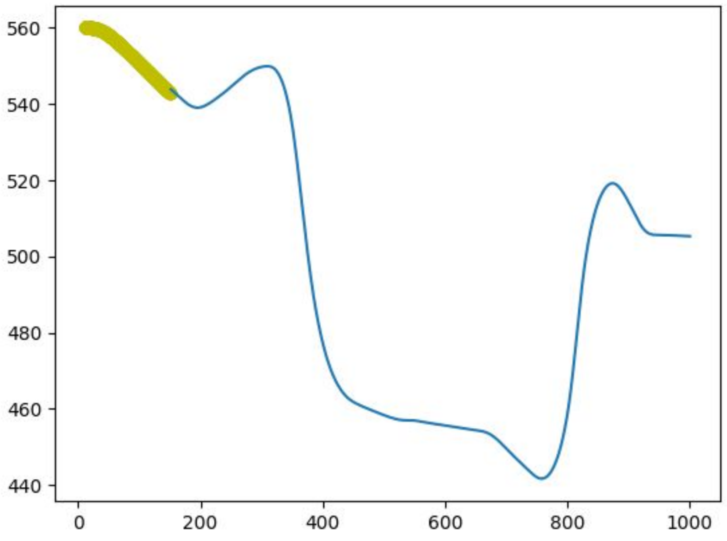
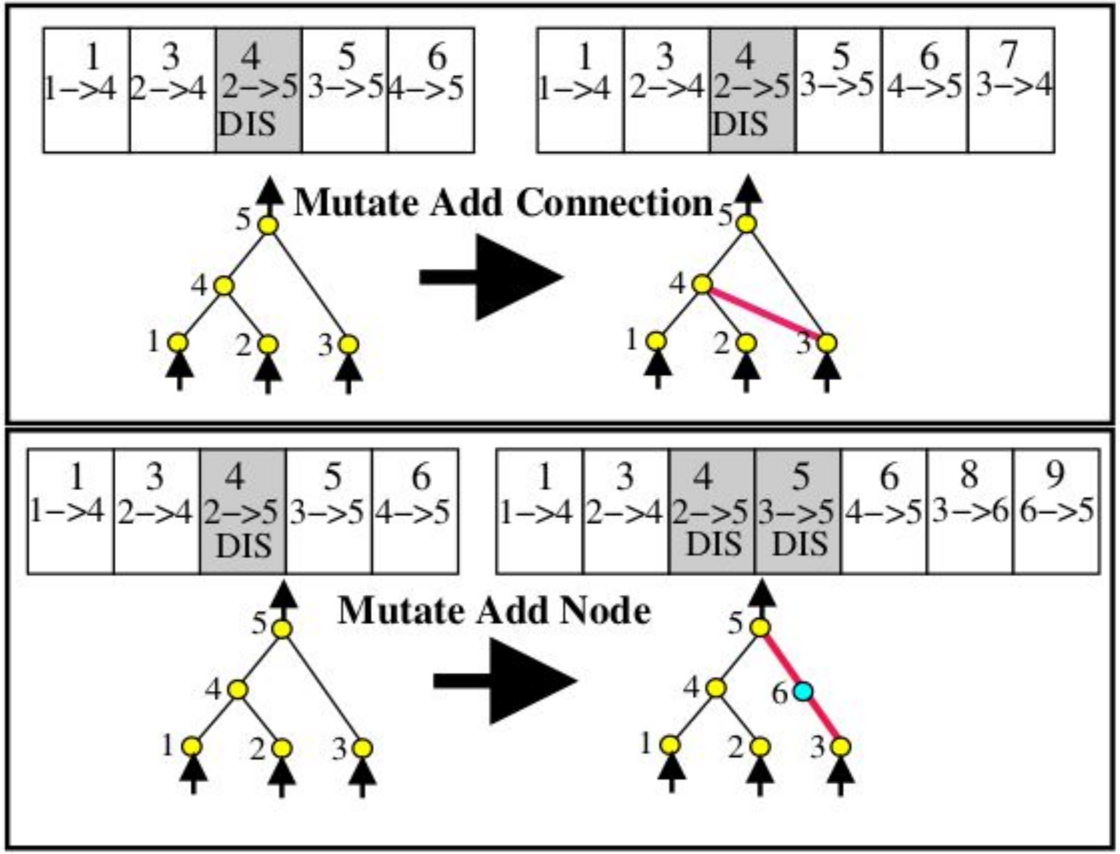
Trackmania Local Minima Optimizer

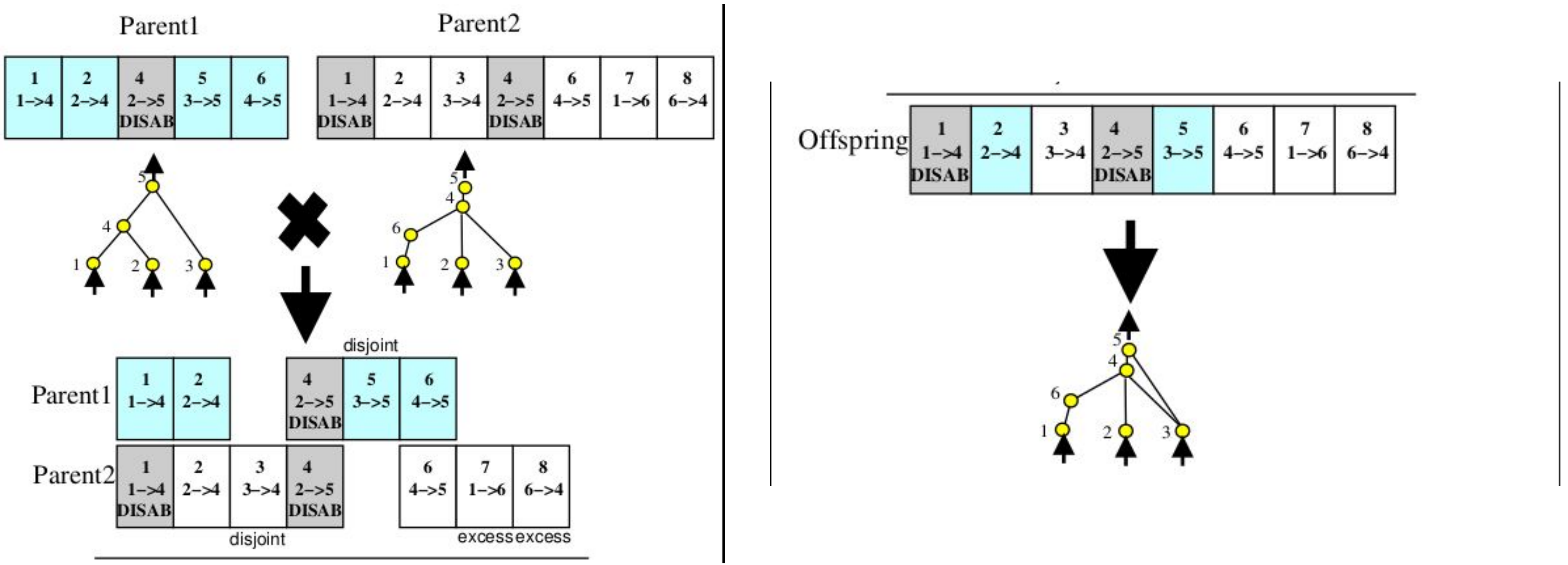
- Formula-type game, purpose: minimize time till finish.
- Discrete inputs: `Gas`, `Brake`, `Steer`.
- Previous supervised approach: train a `Runner` replay to tend to a `Racing Line` replay.



Related work: NEAT

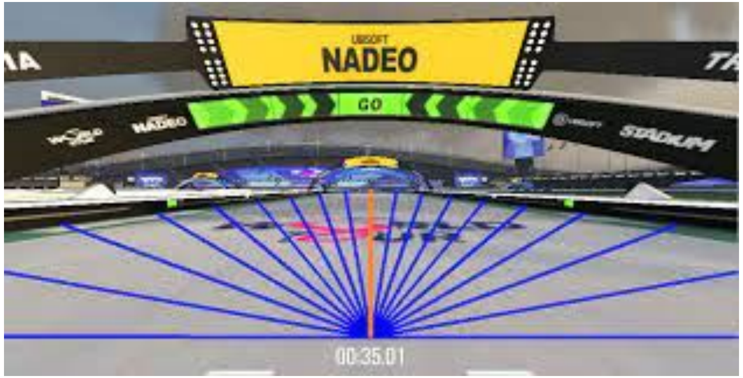


Related work: NEAT



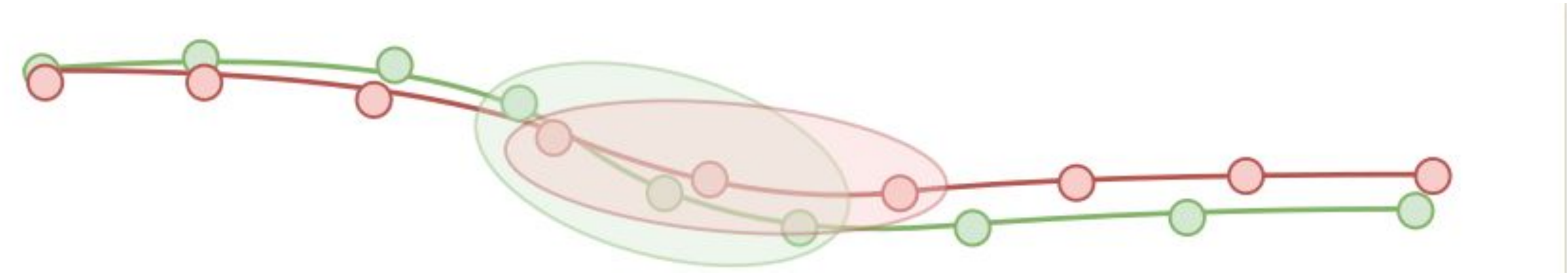
Related work: NEAT, Soft Actor-Critic, IQN

- NEAT: Current approaches: LIDAR input, fitness defined as speed reached / distance travelled.
- Slow to optimize, performs better than Supervised Learning, but worse than humans.
- SAC: Train a Critic network to estimate how much the car would further travel from a state s if it picks an action a.



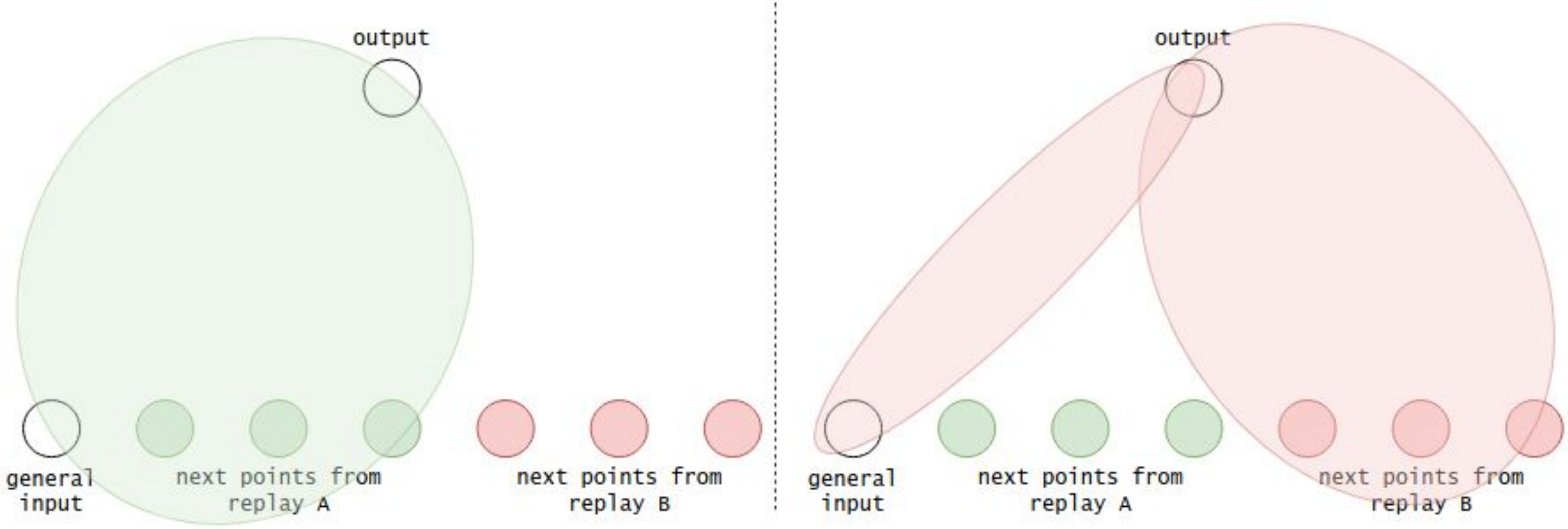
Scratchpad

- Take all K replays for the same map. [Dataset]
- For each replay, take equidistant points on the track, each two no closer than P.
- Use NEAT. A network should receive as input the speed of the car, and for each of the K replays, the relative positions of the next few points.



Scratchpad

- Pre-train the starting networks to follow specific replays.
- NEAT mutation would introduce dependencies between predictions from different replays (i.e. combine playstyles).
- Or start without pre-training and reward following the same track.
- Convolution layers?



Scratchpad

- Q-learning: policy follows the action distribution of close points.
- Reward points by increasing their influence.

Intermediate Project Presentation

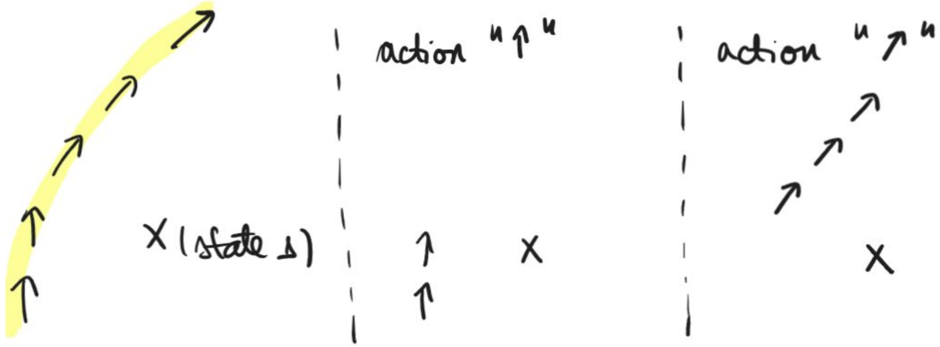
Current implementations:

- baseline: Q-Learning (tabular)
- NN as an approximator for the Q function (DQN)

Recap:

- No visual cues, only a set of human replays: the more the better, roughly following the same path.

Computing the reward function



- Take the closest **k** points to the current state: $(p_1, a_1), (p_2, a_2), \ldots, (p_k, a_k)$
- Compute the importance score for each point:

$$z_i = \exp\left(-\frac{\|p_i - s\|}{2\sigma^2}\right)$$

- Then the action *a* will be additionally rewarded:

$$f(s, a) = \frac{1}{k} \sum_{\substack{i=1 \\ a_i = a}}^{k} z_i$$

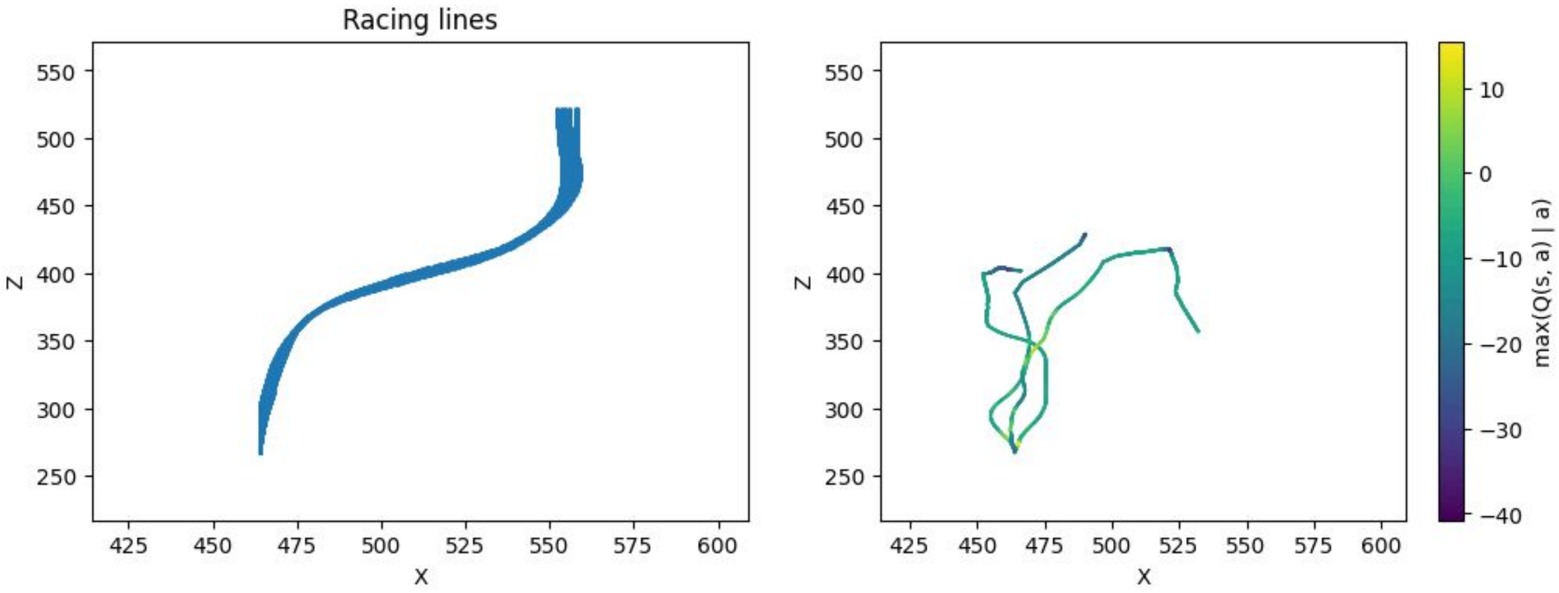
Computing the reward function

- Accounting for the past timestep: $r(s,a) = -1 + \beta f(s,a)$
- We only include any point in $f(s,a)$ at most once, obligating the agent to consider chains of actions that will lead him near other points from replays.

- We finally reward with 0 episodes that have timed out, or with a positive reward inversely scaling with time if the track was finished.

Q-Learning (tabular)

- We are obligated to take a very rudimentary state to not blow up the state space: only (x, y, z), rounded down to the first decimal.
- The algorithm learns surprisingly quickly, thanks to the reward function.



Q-Learning (tabular)

We will show three replays:

- One very early in training (~50 episodes in)
- One close to the end of training (~2300 episodes in)
- An argmax one at the end of training (3000 episodes in)

Q-Learning (tabular)

Q-Learning (DQN)

- The rounding in the tabular variant created discontinuities, which are difficult to mend.
- Easier to approximate the Q function with a NN. Can also introduce more complex state representations.

- Stochastic updates after each episode:

$$L(Q(s_i, a_i)) = \frac{1}{2}(Q(s_i, a_i) - r(s_i, a_i) - \max_{a'} Q(s_{i+1}, a'))^2$$

- Timing problems with the API

Further implementations

- Implement Experience Replay:
  - Playing Atari with Deep Reinforcement Learning
- Q-function approximator may be too optimistic:
  - Dueling Network Architectures for Deep Reinforcement Learning
- Boltzmann machines:
  - Reinforcement Learning with Factored States and Actions
- HyperNEAT:
  - A Neuroevolution Approach to General Atari Game Playing

https://github.com/vlad-ulmeanu01/tm_rl/tree/main/src

Final project presentation: Recap

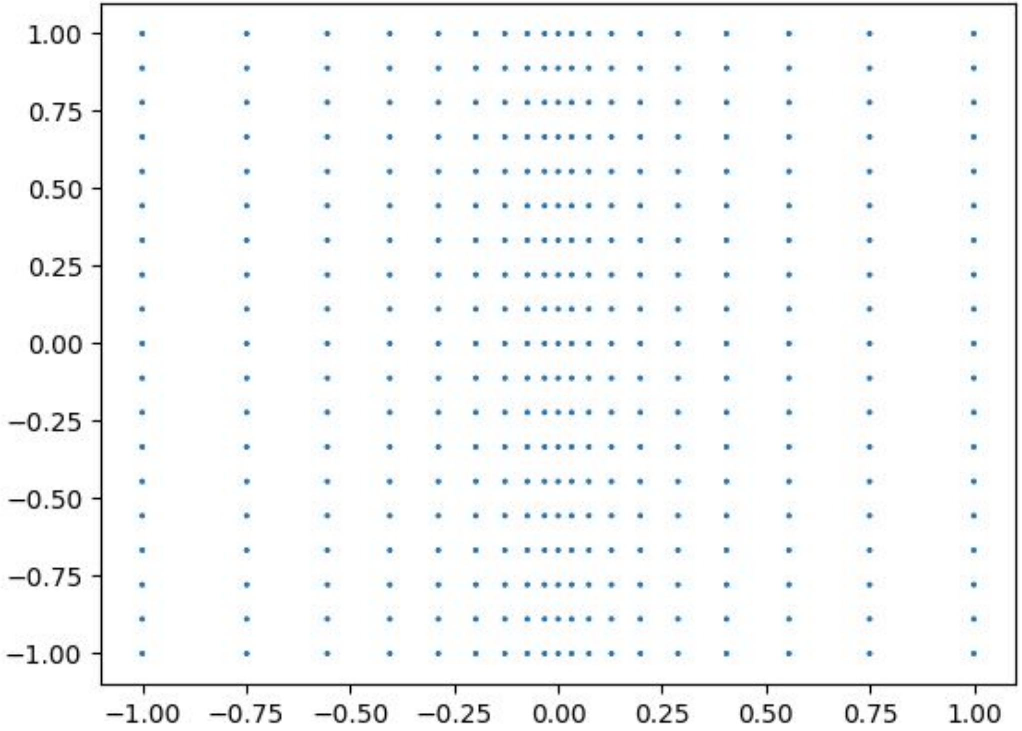
- Tabular Q-learning was ~1s off on test map from theoretical best.
  - State was represented by the position tuple, rounded down to the first decimal.
  - Problem with rounding to the Nearest Neighbour Q value.

- Deep Q Net implementation: simply representing the position as the state isn't feasible, hard to correctly approximate with a NN.

- Experimented with a JAX implementation, assumed that all new transitions must be passed through the net: no longer needed with the Experience Replay Buffer.

DQN: state description

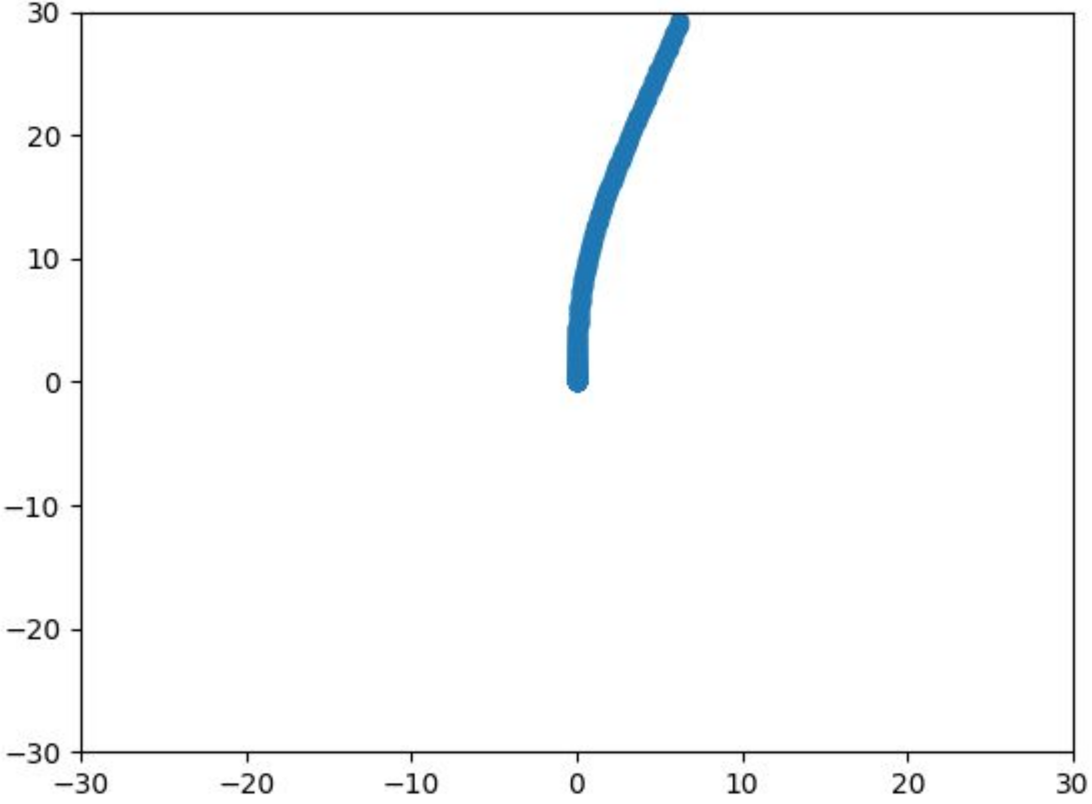
- We move the POV to the car. The state is made out of Replay points close to the current car position.



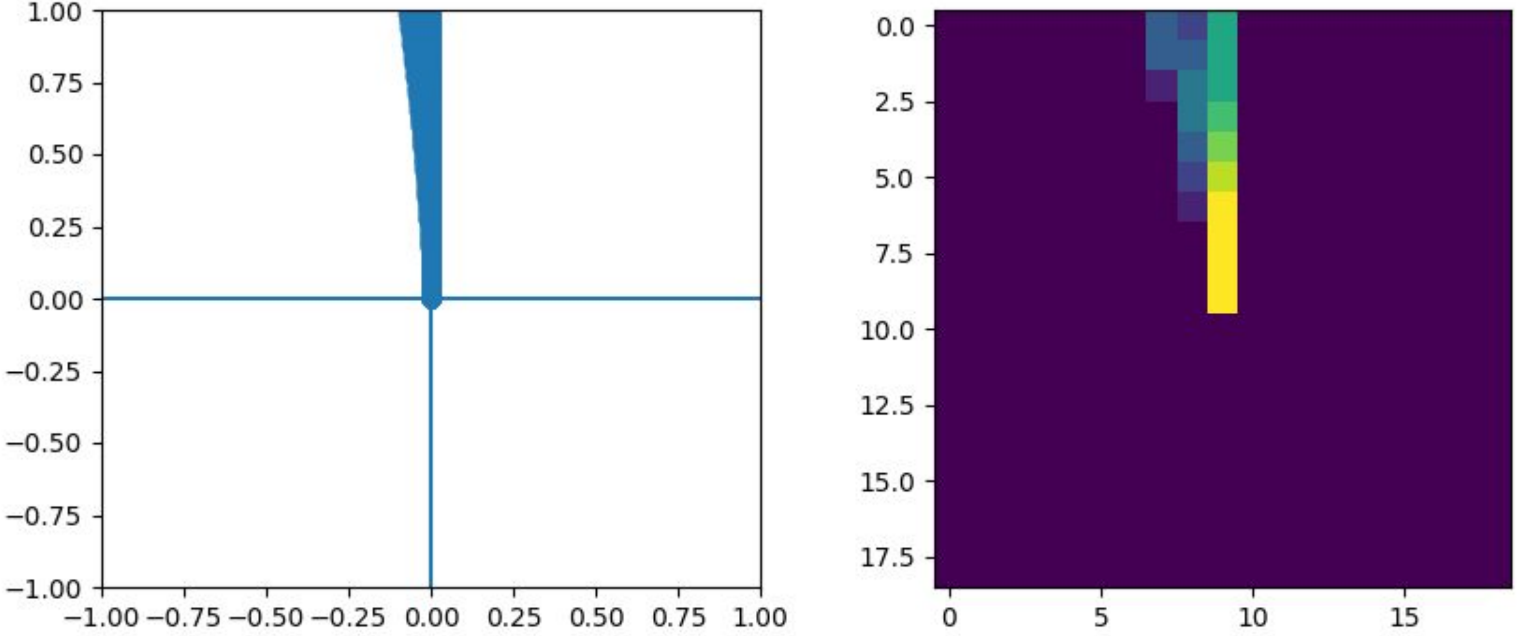
- XZ slice of state sensors.
- We generally expect the car to follow the replay surface, so we should have more points concentrated near the Z axis.
- There are `19 x 9 x 19` sensors in a State representation.
- Map at most one point from the same Replay to the same sensor.

DQN: state description

- State input: replay following itself.
- We never go too far from the Z axis (optimally).



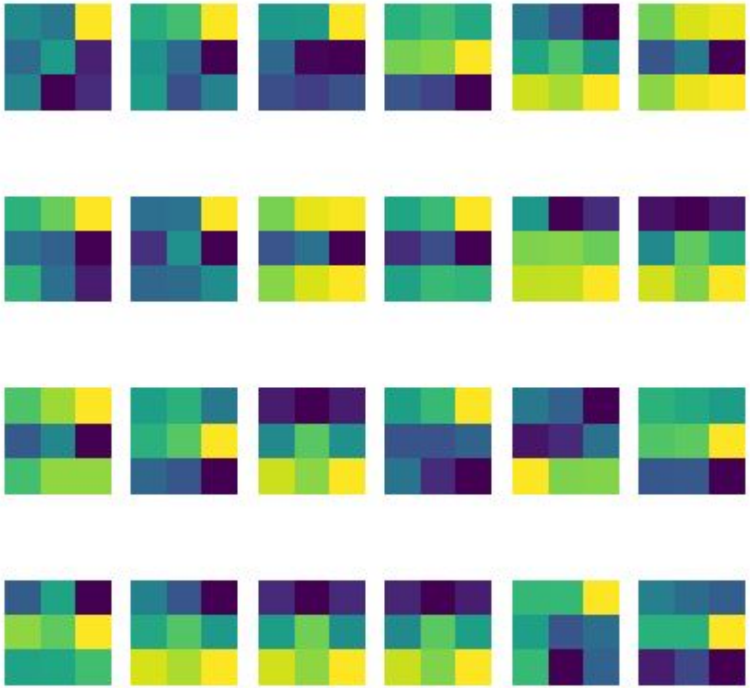
DQN: state description



- (left) Car against actual replay surface. (right) XZ slice of actual DQN input.
- Sensor is saturated when all Replays have a point mapped to it.

DQN: state description

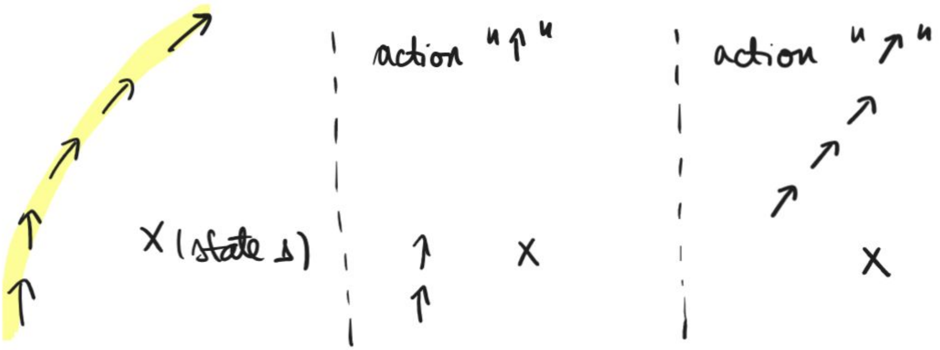
- Actual network has one or two consecutive State Images as an input.
- DQN has 1/2 Conv3d layers, followed by 2/1 Linear layers and a GAP layer.
  - Outputs for a state Q estimations for all actions in the given state.



- First Conv3d layer kernels. Either respond to corners, or horizontal lines.
- Strong response when a turn has to be made. (i.e. bend coming up, represented as a non-vertical line).

Reward function

- Split into two categories, noisy and non-noisy.
- Noisy rewards (continuous) are split again into:
  - Passive (reward per distance travelled between two frames)
  - Active 1 (reward for taking actions similar to the replays in places close to the replays)



  - Active 2 (reward an agent for reaching a Replay point sooner than the replay itself): only reward if certain criteria are met.

Reward function

- Non-noisy rewards (sparse). Reward an agent for reaching a checkpoint or for finishing a map.

- Designing the reward system was much more important than any architectural decision!
  - (as well as correctly backpropagating the reward as further as possible)
  - chose a high decay constant instead of any negative reinforcement

- Non-noisy rewards are (unfortunately) the best by far in determining the success of an episode.

Exploratory policy

- ε-greedy vs softmax action choice policy:

$$\pi_e(a \mid s) = \begin{cases} \operatorname{argmax}_{a'} \hat{Q}(s, a') & \text{with prob. } 1 - \epsilon \\ \text{uniform}(\mathcal{A}) & \text{with prob. } \epsilon, \end{cases}$$

$$\pi_e(a \mid s) = \frac{e^{\hat{Q}(s,a)/T}}{\sum_{a'} e^{\hat{Q}(s,a')/T}};$$

(d2l.ai pics)

- While ε-greedy can be easily implemented to incentivise exploring even for late episodes, the nature of the problem is better suited for softmax:
    - Wrongly choosing an action in a bad part of the track can ruin an episode.
    - There are many bad parts and (relatively) few episodes!

- We would rather choose to explore in areas where the preferred action is not evident.
    - Softmax focuses on exploiting, and may lead to streaks of episodes with highly (or even perfectly) similar behaviour.

Rainbow Optimizations: Priority Buffering

- We want to encourage exploitation on episodes that did well:
  - Separate Transition Buffer that remembers the best performing episodes. 25% of each batch is made with transitions from this.
    - Can quickly lead to overexploitation if the % is too large.
    - While this gives good improvements, the buffer is useless early on, and it becomes stale late in the run.
- Prioritise transitions whose network loss is high:

$$\delta_i = \delta_{s,a,r,s^{(\mathtt{next})}} = r_i + \gamma \cdot \max_{a'} \mathbf{net}_{\bar{\theta}}(s^{(\mathtt{next})}, a') - \mathbf{net}_\theta(s, a)$$

$$\mathtt{prio}_i = |\delta_i| + \epsilon$$

$$P(i) = \frac{\mathtt{prio}_i^\alpha}{\sum_i \mathtt{prio}_i^\alpha}$$

$$W(i) = \frac{1}{(N \cdot P(i))^\beta}$$

Rainbow Optimizations: Multi-step learning

- It is often detrimental to the agent to quickly change its action.
  - We let an agent change an action every `ExpDecay(50 -> 15)` moments.

- We sum up the rewards as:

$$R_t^{(n)} \equiv \sum_{k=0}^{n-1} \gamma_t^{(k)} R_{t+k+1} .$$

- Got better results in practice without the exponential decay
  - Correctness is lost when transitions from different action delays coexist in the transition buffer.

Rainbow Optimizations: Double Q nets

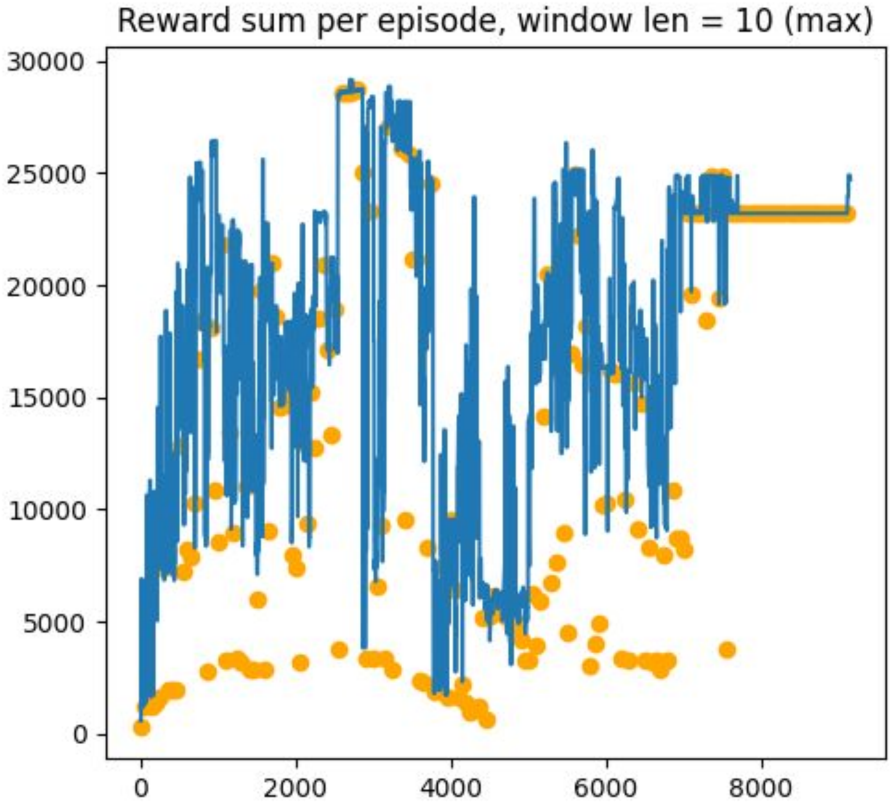
- Maximizing over a’ may lead to overestimation:

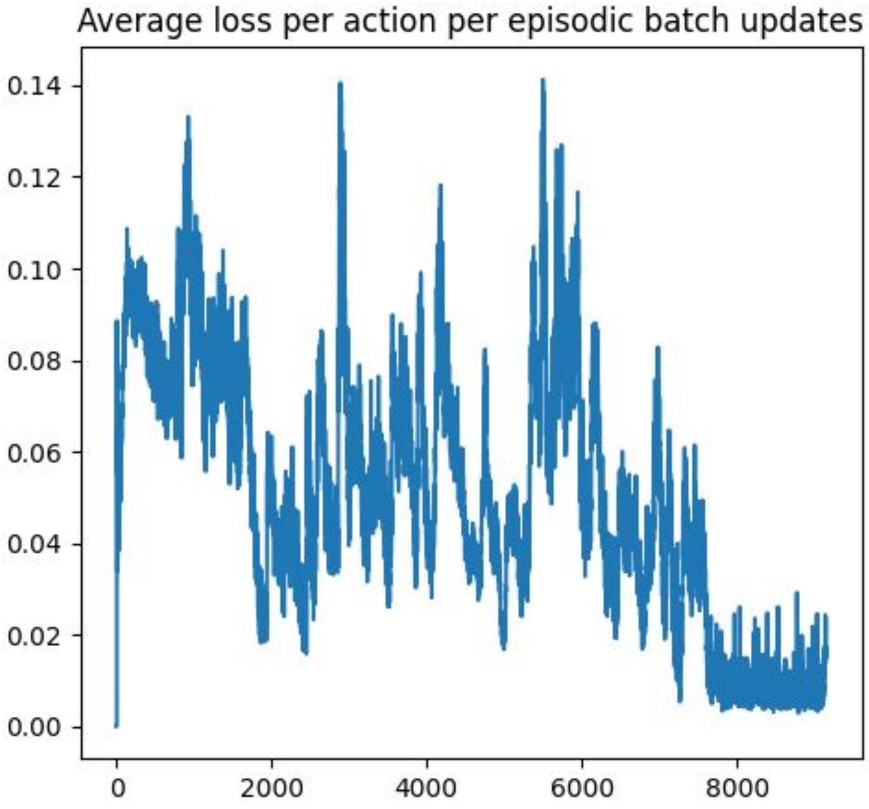
$$r_t + \gamma \cdot \mathbf{net}_{\bar{\theta}}(s_{t+1}, \arg\max_{a'} \mathbf{net}_{\theta}(s_{t+1}, a'))$$

- Let the offline net evaluate the best action given by the online net.

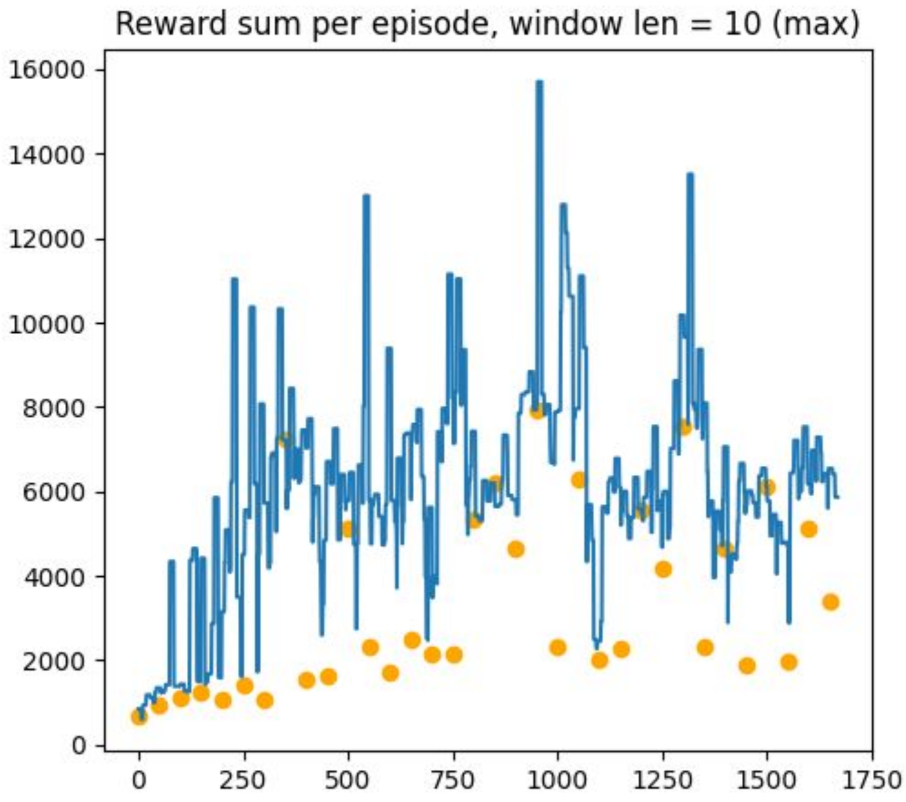
- No evident effects seen during runs.

Run results

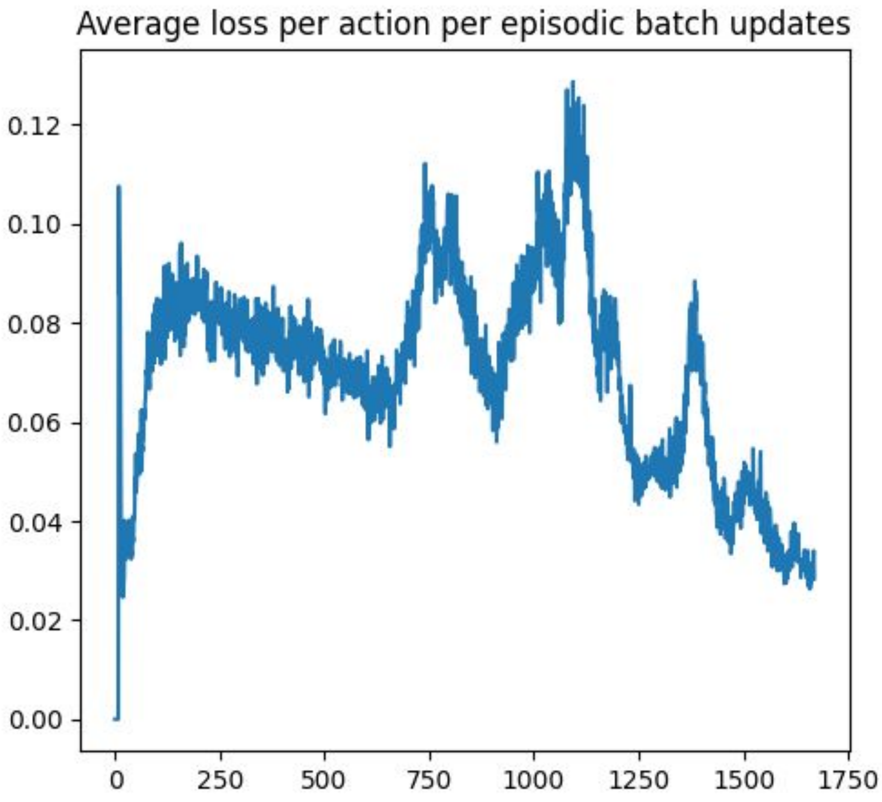

Reward sum per episode, window len = 10 (max)
30000
25000
20000
15000
10000
5000
0
0
2000
4000
6000
8000


Average loss per action per episodic batch updates
0.14
0.12
0.10
0.08
0.06
0.04
0.02
0.00
0
2000
4000
6000
8000

Run results

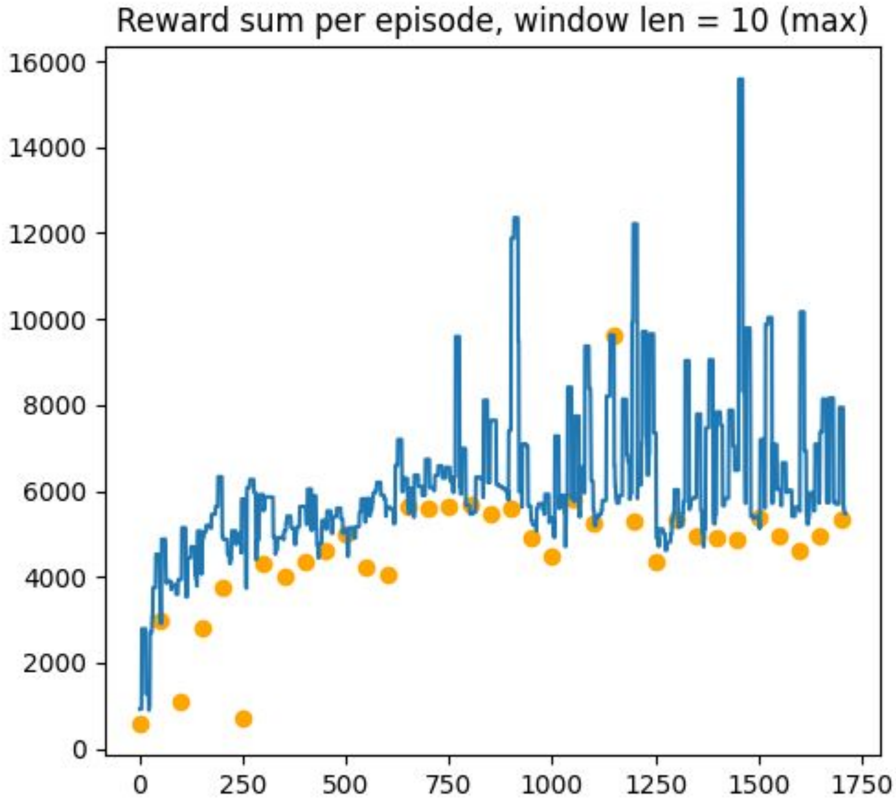


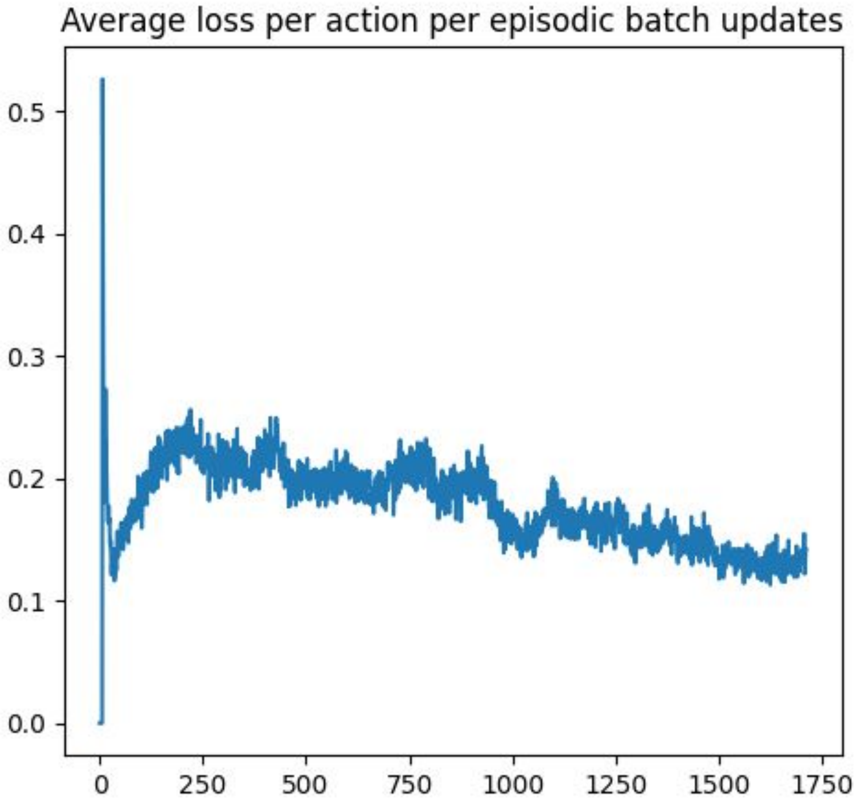
Reward sum per episode, window len = 10 (max)
16000
14000
12000
10000
8000
6000
4000
2000
0
0
250
500
750
1000
1250
1500
1750



Average loss per action per episodic batch updates
0.12
0.10
0.08
0.06
0.04
0.02
0.00
0
250
500
750
1000
1250
1500
1750

Run results


Reward sum per episode, window len = 10 (max)
16000
14000
12000
10000
8000
6000
4000
2000
0
0
250
500
750
1000
1250
1500
1750


Average loss per action per episodic batch updates
0.5
0.4
0.3
0.2
0.1
0.0
0
250
500
750
1000
1250
1500
1750


Trackmania

Conclusions

- Best run on small map 0.03s off my PB and 0.05s off WR.

- Implementation not suited for long chain dependencies between non-noisy rewards.

- Relatively good results considering the number of total seen states per run:
  - ~250K states with a 3K transition buffer.
  - (Rainbow) at least 7M states, 200M states with a 1M transition buffer, 80-200K states seen without any update done! (compared to 128)