lr_2.5e3__3__1707402937	-0.0015	-0.0014	1,000,000	1.896 hr
runs/flatland-rl__5_2_two_agents_lr_2.5e4__3__1707475769	-0.0015	-0.0015	992,000	1.01 hr
runs/flatland-rl__5_3_two_agents_lr_2.5e6__3__1707479477	0	0	992,000	59.24 min

Conclusion

A learning rate of 2.4e-4 seems to be a very reasonable choice. The higher rate of 2.5e-3 at least in this case lead to a very high KL divergence in the first 200k steps. Once these initial large changes are over, the KL is in a similar range to the one of the 2.5e-4 learnign rate, and the arrival ratio also increases rapidly. A KL divergence in the order of magnitude of 0.01-0.015 at least seems reasonable in this post [D] KL Divergence and Approximate KL divergence limits in PPO? : r/reinforcementlearning (reddit.com), so I chose to go with the learning rate of 2.5e-4. What further supports this is that the lower learning rate of 2.5e-6 just converges very slowly.

6 Max Grad Norm

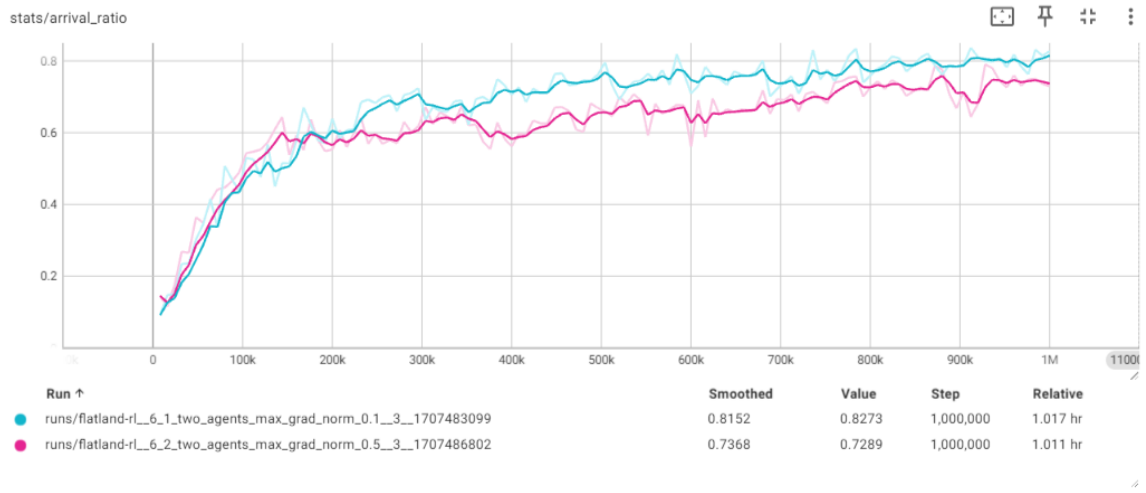
Description

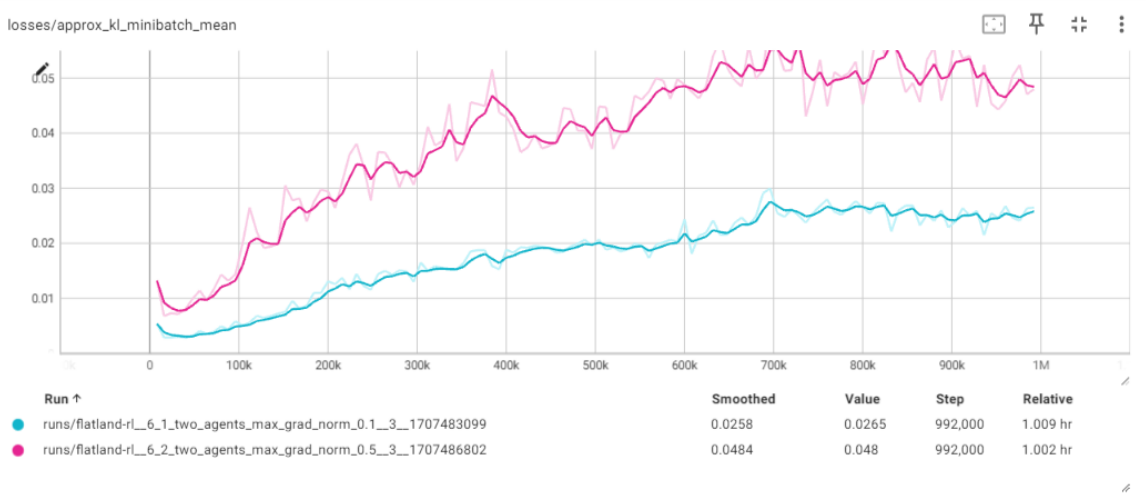
Comparison of two different max grad norms (0.1 and 0.5)

Run Commands

run_commands/6_max_grad_norm

Results





Conclusion

There is no clear difference in the arrival ratio performance. The lower max grad norm value of 0.1 has a slightly lower KL divergence, but I do not know if the KL of around 0.05 is actually a problem in any way.

7 Value Function Coefficient

Description

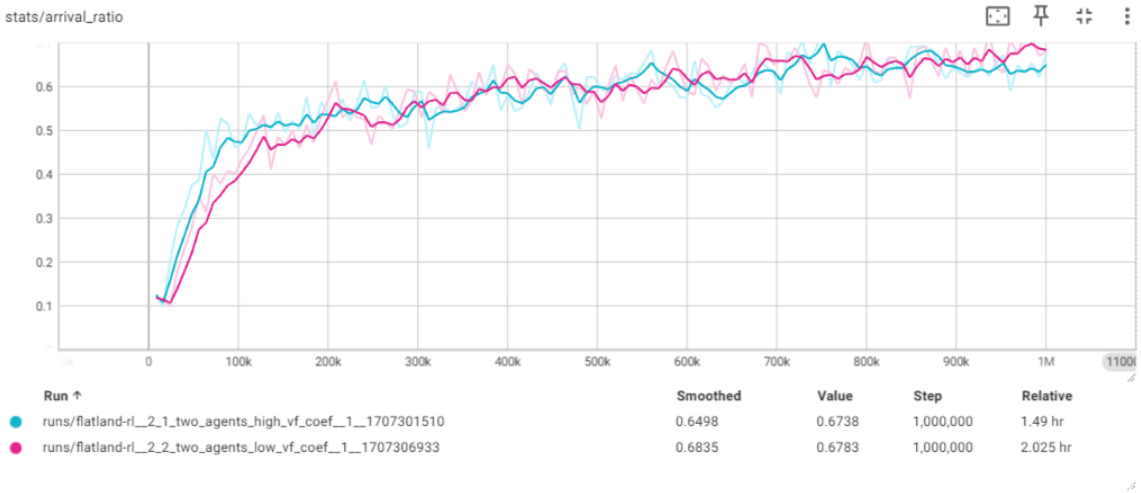
Two different values for the value function coefficient used in the loss calculation were considered (0.01 and 0.00001).

Run Commands

run_commands/7_vf_coef/7_1_two_agents_high_vf_coef.txt

run_commands/7_vf_coef/7_2_two_agents_low_vf_coef.txt

Results



runs/flatland-rl__2_1_two_agents_high_vf_coef__1__1707301510

runs/flatland-rl__2_2_two_agents_low_vf_coef__1__1707306933

Conclusion

The PPO algorithm appears relatively indifferent even to large changes in the value function coefficient, hence it is advisable to remove this parameter from hyperparameter tuning for efficiency reasons.

8 Curriculum Learning for Ten Agents

Description

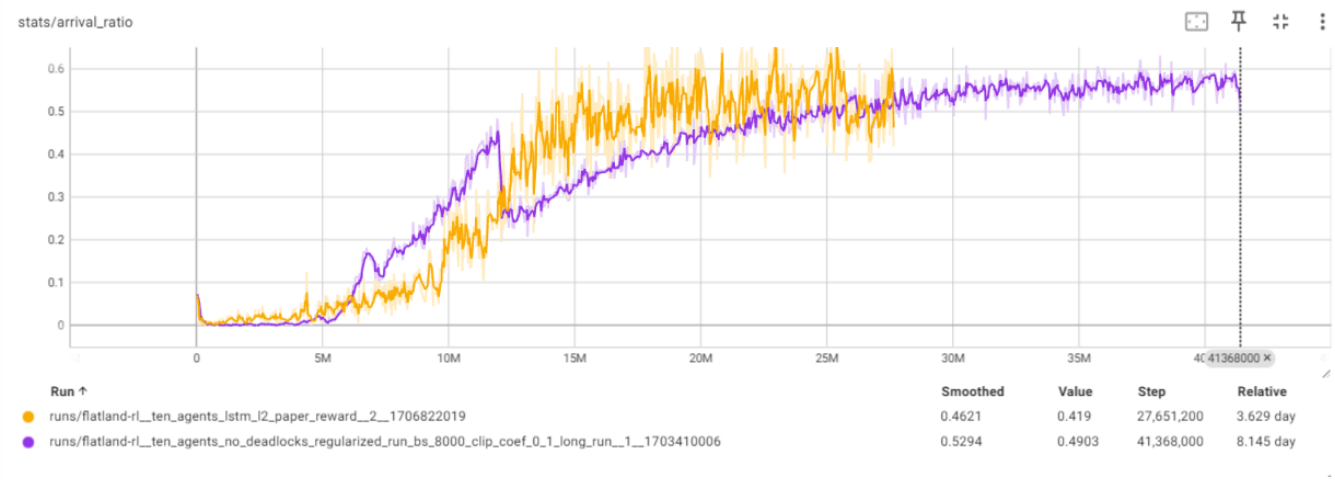
In order to train a game with ten agents, two approaches were used: In one case, ten agents were used from the beginning. In the other case, the model was trained using an increasing number of agents (_, _, _ and then _). It should be noted that these experiments also differ in a number of other parameters (the curriculum option has a lower lwearning rate and uses a different value loss), limiting their value. They could not be repeated due to the long running times.

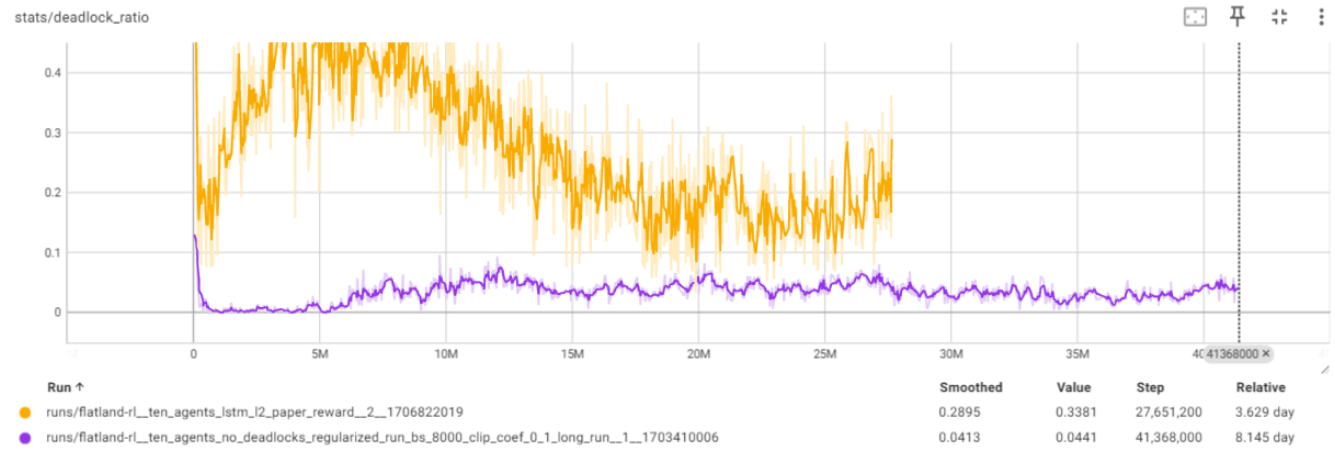
Run Commands

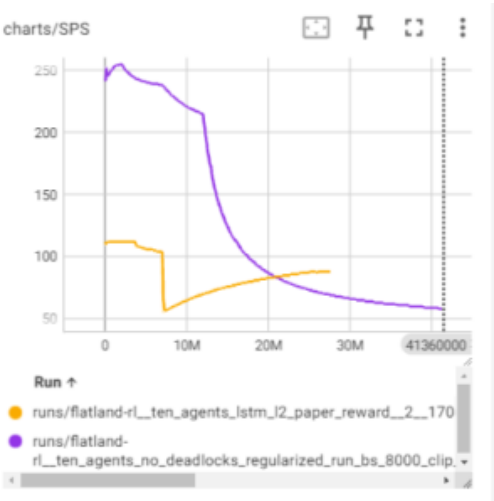
run_commands/8_curriculum_learning/8_1_ten_agents_curriculum.txt

run_commands/8_curriculum_learning/8_2_ten_agents_no_curriculum.txt

Results







Conclusion

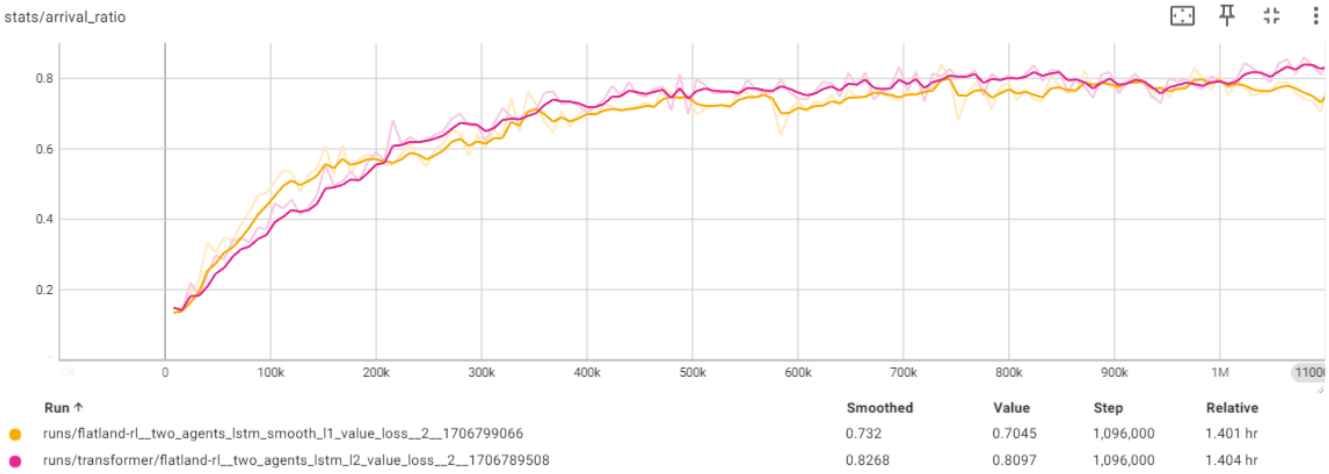
In terms of the arrival rate, both options perform comparably. It is interesting to note that the curriculum learning setting lead to a lower deadlock rate. The difference with the biggest practical impact of this experiment is the running time, where we see that once the ten agents were reached for the curriculum learning setting, its number of steps per seconds (SPS) decreased to just below 60, while the non-curriculum learning one was at around 110 (the large decrease in the non-curriculum learning setting was due to a performance dip in the infrastructure). This is a result of the different choice of number of steps, where the non-curriculum learning setting only conducted 200 steps per rollout, instead of 1000. While the variance in the arrival rate is much larger as a consequence of this, it also seems to be a better choice for the given infrastructure.

9 Value Loss Function

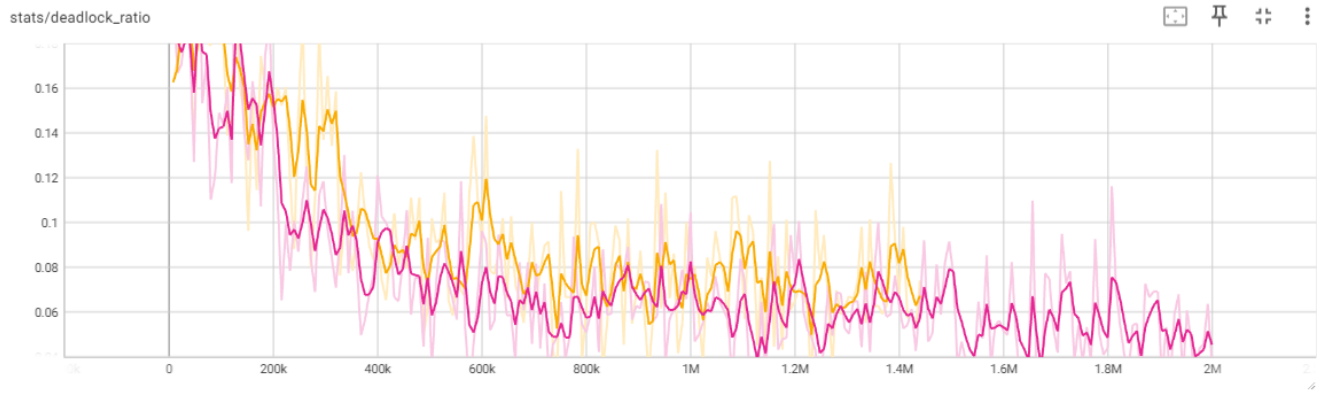
Description

Two different loss functions for the calculation of the value loss were investigated, the default chosen by TorchRL (SmoothL1Loss — PyTorch 2.2 documentation) and l2 loss. This was inspired by [2006.05990] What Matters In On-Policy Reinforcement Learning? A Large-Scale Empirical Study (arxiv.org)

Results



Run	Smoothed	Value	Step	Relative
runs/flatland-rl__two_agents_lstm_smooth_l1_value_loss__2__1706799066	0.732	0.7045	1,096,000	1.401 hr
runs/transformer/flatland-rl__two_agents_lstm_l2_value_loss__2__1706789508	0.8268	0.8097	1,096,000	1.404 hr



Run	Smoothed	Value	Step	Relative
runs/flatland-rl__two_agents_lstm_smooth_l1_value_loss__2__1706799066	0.0673	0.0741	1,440,000	6.235 hr
runs/transformer/flatland-rl__two_agents_lstm_l2_value_loss__2__1706789508	0.0455	0.0366	2,000,000	2.624 hr

Conclusion

The difference is small, but the l2 loss function seems to perform slightly better.

Overall Conclusion:

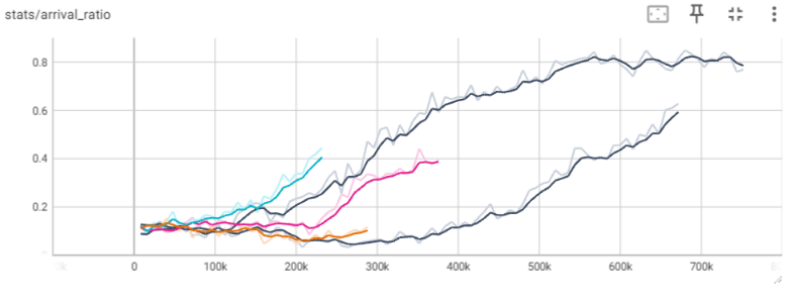
Based on the results above and many failed trials I did, my recommendation for a training setting would be as given below. These settings are in no way claimed to be optimal, and should rather be seen as a reasonable starting point to get some results, upon which one can improve further!

Parameter	Value
Learning Rate	2.5e-4
Clip Coef	0.2
Max Grad Norm	0.1
Value function coefficent	0.01
value loss function	L2
batch size	as big as your machine will allow 😉

⚠️ Big Caveat: Impact of Random Seed ⚠️

Description

Just as a word of caution for the above results, there can be a substantial variance in convergence speed depending on the random seed used. This is partially mitigated above by using the same random seeds, but **must** be kept in mind when comparing different runs. It would be an interesting avenue of further investigation to run the above settings with multiple random seeds and confirm the conclusions in a more general setting.



Run	Smoothed	Value	Step	Relative
runs/flatland-rl__two_agents_no_deadlocks_regularized_run_bs_8000__1__1704807243	0.1029	0.1165	288,000	20.16 min
runs/flatland-rl__two_agents_no_deadlocks_regularized_run_bs_8000__2__1704808525	0.5934	0.6286	672,000	49.09 min
runs/flatland-rl__two_agents_no_deadlocks_regularized_run_bs_8000__3__1704723654	0.7854	0.7692	752,000	55.47 min
runs/flatland-rl__two_agents_no_deadlocks_regularized_run_bs_8000__4__1704813671	0.4058	0.4453	232,000	16.34 min
runs/flatland-rl__two_agents_no_deadlocks_regularized_run_bs_8000__5__1704814702	0.3872	0.3949	376,000	27.13 min

Other Observations (without accompanying experiments)

Just some more things I observed while trying a lot of things (without claim to generality)

- small initializations are valuable, as they make the initial action distribution more uniform and the model avoids getting stuck in the beginning

Hyperparameter Optimization

There is a hyperparameter optimization script in the repo based on cleanrl/tuner_example.py at master · vwxyzjn/cleanrl (github.com), which uses the https://optuna.org/ framework.

Due to the very high duration needed for even short runs (1'000'000 steps per run), no useful results could be obtained. It would be a very interesting path of further research.

Technische Überlegungen/Implementationsdetails

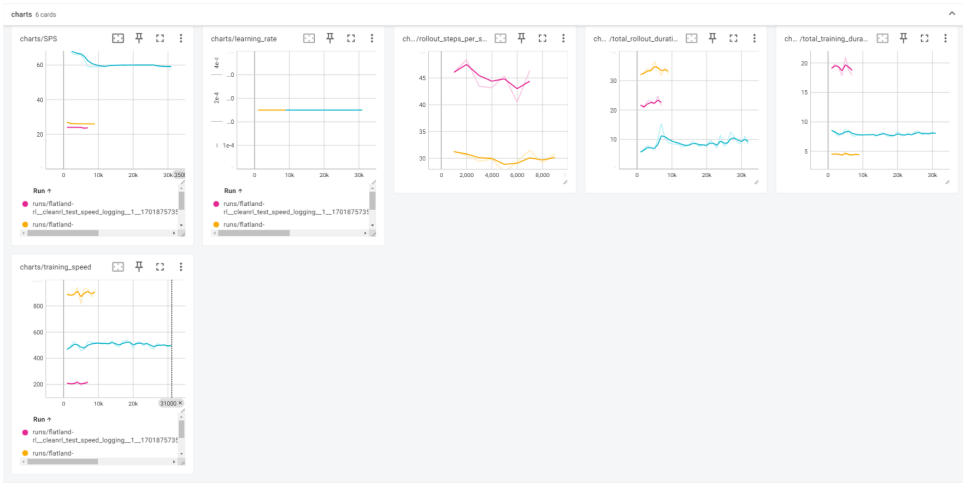
Experimente über die Trainingsgeschwindigkeit (Wall Clock Time)

Alle Experimente wurden auf einem Lenovo Thinkpad P1 G2 durchgeführt.

CleanRL vs TorchRL

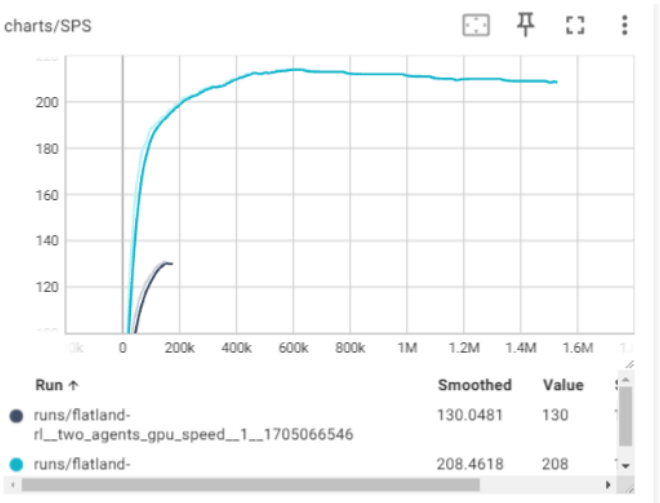
This experiment can't be reproduced with the repo, as the CleanRL code does not work anymore (or would need some teaking 😉) The general point here is: If you can parallelize your rollouts in parallel environment and thereby use all CPU cores you have available, do it! Below we can see that the blue line (highest line in the SPS graph) is almost twice as fast in the overall training than the CleanRL version working without parallelization.

It probably doesn't have to be TorchRL, but any framework of that kind will be very useful.

- 

Training auf GPU vs CPU

Since flatland can only run on CPU, the rollouts will always occur on CPU. The model training however can be done on a GPU, and the very quick experiment below at least suggests that a GPU is indeed highly beneficial and should be used whenever possible.

- 

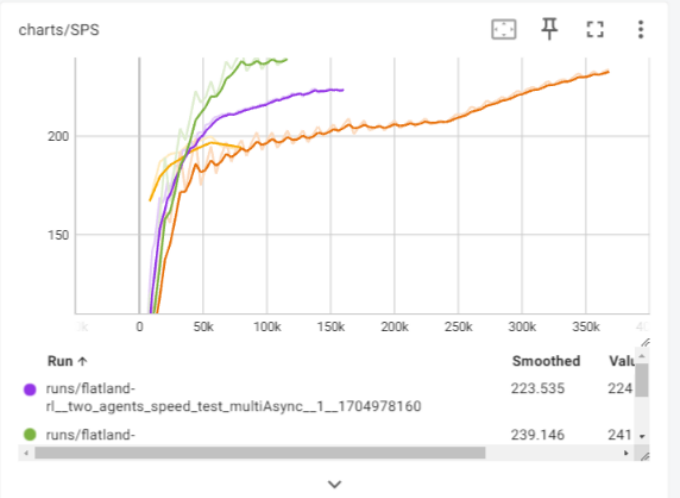
run_commands/12_speed_comparisons/12_1_two_agents_gpu_speed_test_gpu.txt

run_commands/12_speed_comparisons/12_2_two_agents_gpu_speed_test_no_gpu.txt

Parallelisierung von Rollouts und Training

TorchRL allows for further parallelization of the rollouts (on CPU) and the training (on GPU), avoiding the CPU being idle while the GPU is working and vice versa. While this means that there might be a small lag between the model used in the rollouts and the newest trained model, some very short tests did not suggest any massive problems with convergence. Furthermore, the speed increased by about a factor of _

Due to time constraints, this strategy was not investigated thoroughly, and it would be an interesting path of further research. The optimal number of parallel rollouts and batch sizes might also have to be fine-tuned for the specific infrastructure used.



Mögliche RL Frameworks

Name	Link	Eindruck
TorchRL	Introduction to TorchRL — torchrl main documentation (pytorch.org)	▪ sehr neu, durchdacht ▪ von den Leuten von PyTorch ▪ braucht neuen Type Tensordict, welcher sehr praktisch ist ▪ viele vorgefertigte Components, mit denen man mix and match machen kann ▪ Falls Environment noch nicht verfügbar, kann es mühsam sein, das anzupassen
CleanRL	vwxyzjn/cleanrl: High-quality single file implementation of Deep Reinforcement Learning algorithms with research-friendly features (PPO, DQN, C51, DDPG, TD3, SAC, PPG) (github.com)	▪ single-File implementations ▪ super um verschiedene RL Algorithmen zu verstehen ▪ sehr flexibel, man kann viel selber rumhacken
RlLib	RLlib: Industry-Grade Reinforcement Learning — Ray 2.9.1	▪ älter als TorchRL aber sehr verbreitet ▪ hab nicht damit gearbeitet, kann Vor /Nachteile zu TorchRL nicht beurteilen

Transformer Tree Architektur

Experimente

- Positional Encoding ist notwendig
- transformer block ist auch notwendig
- nur gemeinsam kann die architektur das volle potential entfalten

Comparison with LSTM Architecture

Description

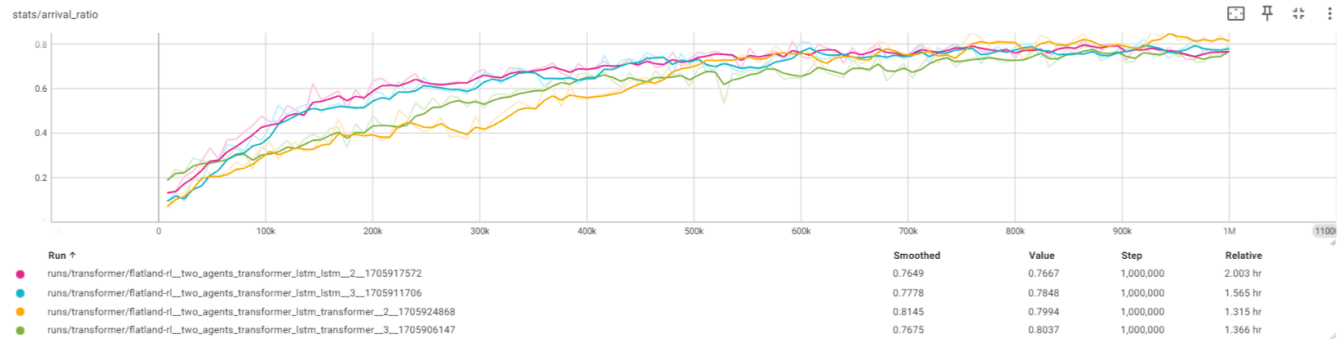
In order to get a general understanding if the transformer embeddings are useful, they were compared to the LSTM architecture from the paper.

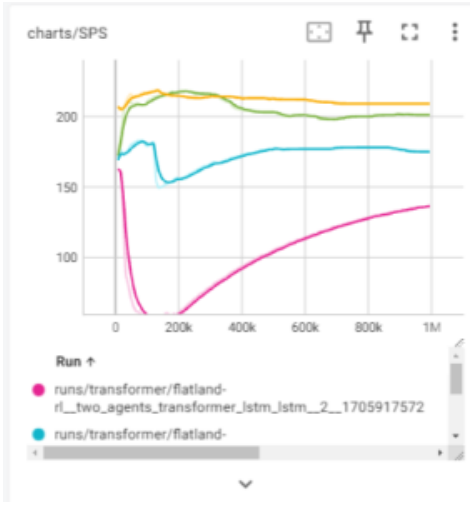
Run Commands

9_1_two_agents_transformer_lstm_lstm.txt

9_2_two_agents_transformer_lstm_transformer.txt

Results





Conclusion

The transformer and LSTM architectures perform comparably. The random seed seems to make the larger difference. The transformer seems to run a bit faster (higher SPS). There is no evidence to prefer one architecture over the other.

Run Names

Run Names
runs/transformer/flatland-rl__two_agents_transformer_lstm_lstm__2__1705917572
runs/transformer/flatland-rl__two_agents_transformer_lstm_lstm__2__1705917572
runs/transformer/flatland-rl__two_agents_transformer_lstm_transformer__2__1705924868
runs/transformer/flatland-rl__two_agents_transformer_lstm_transformer__3__1705906147

Different Choices of Transformer Architecture

Several different choices for the transformer architecture were considered.

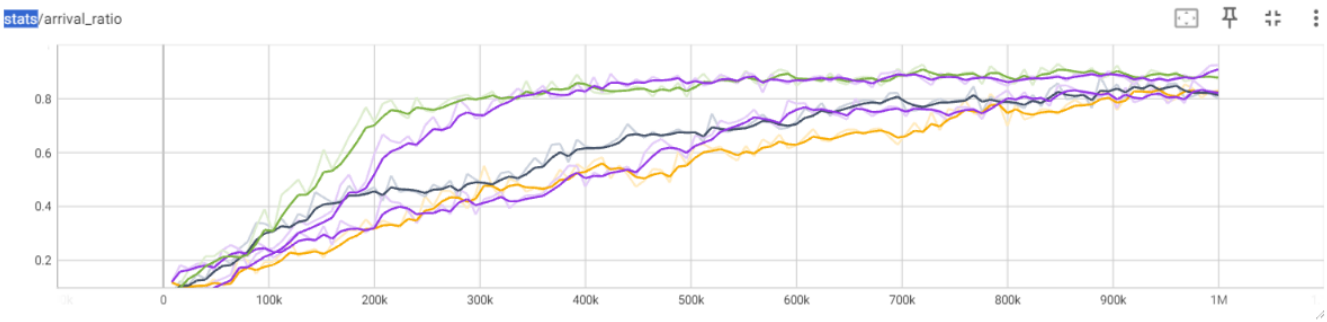
Note: As the architecture of the transformer was not parametrized, these experiments require changing the hard-coded values in the code. Therefore the run commands are generally the same, and reproduction requires the changing of the values in the code aswell.

Description of cases

The following changes were considered:

- Initialization Values
- increasing the number of heads used in multi-head attention
- increasing the number of passes through the transformer
- increasing the size of the embedding dimension

Results



Run	Smoothed	Value	Step	Relative
runs/transformer/flatland-rl__two_agents_transformer_256_hidden_layer.txt__2__1706192372	0.826	0.8313	1,000,000	1.524 hr
runs/transformer/flatland-rl__two_agents_transformer_256_hidden_layer_8_heads.txt__2__1706198162	0.8216	0.8137	1,000,000	1.439 hr
runs/transformer/flatland-rl__two_agents_transformer_2_heads__2__1706168040	0.8127	0.8	1,000,000	1.186 hr
runs/transformer/flatland-rl__two_agents_transformer_2_layers__2__1706101505	0.9093	0.9249	1,000,000	1.637 hr
runs/transformer/flatland-rl__two_agents_transformer_4_layers__2__1706113702	0.8784	0.8735	1,000,000	1.687 hr

Conclusion

- Increasing the number of layers seems to be a good idea (the Attention is all you need paper used 6 here)
- Increasing the size of the hidden layer from 128 to 256, or increasing the number of attention heads is not beneficial, at least in the short run for two agents. It would be interesting to see if the added complexity is beneficial for more complex settings.

Relevant Runs

runs/transformer/flatland-rl__two_agents_transformer_256_hidden_layer.txt__2__1706192372
runs/transformer/flatland-rl__two_agents_transformer_256_hidden_layer_8_heads.txt__2__1706198162
runs/transformer/flatland-rl__two_agents_transformer_256_hidden_layer_8_heads.txt__2__1706198162
runs/transformer/flatland-rl__two_agents_transformer_256_hidden_layer_8_heads.txt__2__1706198162
runs/transformer/flatland-rl__two_agents_transformer_4_layers__2__1706113702

Value of Positional Encoding

Description

To test if the positional encoding/transformer architecture was necessary (and not just fancy overengineering), elements of the architecture were separately removed, namely:

- the transformer block (i.e. just have the positional encoding and node attributes passed through linear layers)
- the positional encoding (i.e. just have the transformer work on the node attributes)
- no embedding of the node attributes at all

Additionally, the order of sum of the node attributes and positional encodings were permuted (such that the root node was not always in the 0-th row of that tensor), to investigate whether the model learned the relative order of the nodes, or the actual positional encoding.

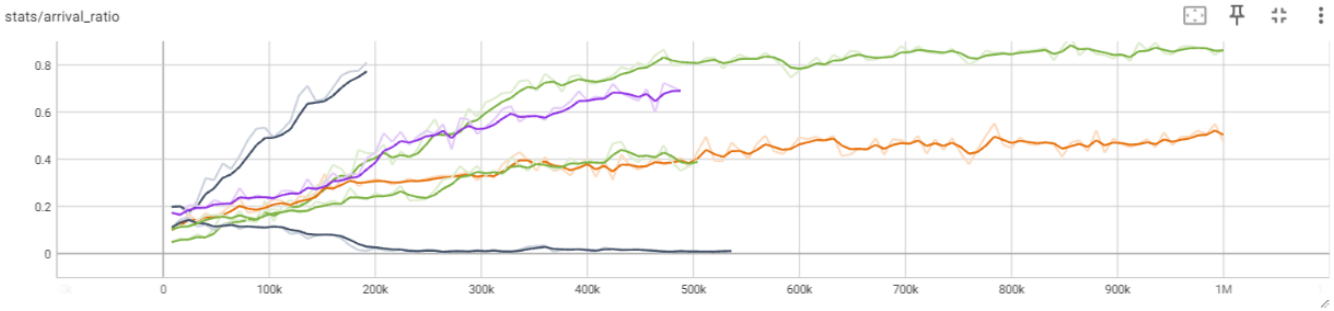
runs/transformer/flatland-rl__two_agents_transformer_net_tree_permutation__3__1705590076

runs/transformer/flatland-rl__two_agents_transformer_net_tree_permutation_no_pos_emb__3__1705593972

	with permutation	**without permutation**
no transformer (just linear layers layers)	runs/transformer/flatland-rl__two_agents_transformer_just_mlp_perm__2__1706176326	runs/transformer/flatland-rl__two_agents_transformer_just_mlp__2__1706172601
no positional encoding	runs/transformer/flatland-rl__two_agents_transformer_net_tree_permutation_no_pos_emb__3__1705593972	runs/transformer/flatland-rl__two_agents_transformer_net_small_initialization_no_pos_enc__1__1705498808 runs/transformer/flatland-rl__two_agents_transformer_net_small_initialization_no_pos_enc__1__1705498808
full model	runs/transformer/flatland-rl__two_agents_transformer_net_tree_permutation__3__1705590076	
tree embedding set to zero		runs/transformer/flatland-rl__two_agents_tree_embedding_zeroed__2__1705564768

Run Commands

Results



Run	Smoothed	Value	Step	Relative
runs/transformer/flatland-rl__two_agents_transformer_just_mlp__2__1706172601	0.8618	0.865	1,000,000	1.011 hr
runs/transformer/flatland-rl__two_agents_transformer_just_mlp_perm__2__1706176326	0.5036	0.4757	1,000,000	1.077 hr
runs/transformer/flatland-rl__two_agents_transformer_net_small_initialization_no_pos_enc__1__1705498808	0.7728	0.8095	192,000	17.47 min
runs/transformer/flatland-rl__two_agents_transformer_net_tree_permutation__3__1705590076	0.69	0.691	488,000	1.031 hr
runs/transformer/flatland-rl__two_agents_transformer_net_tree_permutation_no_pos_emb__3__1705593972	0.3885	0.3992	504,000	2.098 hr
runs/transformer/flatland-rl__two_agents_tree_embedding_zeroed__2__1705564768	0.0108	0.0122	536,000	47.88 min

The plot is tedious and interpretation is limited, due to the differing seeds used ⚠️.

Comparing the purple run with the lower gree one (permuting the tree once without and once with positional encoding), it is clear that the purple version with positional encoding performs better (these also use the same seed).

However, similar learning was observed for the case of having no permutation and no positional encoding (i.e. just the transformer block) or having just the linear layers and positional encoding (no transformer block).

To summarize, if we want the best observed performance, we can remove quite a bit from the transformer modell (either the transformer or the encoding), but both these options suffer if we start permuting the node order, indicating that the model is learning the tree structure from the node order. It would be interesting to investigate if this strategy still works for a higher number of agents, as the correspondence between node order and tree structure breaks down more and more the deeper we are in the tree.

If we force the model to learn the tree structure via the positional encodings by permuting the nodes, then only the transformer with positional encoding performs well.

As a sanity check, not having any tree observations at all performs badly.

Investigating the Use of Transformers for Tree Embeddings

Context

In the current version of the Flatland paper, the authors use a TreeLSTM structure obtain a fixed-length embedding of the observation tree, which consists of node attributes and the tree structure. Given the dominance of transformers over LSTM architectures in NLP tasks, it was investigated whether transformer architecture could also be applied to obtain an embedding of the observation tree. The information on the last node in the tree (which is presumably closest to the final destination) does not have to propagate through several recurrent layers in the transformer, which is a reason that makes this approach interesting.

Adapting the Transformer for tree structures

In the classical transformer as introduced in [1706.03762] Attention Is All You Need (arxiv.org), the input is a sequence of word vectors. Because transformers do not feature recurrency, the problem of informing the model about the relative position of the words in a sentence was solved by adding positional encodings to the word vectors. Then, the transformer could process all word vectors at once, while still having information about their positions in a sentence.

The positional encodings used in the original paper are only applicable to sequences, hence another approach was needed. We adapted the approach described in Novel positional encodings to enable tree-based transformers - Microsoft Research in the following manner:

- given the sequential nature of the decisions in Flatland (the agent first has to decide which way to go at the switch closes to it, i.e. the node with depth 1 in the tree, etc.), the encodings build up by depth. That is, the first three digits will refer to the decision needed to get to the node at the node of depth 1, the next three digits for the node of depth 2 etc.
- That way, the positional encoding could also be smaller than three times the maximal tree depth, since one could easily cut off the information about the deepest decisions (assuming that these will be less relevant to the current decision)

Transformer architecture

The parameter size of the transformer increases rapidly, so we initially settled on a size that has roughly the same number of parameters as the LSTM model.

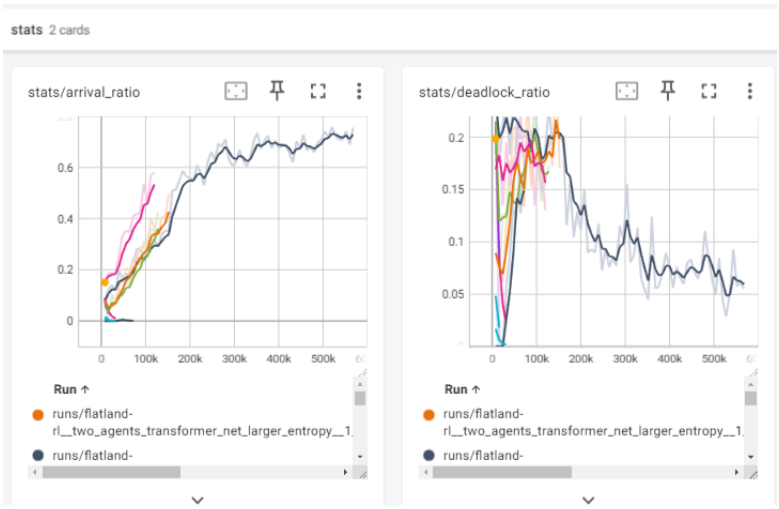
Training Settings

So far, the following peculiarities have been observed:

- the transformer seems to be very sensitive to deadlock penatlies, even deadlock penalties of 2.5 make fall into a local optimum and stop departing at all
- Scal

Layer normalization (two_agents_transformer_net_larger_entropy)

- normalizing both agent attr embedding and tree transformer output
- entropy coefficient of 0.1
- only arrival reward of 5

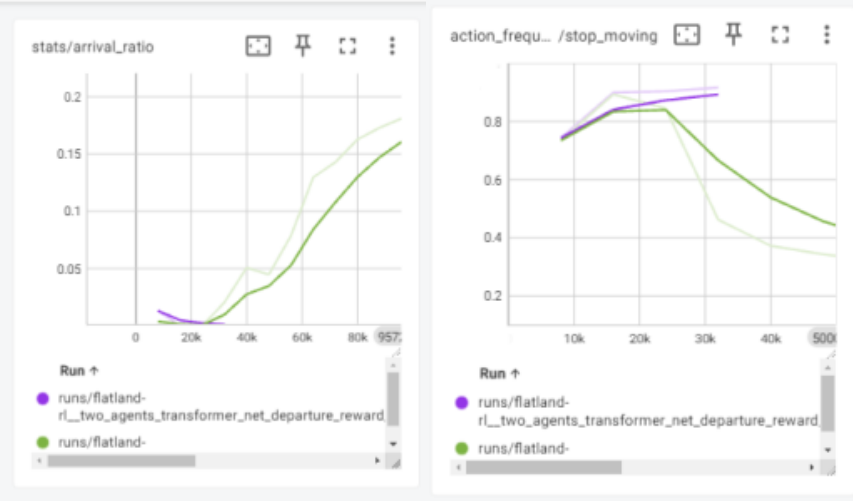


- we see that the model can indeed return good results
- however, the training depends very much on the seeding
    - e.g. the pink and the light blue curve were both conducted for the same parameters, only differing in the seed

- if the random initialization leads to unbalanced action probabilities (strongly favoring stopping), the model gets stuck

Departure Rewards

- setting the departure reward to 1 (keeping all other aspect equal to above scenario), the model can still get stuck
- however, setting the entropy coefficient to 1 makes it possible to get unstuck, pushing the probabilities of all actions closer to each other
    - with the high entropy coefficient, it seems impossible to get to higher arrival ratios however



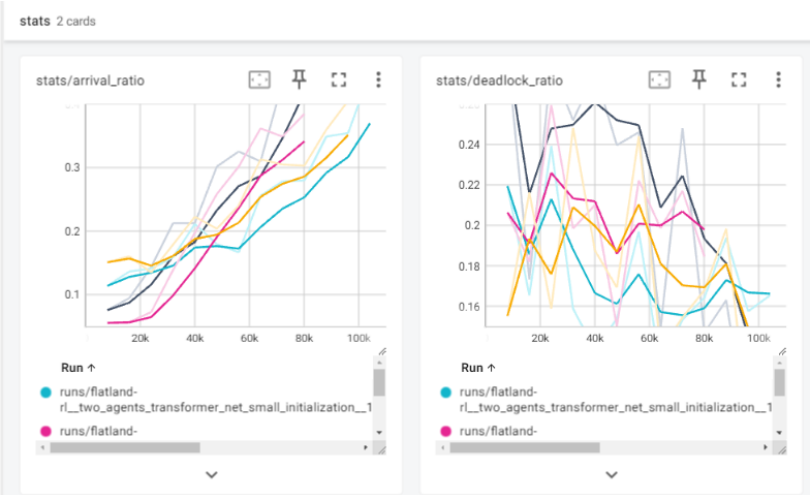
purple: entropy coefficient of 0.1

green: entropy coefficient of 1

- we see that initially the action frequency in both cases is very similar, highly favoring stopping. after a couple of rounds of optimization however, the model pushes towards more entropy, lowering the frequency of stop moving

- the question now is: how can we initialize the model such that we immediately start in a place where all actions are equally probable?
    - the advantage of this inititalization would be twofold
        - less time wasted in beginning to get to good point
        - high entropy coefficient does not seem to allow for good perfomance, it just prevents collapse. would need to change entropy coefficient after "warm-up" anyway.

Experiment: initializing entire network with sd 0.01



initializing all layers (except tree transformer) with a normal dist with sd 0.01 yields much more stable training, as the actions are much more equally likely

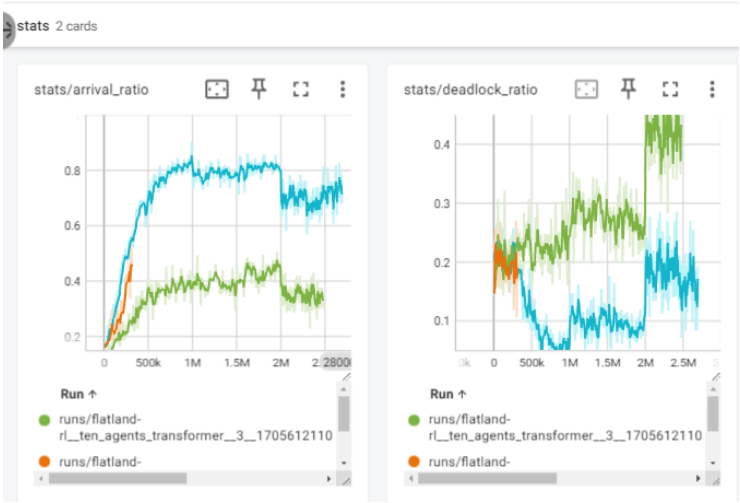
Test if positional encodings are relevant: disable them, with same setting as above, see what happens

- transformer still seems to be able to learn well, reaches arrival ratio of over 0.6
- check if maybe dropping the batch norm, and just doing small initializations is going working generally
    - maybe the embeddings need to be more "fixed"
    - try randomly permuting the order of the node attributes, that way we can determine if the model is learning the order of the nodes that's when positional encoding should really make a difference

Experiment: only initializing last layer of action network with sd 0.01

Interaction between sd and deadlock penalty: is training still so unstable if we set deadlock penalty to 2.5?

runs/flatland-rl__ten_agents_transformer



green line: Transformer with permuted tree node order without positional encoding

blue lien: transformer with permuted tree node order with positional encoding

the positional encodings clearly allow the model to learn more

Verständnis des Papers

Unterseite	Was
TreeLSTM in Flatland MARL	Mathematische Beschreibung von wie sie LSTM implementiert haben im Code (als N-ary Tree LSTM und **nicht** wie im Paper beschrieben als Child-Sum TreeLSTM
Empirical Analysis of Tree Embeddings	Versuch zu verstehen, was die Tree Embeddings so an Informationen weitergeben
Investigate C-Utils Tree Generation in Flatland	Beschreibung der Observations

Empirical Analysis of Tree Embeddings

In an effort to understand if the tree embeddings provided by the pretrained network provide enough information for the action layer to learn left/right decisions in a single agent setting, the embeddigns and available actions where recorded for a rollout of 10'000 steps using the original 50 agent network from the paper without any modifications.

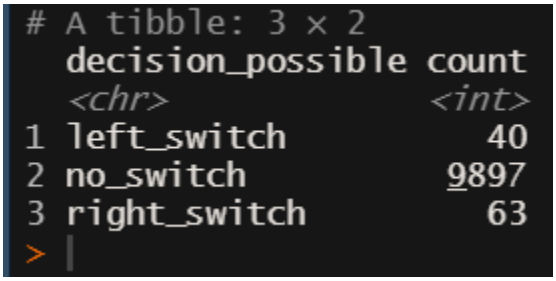
Settings for Data Generation

- Use pretrained network
- No training during the rollout
- single agent
- figure of eight track layout

Results

Overview of Rollout Data

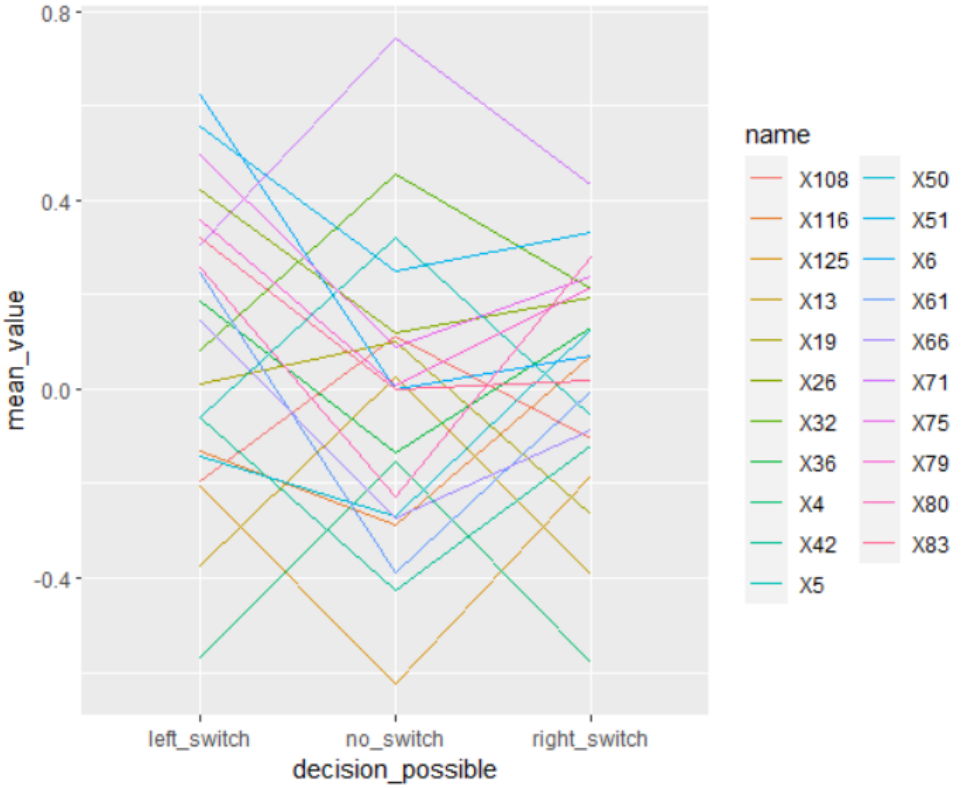
The rollout data shows how generally there wer much fewer situations where the train could switch left or right, which was to be from the track layout. We still get at least 40 observations, which should allow for ok inference.



```
# A tibble: 3 × 2
  decision_possible count
  <chr>              <int>
1 left_switch           40
2 no_switch           9897
3 right_switch          63
> 
```

Values of embedding variables grouped by possible decisions

In the following plot, we are only considering variables for which the difference between the max and max value of means per possible decision was greater than 0.3 for clarity. These are the variables with the strongest signals. Interestingly, there are quite a few variables that have a V or reversed V shape, meaning that they take on similar values for any kind of switch, and similar values if there is no decision possible. However, there is less signal that differentiates between left and right turns. tThere are some variables however that do not differ greatly between no_switch and one direction, and have a larger difference for the other direction. Apparently this is enough for the model to learn decisions.



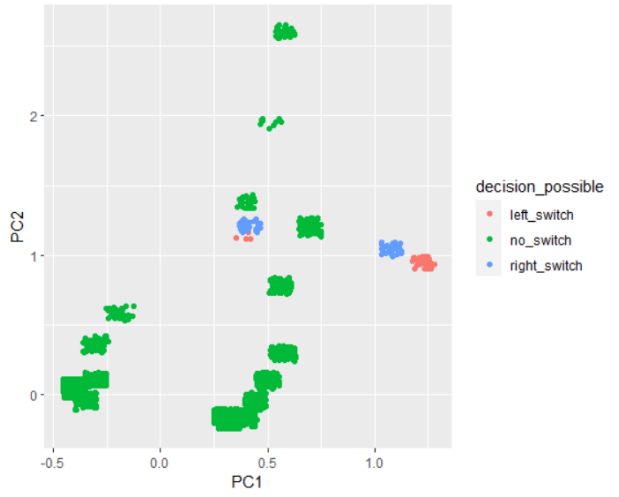
PCA of Embeddings

First two prinicpal components already explain quite a bit of variance 🙂.

```
> summary(pca)
Importance of components:
                          PC1    PC2     PC3     PC4     PC5     PC6     PC7     PC8     PC9    PC10    PC11
Standard deviation     0.3870 0.2944 0.16121 0.14361 0.10659 0.07318 0.05616 0.05140 0.04729 0.04124 0.03812
Proportion of Variance 0.4783 0.2767 0.08298 0.06585 0.03628 0.01710 0.01007 0.00843 0.00714 0.00543 0.00464
Cumulative Proportion  0.4783 0.7550 0.83800 0.90385 0.94012 0.95722 0.96729 0.97573 0.98287 0.98830 0.99294
```

Added jitter to the plot, as we only have a limited number of positions on our map, so this corresponds to a limited number of values the embedding returns.

We see how even just using the first two principal components, the cases do get projected to different spots, but maybe less pronouncedly different than what one could expect.



Conclusion

There are differences between the turning left and right cases which in theory should be enough for the model to learn. For more robust conclusion, it would be necessary to consider rollouts on a randomized map, as these would include more varied data.

Another option would be to consider the weight matrices used in the treelstm cells and see if there is any recognizable pattern there, as these are where left/right specific values can be encoded.

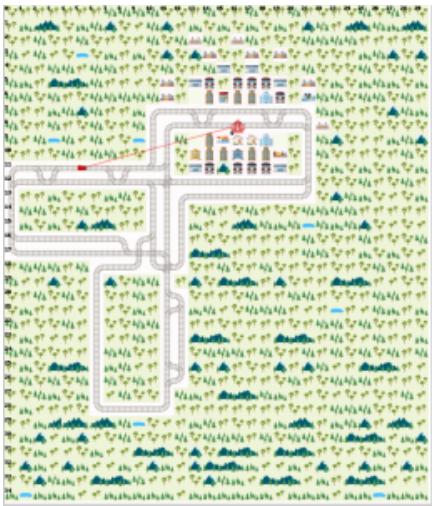
Investigate C-Utils Tree Generation in Flatland

The Flatland submission GitHub - RoboEden/flatland-marl: A multi-agent reinforcement learning solution to Flatland3 challenge. uses a custom observation generator written in C. Here we aim to understand the outputs generated by investigating the trees generated for a single agent in a random environment.

The environment was created using the demo script in the above repo, and investigated with manual calls to env.step() and debug_show(env).

Object	type	What
observations	tuple	
observations[0]	list	
observations[0][0]	list	
observations[1]	tuple	
observations[1][0]	list	node attributes over all agents
observations[1][0][0]	list	observations for all nodes for agent 0
observations[1][0][0][0]	list	observations for node 0
observations[1][1]	list	adjacency of tree for all agents
observations[1][1][0]	list	adjacency of tree for agent 0
observations[1][1][0][0]	list	first edge of adjacency list for agent 0
observations[1][2]	list	node order for all agents
observations[1][2][0]	list	node order for 0th agent
observations[1][3]	list	edge order for all agents
observations[1][3][0]	list	edge order for 0th agent

Description of Entries



Observations[0]: agent attributes

Information about the current state of the corresponding agent

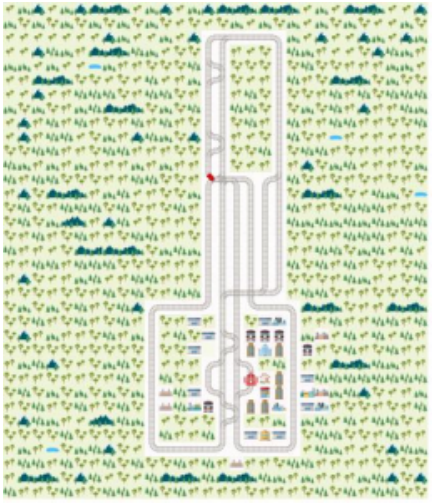
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	4
0	1	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	0	0	1	1	1	0	1	0

	State machine (aicrowd.com)						state one hot	road type doesn't make sense, flatland has 8 cell types										max nr malfunctions										initial direction				current direction				old direction?				moving	is_deadlocked	in_malfunction	malfunction_counter_complete	speed_cell_entry	speed_cell_exit	is_malfuction_state	is_off_map_state	is_na_ta

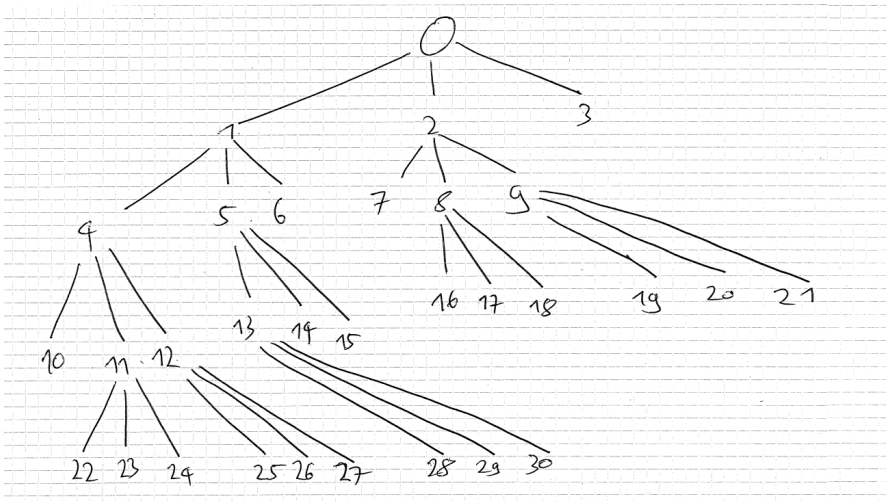
Observations[1]: Information about trees a tuple

Observations[1][0] attributes for the 31 nodes in the tree

General Structure of a Tree

	Flatland C-Utils (Challenge Paper)
number of nodes in a tree	fixed, in their case 31
number of children	each non-leaf node has **exactly 3 children**, corresponding to left, forward, right
Example	
Adjacency List	

Graph



Node Attributes

Nodes whose attributes are all -1 correspond to impossible choices (i.e. going right on a forward/left switch), but must be included in order to have a full ternary tree

	0	1	2	3	4	5	6	7	8	9	10	11
	distance to own target	distance to other targets	distance to other agents	distance to conflict	dist unusable switch	dist next branch	dist min to target	num agents same direction	num agents opposite direction	num agents malfunctioning	speed min fractional	num agents ready to depart
0	0	0	0	0	0	0	0.53846157	0	0	0	1	0
1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
2	-1	-1	-1	-1	-1	0.07692308	0.46153846	0	0	0	1	0
3	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
4	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
5	-1	-1	-1	-1	0.11538462	0.1923077	0.34615386	0	0	0	1	0
6	-1	-1	-1	-1	0.11538462	0.15384616	0.46153846	0	0	0	1	0
7	-1	-1	-1	-1	-1	0.46153846	0.15384616	0	0	0	1	0
8	-1	-1	-1	-1	0.23076923	0.46153846	0.07692308	0	0	0	1	0
9	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
10	-1	-1	-1	-1	0.1923077	0.26923078	0.34615386	0	0	0	1	0
11	-1	-1	-1	-1	-1	0.23076923	0.76923078	0	0	0	1	0
12	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
13	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
14	-1	-1	-1	-1	0.5	0.65384614	1.03846157	0	0	0	1	0
15	-1	-1	-1	-1	0.5	0.53846157	0.07692308	0	0	0	1	0
16	-1	-1	-1	-1	0.5	0.65384614	1.03846157	0	0	0	1	0
17	0.53846157	-1	-1	-1	-1	0.53846157	0	0	0	0	1	0
18	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
19	-1	-1	-1	-1	-1	0.53846157	0.15384616	0	0	0	1	0
20	-1	-1	-1	-1	0.30769232	0.53846157	0.07692308	0	0	0	1	0
21	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
22	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
23	-1	-1	-1	-1	-1	0.26923078	0.73076922	0	0	0	1	0
24	-1	-1	-1	-1	-1	0.38461539	1.07692313	0	0	0	1	0
25	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
26	-1	-1	-1	-1	0.69230771	0.96153843	0.73076922	0	0	0	1	0
27	-1	-1	-1	-1	0.69230771	0.73076922	0.96153843	0	0	0	1	0
28	-1	-1	-1	-1	0.57692307	0.73076922	1.03846157	0	0	0	1	0
29	0.61538464	-1	-1	-1	-1	0.61538464	0	0	0	0	1	0
30	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

Observations[1][1][0]:adjacency

Adjacency List of the nodes, so the first first row describes edge 0, which goes from node 0 to node 1 and corresponds to the action direction -1

	idx_parent	idx_node	action_direction
0	0	1	-1

1	0	2	0
2	0	3	1
3	2	4	-1
4	2	5	0
5	2	6	1
6	5	7	-1
7	5	8	0
8	5	9	1
9	6	10	-1
10	6	11	0
11	6	12	1
12	7	13	-1
13	7	14	0
14	7	15	1
15	8	16	-1
16	8	17	0
17	8	18	1
18	10	19	-1
19	10	20	0
20	10	21	1
21	11	22	-1
22	11	23	0
23	11	24	1
24	14	25	-1
25	14	26	0
26	14	27	1
27	15	28	-1
28	15	29	0
29	15	30	1

Node and Edge Order

Node and edge order are used in the TreeLSTM to determine the sequence of evaluation. The goal is that the leaf nodes pass through the LSTM cell first, and then the model goes upwards towards the roots.

observations[1][2]: node order

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
0	5	0	4	0	0	3	2	2	1	0	1	1	0	0	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

observations[1][3]: edge order

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29
0	5	5	5	4	4	4	3	3	3	2	2	2	2	2	2	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

TreeLSTM in Flatland MARL

In 2210.12933.pdf (arxiv.org), it is said that the use of a child-sum treeLSTM architecture enables the strong performance. However, the following experimental results cast serious doubt over these claims.

- In the constructed trees, all non-leaf nodes have exactly three children.
- When one permutes the order of these three childern (corresponding to actions left, forward, right) for each parent, the arrival rate drops to 0%, i. e. the network does not work at all.
- When one changes the encoding the adjacency list uses to show if an edge corresponds to the action left, forward or right (encoded via a column containing -1, 0 or +1 in the adjacency list), there is no noticable difference in network performance.

These results suggest that the order of the children is important to the model. Coupled with the constant number of children, this looks more similar to the architecture of a N-ary tree LSTM than a child-sum tree LSTM.

The goal of this page is to understand the TreeLSTM code published in GitHub - RoboEden/flatland-marl: A multi-agent reinforcement learning solution to Flatland3 challenge. and argue the hypothesis that the code is actually an implementation of a N-ary tree LSTM.

Functions

Name	Input dimension	Output Dimension	Type	Comment
self. W_iou	12, number of attributes for a node	3 * 128, tree embedding size	Linear	we output the same size as the tree-embedding output, because all the involved vectors need to have that length for element-wise addition and multiplication to be possible
self. U_iou	3 * 128	3 * 128	linear	
self.W_c	3* 128 *because of the three child nodes probably*	128	linear	
self.W_f	12	128		
self.U_f	128	128		

Calculation Example

For this example, let us assume that in our current iteration step, we have 100 parent nodes whose values we want to calculate. Collectively these parents have 300 child nodes. The internal represantations of a LSTM cell have 128 entries.

Shape of the adjacency list

The adjacency list is a matrix of dimensions n_agents * n_edges x 3. Rows come in packets of three, corresponding to three edges eminating from the same parent node (note that all non-leaf nodes in these trees must have 3 children). The indices refer to the row in which the node's information is stored in forest. Because of the modify_adjacency function, the adjacency lists of all trees are in one matrix, and all nodes have unique indices.

example block		
index of parent	index of first child	
index of parent	index of second child	
index of parent	index of third child	

Defining the masks

We define the node_mask and edge_mask of length batch_size = n_agents * n_edges. The masks help us only operate on the elements of matrices c and h that we are currently interested in, i.e. the 100 parents in question and their 300 children.

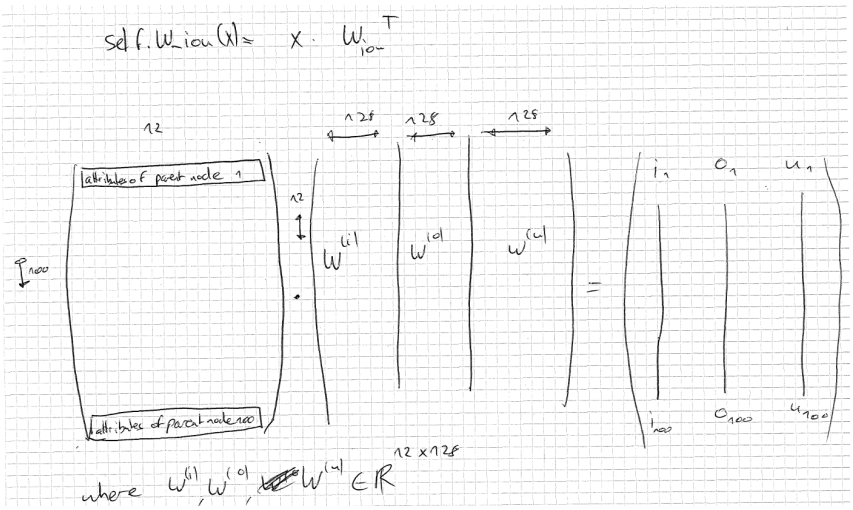
Propagation idea

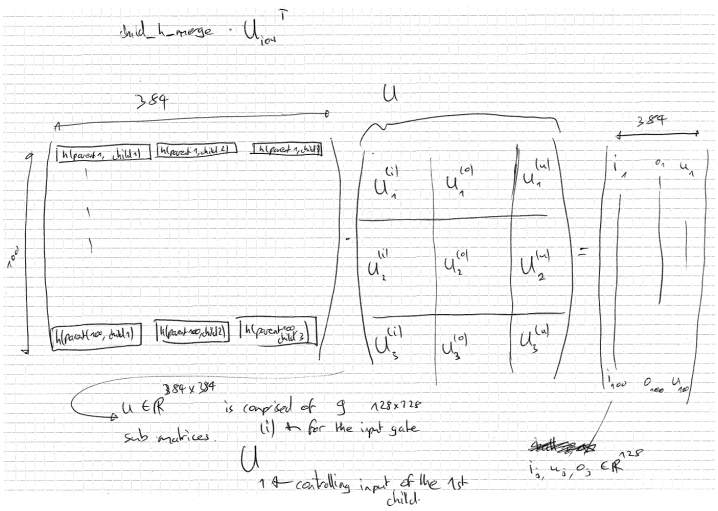
We have a total of batch_size nodes. For each of these nodes, we have a TreeLSTM cell. Note that these nodes belong to many different trees. Our goal is to iterate through these nodes propagating forward the information of the treeLSTM cells. We do this over all trees at once, with nodes being grouped not by their tree, but by their maximum distance to a leaf (i.e. the first step initializes the treeLSTM cells for the leaves, the second step does it for the parents of these nodes, etc).

One propagation step

We assume the more general case that iteration != 0, i.e. our nodes are not leaves.

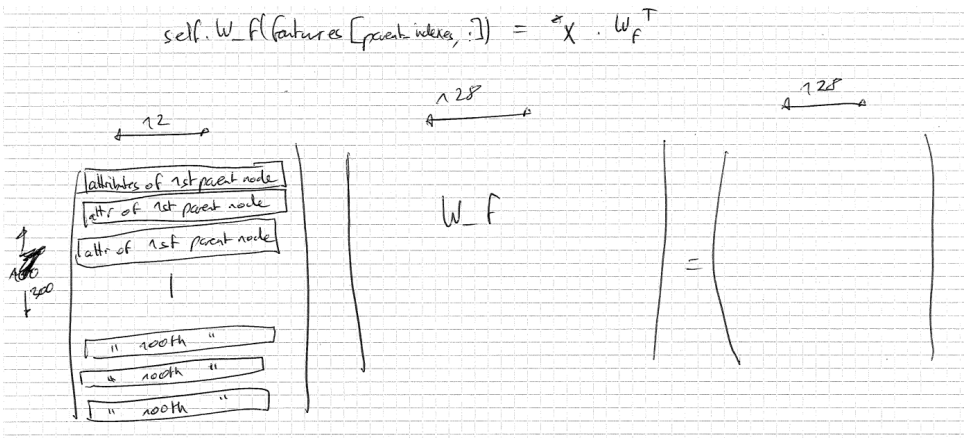
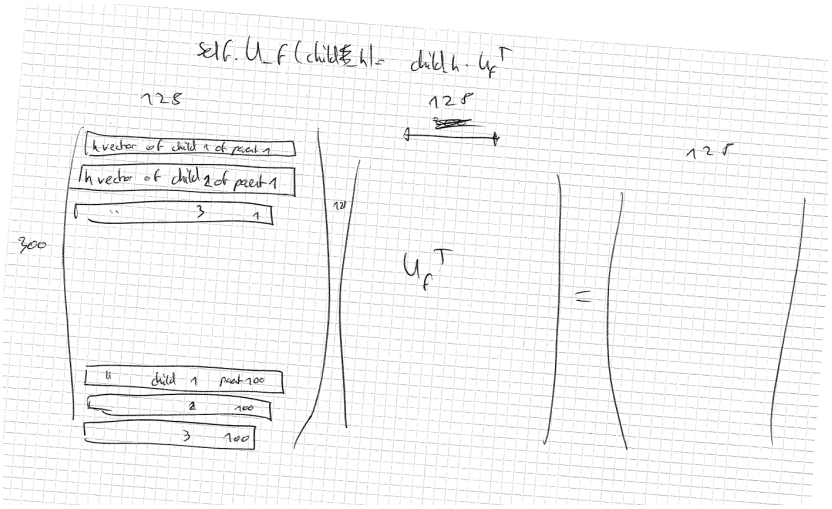
Calculating i, o, u

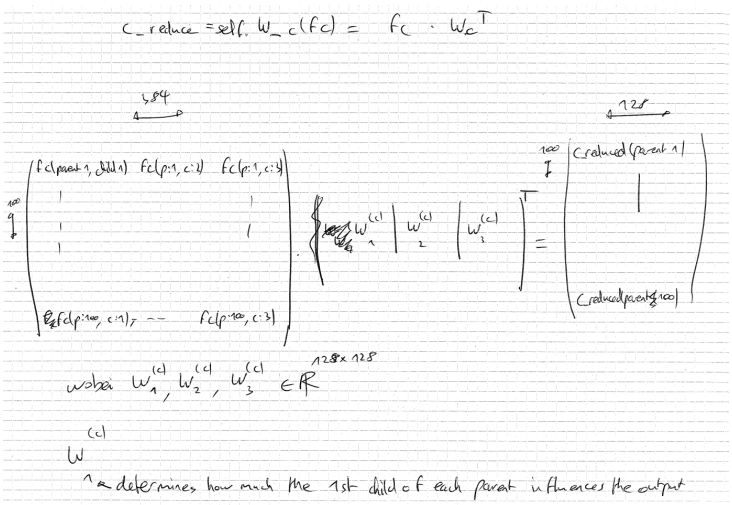
adjacency_list = adjacency_list[edge_mask, :]	We select only those edges from the adjacency list that we are interested in at this iteration step.	
parent_indexes = adjacency_list[:, 0]	The indexes of the parent nodes we are interested in at this time. Note that each index will appear 3 times consecutively.	
child_indexes = adjacency_list[:, 1]	The indexes of the child nodes we are interested in in this iteration. Note that packets of three consecutive child nodes all belong to the same parent.	
child_h = h[child_indexes, :]	child_h is a matrix of size edge_mask x 128, containing the h-vectors of all the children of the nodes we are trying to fill in this iteration.	torch.Size([300, 128])
child_c = c[child_indexes, :]	same but for c	torch.Size([300, 128])
child_h_merge = child_h.unflatten(0, (i_dims, 3))flatten(start_dim=1)	instead of having the three h-vectors of length 128 for the three children of each parents beneath each other in child_h, we rearrange them such that they appear as one concatenated vector, i.e. one row of child_h_merge has 384 entries and contains the h-vectors for the three children of that parent node one after the other.	torch.Size([100, 384])
self.W_iou(x)		100 x 384
self.U_iou(child_h_merge)		100 x 384

iou = self.W_iou(x) + self.U_iou(child_h_merge)		
i, o, u = torch.split(iou, iou.size(1) // 3, dim=1)	we cut iou into three separate matrices along its horizontal axis, see images above for it to make more sense	

Calculating f

Calculation of the forget gates follows a similar path as the calculation of i, o and u, but it is a little more involved in that

self.W_f(features[parent_indexes, :])	linear combo of each feature, but we repeat it 3x for each parent node 	300 x 128
self.U_f(child_h)		300 x 128

f = self.W_f(features[parent_indexes, :]) + self.U_f(child_h)	sum of the above two terms notice how here each row corresponds to a the sum of two linear combinations of one specific child node. There is no "mixing of information" across the child nodes at this point, i.e. it could not be the case that the value of x or h for one child node affects how strongly another child node is reflected in the parent's memory cell	300 x 128
f = torch.sigmoid(f) fc = f * child_c	apply sigmoid function and component-wise multiplication of memory cell of the children with the forget gate interesting: at this point, the f only depends on each child nodes' own hidden state, and not the other children of the same parent.	300 x 128
fc = fc.unflatten(0, (fc.shape[0] // 3, 3)).flatten(start_dim=1)	again, instead of having the information from the three child nodes of a parents as consecutive rows in matrix fc, we write it so that they are concatenated and used as one row.	100 x 384
c_reduce = self.W_c(fc)	Here we mix across the children of a parent. **This is where the order of the nodes comes into play.** 	100 x 128
c[node_mask, :] = i * u + c_reduce # here we calculate the sum	combination of input_gate * preliminary memory + forget_gate + child memory	100 x 128
h[node_mask, :] = o * torch.tanh(c[node_mask])		100 x 128

Conclusion

The implementation does a modified N-ary tree LSTM. While it is unclear if the changes have any advantages or disadvantages, the calculations show why the order of the branches is important, and why each parent must have exactly three children. This answers the question: How does Tree LSTM know whether the information comes from the left, forward or right branch?