Learning-Based Path Tracking for the ECE484 Crazyflie Race: Distilling MPCC into a 13 k-Parameter MLP with DAgger and PPO Fine-Tuning

ECE484 Crazyflie Drone project

November 9, 2026

Abstract

We replace the optimal control problem of our MPCC controller (`mpcc_report.pdf`) by small neural networks that map the reference path ahead of the drone, the MPCC weights along it and the drone state to the attitude/thrust command. Everything else of MPCC is kept: route, cubic-spline path, gate-aware contour weight, safe corridor, replanning when a gate is revealed and the disturbance observer. The networks are trained only on randomly generated tracks; the three provided tracks are used for testing only. Behaviour cloning of MPCC reaches 32 % success on held-out random tracks, DAgger with the MPCC in the loop 79 % (single 140-64-64-3 network, 13 k parameters). On the way we found and fixed a margin problem in the MPCC itself (its safe corridor left 1 mm between the drone's collision box and the gate frame), which raised the expert's own success on random tracks from 88 % to 98 % (nominal gates). PPO fine-tuning with our own reward and the MPCC as safety monitor did not improve the policy within our compute budget (two runs, documented). What did work is an ensemble of five independently trained 64×64 networks (67 k parameters in total): on the never-seen provided tracks it finishes **88/90** runs (MPCC: 90/90; single DAgger network: 74/90), with lap times within 0.3 s of MPCC, and 117/144 on held-out random tracks (MPCC: 124/144). A control step takes 0.7 ms (median, including features) vs. 3.5 ms for MPCC, and needs no solver. All settings, the reasons for them and all failures are documented here; the raw log with every number is `docs/distill_experiment_log.md`.

Contents

1 Goal and overall design **3**

2 Inputs and outputs of the network **3**
2.1 Why these inputs . . . 3
2.2 Why this output . . . 4
2.3 Network size . . . 5

3 Training data: random tracks **5**

4 Training: from behaviour cloning to DAgger **6**
4.1 Behaviour cloning and why it is not enough . . . 6
4.2 DAgger with the expert in the loop . . . 6
4.3 Loss . . . 6
4.4 Two hypotheses about label noise (both rejected) . . . 6

5 Improving the expert: the MPCC corridor had no margin **6**

6 DAgger results **7**
6.1 Test results . . . 8
6.2 Where the student fails . . . 8

7 RL fine-tuning with PPO **8**
7.1 Why RL after DAgger 8
7.2 Reward 9
7.3 Algorithm and settings 9
7.4 Results: no improvement 9

8 Ensembles: what finally worked **10**

9 Final test **10**

10 Limitations and next steps **11**

11 Deployment notes **11**

12 Reproducing **11**

1 Goal and overall design

MPCC solves, at every 10 ms step, a nonlinear program over a 1 s horizon (25 stages). It works well (90/90 on the provided tracks), but it needs acados, code generation and a few milliseconds per step. The goal is a controller that

1. flies like MPCC: same inputs, same command interface (roll, pitch, yaw = 0, collective thrust);
2. works for *any* path, like MPCC, i.e. does not memorize the three provided tracks;
3. is tiny and extremely fast (no optimization at run time);
4. is safe and smooth (it will be deployed on real hardware).

Design decision: distil only the optimizer. The planning half of MPCC (route, path, weights, corridor, replanning) is cheap and has no learning problem. Learning it would only make the network bigger and the task harder. We therefore keep it exactly (same code, `ece484_fly/learning/tracker.py`) and let the network replace only the solver. This also keeps the network interpretable: it is a *path tracker* that is told where the path goes and how precisely to follow it.

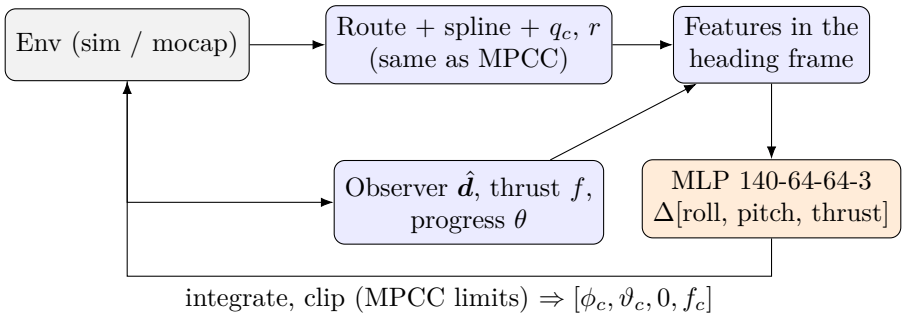


Figure 1: The distilled controller (`nn_tracker_controller.py`). Only the orange block is learned.

2 Inputs and outputs of the network

2.1 Why these inputs

The MPCC solution is a function of the initial state $\boldsymbol{x}_0$ and the per-stage parameters (path expansion, q_c, r^2, $\hat{\boldsymbol{d}}$, obstacles, speed caps) plus a few internal states (the last command, the thrust estimate, the progress θ, v_θ and the warm start). To make imitation a well-posed regression problem the network must see (almost) the same information. Everything is expressed relative to the drone so that the same situation on two different tracks looks the same to the network.

Table 1: The 140 features (`TrackerCore.observe`).

Block	Size	Content and reason
Path	20×3	$\boldsymbol{p}_d(\theta + k\,\Delta s) - \boldsymbol{p}$, $k = 0..19$, $\Delta s = 0.14\,\text{m}$, in the heading frame.
Contour weight	20	$(q_c - q_{\text{nom}})/q_{\text{gate}}$ at the same samples: ≈ 0 between gates, ≈ 1 at a gate. This is the "how tightly must I track here" signal of MPCC.
Corridor	20	$r(\theta)/0.25\,\text{m}$: how far the drone may deviate (small inside gates/near poles).
Speed cap	20	MPCC's progress speed cap /2.8 m/s (1.5 m/s near gates whose pose is not revealed yet, lower under a large disturbance estimate).
State	10	velocity (heading frame), roll, pitch, Euler rates, altitude z, thrust state f.
Last command	3	roll, pitch, thrust sent at the previous step (a state of MPCC, see below).
Disturbance	3	$\hat{\boldsymbol{d}}$ of the observer (heading frame): wind, mass error, battery sag.
Obstacles	4	xy offset of the two closest poles, clipped to 1.5 m.

Heading frame. Origin at the drone, x/y rotated by the drone's yaw, z up. We do not use the full body frame because gravity must stay along $-z$ (the thrust needed to hover does not depend on the tilt of the frame we describe the path in). Since MPCC commands yaw $= 0$ and the roll/pitch commands act in the yaw frame, this frame is also the frame of the outputs. No absolute x/y position is given, so the network cannot learn "where on the map" it is.

No gate orientation, no gate index. As in MPCC, a gate is not an input by itself. Its position and orientation are encoded in the path (the route crosses each gate perpendicularly through pre/post points) and its importance in q_c and r. This is what makes the controller usable for any path: if one wants a different trade-off, one changes q_c exactly like for MPCC.

Why 20 points, 0.14 m apart (arc length, not time). MPCC uses $N = 25$ stages of 40 ms, i.e. 1 s. Its stages are time samples whose spacing in space depends on the speed MPCC chooses, which the network does not know in advance. We therefore sample the path in arc length. The look-ahead $20 \times 0.14 = 2.8\,\text{m}$ equals the distance MPCC's horizon covers at its maximum speed of 2.8 m/s, so the network sees as far as MPCC. 0.14 m is fine enough to resolve the Gaussian weight bump at a gate ($\sigma = 0.25\,\text{m}$). (The number 20 was proposed by the user; 25 points at 0.112 m would cover the same distance and was not tested.)

Progress θ. The student keeps its own progress: a monotonic, forward-biased projection of the drone onto the path (window $[\theta - 0.1, \theta + 0.6]\,\text{m}$, widened to $[\theta - 0.3, \theta + 1.5]$ when more than 0.6 m away), re-projected after every replan. Monotonic because the MPCC report describes how a non-monotonic projection trapped the drone in tight loops.

2.2 Why this output

The output is the change of the [roll, pitch, thrust] command over one control period, i.e. MPCC's first optimal input $u_0\,\delta$ (MPCC's decision variables are command *rates*). It is clipped to MPCC's rate limits (8 rad/s, 6 N/s), integrated, and clipped to MPCC's bounds ($|\text{tilt}| \le 0.7\,\text{rad}$, $0.12 \le f_c \le 0.72\,\text{N}$); yaw is always 0.

- **Thrust is an output** (third component). Yaw is not, because MPCC always commands 0.
- Predicting the *change* makes the commands exactly as smooth and bounded as MPCC's, by construction, whatever the network outputs.
- Because MPCC's command depends on its previous command, the previous command must be an input (table 1); otherwise the target would not be a function of the input.

2.3 Network size

$140 \times 64 \times 64 \times 3$ with tanh (smooth, bounded), 13 379 parameters, as requested. We checked that this size is not the bottleneck: a 256×256 network on the same data reached a *higher* validation error ($1-R^2 = 0.197/0.229/0.162$ vs. $0.168/0.191/0.143$ for roll/pitch/thrust); the remaining $\approx$17 % of unexplained variance is not predictable from the inputs (it comes from MPCC states the network cannot see, mainly its progress speed v_θ, which differs from the drone's tangential speed by 0.47 m/s (std), and from the single SQP iteration per step). A forward pass takes 5.6 μs in numpy.

3 Training data: random tracks

Why random tracks. If we trained on the three provided tracks, a good test result would not show that the network can track *any* path. We train only on generated tracks and test on the provided ones (never seen) and on held-out generated tracks (different seeds).

Generator (`ece484_fly/learning/tracks.py`).

- 3–6 gates, placed by a random walk with 0.9–2.0 m steps whose heading turns by $\mathcal{N}(0, 1\,\text{rad})$ (80 %) or uniformly (20 %): track-like but varied. Gates at least 0.9 m apart (the frames are 0.72 m wide), centres within $|x| \le 1.95$, $|y| \le 1.0$ m.
- Pass direction: bisector of incoming and outgoing direction $+\mathcal{N}(0, 0.3)$, or fully random in 15 % of the gates (forces detours around a gate).
- Height: 0.7 or 1.2 m (the official short/tall gates) in 75 %, otherwise uniform in $[0.55, 1.35]$ m.
- 0–3 obstacle poles, 60 % of them placed 0.28–0.6 m beside the path (so MPCC's keep-out constraint matters); unused obstacle slots are parked in a far corner.
- Random start on the floor, $\ge$1.3 m from the first gate (like the provided tracks), yaw $\pm$0.8 rad.
- A track is accepted only if its MPCC reference path stays inside the flight volume ($\pm 2.4 \times \pm 1.4 \times [0, 1.85]$ m) with $\ge$0.15 m clearance to all frames, poles and obstacles, and curvature $\le 16\,\text{m}^{-1}$. The three provided tracks pass the same check.

Failures and changes. The first generator (uniform walk, 0.9–2.2 m, 3–7 gates) accepted 12 % of the samples and almost never a 7-gate track, and sometimes failed altogether; the heading-biased walk and at most 6 gates fixed that ($\approx$1 s per track). A first gate 1.2 m from the start made MPCC itself crash into it when it was shifted; hence the 1.3 m rule.

Conditions. Every episode uses the configs' default disturbances (action noise, random force noise of $\pm$0.1 N, mass $\pm$5 g, inertia, start pose) and one of: *base* (35 %); *shift* (45 %): gates moved by up to $\pm$0.15 m (xy), $\pm$0.1 m (z), $\pm$0.15 rad (yaw) and obstacles by $\pm$0.15 m, revealed only within the 0.7 m sensor range, so the path is replanned in flight; *shift+wind* (20 %): additionally 0.3–1.2 m/s wind, gusts up to 0.8 m/s and up to 12 % battery sag. 2 laps per training episode (the first lap has unrevealed gates, the second does not).

Simulator details that mattered. Only the number of gates and obstacles is compiled into the simulator; the layout lives in mocap bodies and is swapped at every reset (`env.unwrapped.track`). Environments are therefore cached per (gate count, condition) in every worker. Each worker uses 1.3–1.5 GB; with 14 workers and a cache of 6 environments the 30 GB laptop ran out of memory, so we use 12 workers and a cache of 2.

4 Training: from behaviour cloning to DAgger

4.1 Behaviour cloning and why it is not enough

Iteration 0: MPCC flies 500 random-track episodes and we regress its outputs. The network then flies on its own and makes small errors; these lead to states MPCC never visited, where the network was never trained, and the errors compound. Measured: 32 % success on held-out random tracks after behaviour cloning (table 3).

4.2 DAgger with the expert in the loop

DAgger [1] collects data with the *student* flying and asks the expert what it would have done in every visited state. Our implementation (`ece484_fly/learning/dagger.py`):

- MPCC runs in shadow mode at every step on the very same observation and its output is the label. Its internal command state is overwritten with the command that was actually sent (last command, thrust estimate), so that MPCC answers "what would you do from *here*".
- The student's features always come from `TrackerCore`, the same code as at deployment. During collection it adopts MPCC's reference path after every replan (identical planner), so the label and the features refer to the same path.
- **Takeover**: if the student leaves the MPCC corridor by more than 2 cm, flies faster than 4 m/s, or has not taken off after 0.5 s, MPCC flies for 1 s (still labelling) and then hands back. Episodes are not wasted and we get exactly the recovery data that is missing.
- 20 % of each iteration's episodes are MPCC episodes with command noise (DART [2]).
- All data of all iterations is aggregated and the network is retrained (warm-started).

DART noise bug. Our first command noise made MPCC crash in 28/28 episodes. MPCC integrates its command, so additive noise on the command becomes a random walk. We now add the *increments* of an Ornstein–Uhlenbeck process ($\sigma = 0.05$ rad, 0.02 N, $\tau = 0.15$ s), so the perturbation of the command state is the bounded OU process itself: 25/28 finished (93 % without noise).

4.3 Loss

Mean squared error on the standardized command change (inputs and outputs standardized with the training statistics, stored in the model file), AdamW (lr $2 \cdot 10^{-3}$, cosine decay, weight decay 10^{-5}), batch 2048, 40 epochs, 5 % of the episodes held out for validation. Training takes 10–90 s on the CPU. Each sample is weighted by $1 + 3 \max_{k<5} q_{c,k}$ (normalized q_c of the first five path samples): errors matter most at gates (section 6.2).

4.4 Two hypotheses about label noise (both rejected)

The validation error on student-visited states ($1-R^2 \approx 0.3$–0.5) is higher than on expert states (0.17). (i) MPCC's single real-time iteration starts from a warm start that assumed its own commands; we relabelled 120 identical student episodes with 3 SQP iterations: 0.417/0.503/0.285 vs. 0.447/0.522/0.260 — no real difference. (ii) MPCC's own progress θ could drift away from the student's: the difference has a std of 6.9 cm with the student flying vs. 5.6 cm with MPCC flying — not a student-specific mismatch. The states the student visits are simply harder (off-nominal, close to takeovers).

5 Improving the expert: the MPCC corridor had no margin

The student cannot be better than its teacher, so we measured the teacher on random tracks (`scripts/diagnose_mpcc.py`, 60 tracks, 2 laps): 53/60 (nominal) and 48/60 (shifted). **Every**

failure was a gate-frame hit, with the drone centre 0.07–0.11 m from the frame, at 1.8–3.0 m/s, often more than 1.5 s after the last reveal (so not caused by replanning). Track statistics (gate spacing, change of pass direction, curvature near gates) did not separate failed and successful tracks; the provided lemniscate has a higher curvature at its gates ($13.9\,\mathrm{m}^{-1}$) than the median failed track ($10.3\,\mathrm{m}^{-1}$).

The cause is geometric. The simulator's collision shape of the drone is a box with half extents $0.07 \times 0.07 \times 0.02$ m, i.e. a horizontal half-diagonal of 0.099 m. MPCC's corridor radius is $r = \mathrm{clr} - r_{\mathrm{drone}}$ with $r_{\mathrm{drone}} = 0.10$ m, so inside a gate ($\mathrm{clr} = 0.2$ m) the drone centre was allowed 0.10 m off-centre, leaving 1 mm. The corridor is a soft constraint, so small violations at speed ended in a frame hit. On the provided tracks this went unnoticed (10/10, but gate offsets up to 0.12 m in the MPCC report).

Table 2: MPCC on 60 random tracks (2 laps) and on the provided tracks (10 seeds, 5 laps).

Setting	random, nominal	random, shifted	provided tracks (3×3 conditions)
$r_{\mathrm{drone}} = 0.10$ (original)	53/60	48/60	90/90 (report)
$r_{\mathrm{drone}} = 0.13$ (adopted)	**59/60**	**52/60**	**90/90**, same lap times
$r_{\mathrm{drone}} = 0.13$, $v_{\mathrm{unrevealed}}$ 1.5→1.1 m/s	–	49/60	–

We adopted $r_{\mathrm{drone}} = 0.13$ m in `mpcc_controller.py` (and in the tracker settings) and updated the MPCC report. The remaining shifted-gate failures happen 0.3–0.9 s after a gate is revealed; a slower approach did not help (the drone is still at ≈1.4 m/s when it crashes), so they come from the mid-flight replan itself. We left that open. Then we restarted DAgger from scratch with the improved expert (run 2).

6 DAgger results

Table 3: Success on 48 held-out random tracks × 3 conditions (3 laps). Run 1: expert with $r_{\mathrm{drone}} = 0.10$, evaluated on the later test seeds; run 2: improved expert, evaluated on separate *validation* seeds (model selection), the test seeds were not touched.

Iteration	0	1	2	3	4	5	6	7	8
Run 1 (of 144)	42	60	84	–	102	103	93	94	101
Run 2 (of 144)	46	77	95	105	114	**115**	113	114	109

Run 1 changes after iteration 2 (section 6.2): takeover threshold corridor $+0.15 \rightarrow +0.02$ m, gate-weighted loss, 300→400 episodes per iteration. Run 2 used them from the start; it reaches a plateau of ≈79 % after four iterations (±5 points is the noise of 144 episodes). We selected iteration 5 (`models/final/single64_dagger.npz`). Collection statistics of run 2: the student finished 57 % of its episodes in iteration 1 and 85 % from iteration 5 on, with about 2 MPCC takeovers per episode.

6.1 Test results

Table 4: Test of the single DAgger network: provided tracks (never used in training), 10 seeds, 5 laps. Time: mean finish time of the successful runs (MPCC in brackets).

Track	nominal	shifted gates	shifted + wind/battery	time nominal [s]
circle	10/10	9/10	7/10	25.06 (25.21)
uturn	10/10	10/10	10/10	17.28 (17.08)
lemniscate	6/10	5/10	7/10	25.80 (26.09)
DAgger network		74/90		
MPCC		90/90		

Table 5: Test of the single DAgger network: 48 held-out random tracks × 3 conditions, 3 laps (test seeds).

Controller	nominal	shifted	shifted+wind	all	max gate offset	step time (median)
DAgger network	44/48	32/48	25/48	101/144	0.135 m	0.6 ms
MPCC	47/48	39/48	38/48	124/144	0.109 m	3.5 ms

The network is as fast through the course as MPCC (same lap times) and nearly as good on nominal tracks, but loses more episodes when gates are shifted and when there is wind.

6.2 Where the student fails

Run 1, iteration 1. All crashes were gate-frame hits with the drone 0.11–0.17 m from the gate centre, while its distance to the path was only 0.12–0.17 m: the student tracked the path roughly, but not precisely enough *inside gates*. Our takeover threshold (corridor +0.15 m = 0.25 m at a gate) never fired before these crashes, so the episodes ended and no recovery data was collected. Hence the tighter threshold and the gate-weighted loss.

Lemniscate. All 12 lemniscate failures of the DAgger network happen at two mirrored places: after the apex gate at 1.2 m, in a 90° turn that descends to a 0.7 m gate. The network drops the thrust to 0.22 N, saturates the roll at −0.7 rad, deviates 0.24 m from the path while sinking at 1.3 m/s and hits the *top* frame of the next gate (drone at 0.92 m, gate centre at 0.70 m). MPCC flies the same seeds without problems. This is a recovery-at-the-limit failure: the imitation loss does not know that this particular error is fatal. That motivates the RL fine-tuning.

7 RL fine-tuning with PPO

7.1 Why RL after DAgger

Imitation weights all command errors equally; a few centimetres matter only inside gates and in manoeuvres at the limit. Reinforcement learning optimizes what we actually want (pass the gates, do not crash, do not need rescuing) and can in principle exceed the teacher. Starting from the DAgger policy is essential: from scratch, RL would need orders of magnitude more samples.

7.2 Reward

The simulator's reward (`race_core.reward`) is -1 once the race is finished and 0 otherwise; its docstring explicitly says to define one's own. Ours (`ece484_fly/learning/rl.py`), per 10 ms step:

$$r_t = 5\,\Delta\theta \;-\; 0.05\,q_c\Big(\frac{\|\boldsymbol{p}-\boldsymbol{p}_d(\theta)\|}{0.1\,\mathrm{m}}\Big)^2 \;-\; 0.02\,\Big\|\frac{\Delta\boldsymbol{u}}{\Delta\boldsymbol{u}_{\max}}\Big\|^2 \;+\; 2\,(1-\mathrm{clip}(\tfrac{e_{\mathrm{gate}}-0.05}{0.10},0,1))\;[\text{gate passed}] \tag{1}$$

plus -20 on a crash (terminal), -10 when the drone leaves the MPCC corridor by 2 cm (the policy segment ends there, MPCC takes over for 1 s, then a new policy segment starts) and $+5$ for finishing. Reasons:

- **Progress** $\Delta\theta$ (arc length, same as MPCC's v_θ reward): about 0.1 per step at 2 m/s. Measured along the path, so cutting corners like MPCC is allowed.
- **Contour error weighted with** q_c: the same precision profile as MPCC, a dense signal exactly where the crashes happen.
- **Gate bonus** decreasing with the distance to the gate centre: full within 5 cm, zero beyond 15 cm (where frame hits start).
- **Command-rate penalty**: smoothness; the commands are already rate-limited.
- **Crash** -20**, corridor exit** -10: large compared to the progress reward of a second (10), so that slowing down to be safe is worth it (safety first).

7.3 Algorithm and settings

PPO [3] with a Gaussian policy in the network's normalized output space: the mean is the deployed MLP, one learnable log-std per output (deployment uses the mean, so the deployed network is unchanged in size and inputs). A separate 64×64 critic (training only). $\gamma = 0.99$ (1 s horizon), GAE $\lambda = 0.95$, advantages normalized. An imitation term $\beta\,\|\mu(x) - y_{\mathrm{MPCC}}\|^2$ on the MPCC labels of the visited states keeps the policy close to MPCC where the RL signal is weak. The critic is fitted alone for the first two iterations (otherwise the first policy updates use a random value function and can destroy the initial policy). 120 episodes (≈100 k policy steps) per iteration on 12 workers.

7.4 Results: no improvement

Table 6: PPO runs, starting from the selected DAgger policy (115/144 on the validation tracks). Training columns: stochastic policy, 120 episodes per iteration.

Run, iteration	lr / clip / epochs / β	KL per update	crashed	takeovers/ep	validation
1, it 4	10^{-4} / 0.1 / 5 / 0.5	0.004	0.12	2.29	116/144
1, it 9		0.004	0.13	2.18	114/144
2, it 4	$3 \cdot 10^{-4}$ / 0.2 / 10 / 0.1	0.029	0.20	2.31	110/144
2, it 9		0.031	0.13	2.17	110/144

Run 1 barely moved the policy (KL 0.004 per update: small learning rate, strong imitation anchor, few epochs). Run 2 moved it (KL 0.03, early stopping active) but not in a useful direction; the training-time crash rate fluctuated between 0.11 and 0.22 without trend. Both runs were stopped after 10 iterations (≈1 M policy steps, ≈30 min each).

Why it did not work (our interpretation). The failures that remain are rare (about one crash in 7 episodes) and are near-misses at a gate frame: the difference between a 10 cm and a 13 cm offset is one terminal event, preceded by hundreds of nearly identical steps. With ≈100 k steps per update, the advantage estimates of these few events are dominated by noise from the exploration and the disturbances. Making RL work would need much larger batches (vectorized simulation on

the GPU), a lower-variance credit assignment (e.g. rewarding the gate offset directly at every step close to a gate), or a safety-critic approach. Since the next idea (ensembles) solved the problem much more cheaply, we did not pursue this further. The code (`ece484_fly/learning/rl.py`, `scripts/rl_finetune.py`) is kept for future work.

8 Ensembles: what finally worked

Idea. Each network imitates MPCC with a residual error; with different random initializations and data orders, part of that error is independent between networks. Averaging the outputs cancels the independent part. In closed loop this matters most exactly where single networks fail: a few centimetres at the gates.

Table 7: Validation (48 random tracks × 3 conditions, validation seeds).

Policy	parameters	success (nominal / shifted / shifted+wind)
DAgger iteration 5 (single 64×64)	13.4 k	115/144 (44/38/33)
Average of DAgger iterations 4–8 (warm-started, correlated)	67 k	118/144 (46/38/34)
Single 64×64 trained from scratch on all data	13.4 k	113/144 (46/38/29)
5 independent 64×64, outputs averaged	67 k	**123/144** (47/39/37)
64×64 trained on the ensemble's outputs	13.4 k	115/144 (46/35/34)
128×128 trained on the ensemble's outputs	35 k	121/144 (46/38/37)

- Averaging warm-started iterations helps little (they are correlated); independent seeds help much more (+10 episodes over a single network).
- **Self-distillation**: we trained one network to reproduce the ensemble's averaged output on all 3.1 M stored states. The 64×64 network fits it well ($1-R^2 = 0.03$) but flies no better than a single network: the ensemble's advantage lives in closed loop, where independent errors cancel along the trajectory. A 128×128 network ($1-R^2 = 0.01$) keeps most of the gain. On smooth ensemble targets the larger network no longer overfits, unlike on the raw MPCC labels.

9 Final test

All numbers on seeds that were never used for any decision. Provided tracks: 10 seeds, 5 laps; random tracks: 48 tracks × 3 conditions, 3 laps.

Table 8: Final comparison.

Controller	parameters	provided tracks	random test tracks	step time
MPCC ($r_{\text{drone}} = 0.13$)	–	90/90	124/144 (47/39/38)	3.5 ms
Ensemble 5×64×64 (default)	67 k	**88/90**	**117/144** (47/36/34)	0.7 ms
Single 128×128 (from the ensemble)	35 k	85/90	111/144 (46/35/30)	0.6 ms
Single 64×64 (DAgger)	13 k	74/90	101/144 (44/32/25)	0.6 ms

Table 9: Provided tracks in detail: success and mean finish time (5 laps) of the ensemble, MPCC in brackets.

Track	nominal	shifted gates	shifted + wind/battery
circle	10/10, 25.09 s (25.21)	10/10, 24.71 s (24.93)	10/10, 27.38 s (26.92)
uturn	10/10, 17.39 s (17.08)	10/10, 17.67 s (17.36)	10/10, 19.46 s (19.07)
lemniscate	10/10, 26.36 s (26.09)	9/10, 26.71 s (26.35)	9/10, 30.08 s (29.13)

Summary. The ensemble flies the three tracks it has never seen almost like MPCC (88/90 vs. 90/90, same lap times within 0.3 s on nominal tracks, up to 1 s slower with wind), and reaches 94 % of MPCC's success on random tracks. Its remaining weakness is the same as MPCC's: gates that are shifted and revealed late (36/48 vs. 39/48), plus wind. Maximum gate offsets are 0.11–0.13 m (MPCC: 0.10–0.12 m). The step time is dominated by the features (spline evaluation, projection); the networks themselves take 30 μs.

10 Limitations and next steps

- The network can only be as good as MPCC; remaining failures of both are mostly late-revealed shifted gates. Improving MPCC's replanning (e.g. keep the old path and blend, or a curvature-aware speed cap after a reveal) would improve both.
- The crash rate is not zero: 2/90 on the provided tracks (both on the lemniscate with shifted gates), 19 % of the random test episodes, almost all with shifted gates or wind (MPCC: 14 %). For hardware, a supervisor that hands over to MPCC when the drone leaves the corridor (the takeover rule used for data collection) would combine the speed of the network with MPCC's recovery, at the cost of running both.
- RL fine-tuning needs a much larger sample budget (GPU-vectorized simulation) to be useful here.
- Delay, estimator noise and the real Crazyflie's deviations from the identified model are not part of the training; they should be added as randomizations before deployment.

11 Deployment notes

- **Gate poses on hardware** (`real_race_env.py`): exactly as in simulation. The nominal poses are the ones in the toml file. Motion capture measures the real gate and obstacle poses once at reset (`_update_track_poses`); `obs()` reports the nominal pose until the drone is within `sensor_range` (horizontal distance to the *real* pose), then the real one. No noise is added. So the gates have to be placed close to the toml poses (with `check_race_track` the deviation is checked against the configured tolerances); alternatively `scripts/save_track_as_config.py` writes the measured poses into a toml (nominal = real); `real_track_objects = false` uses the nominal poses everywhere.
- **Replanning latency**: the p99 of the control step reaches 0.1–0.3 s at a gate reveal, for both the network and MPCC, because the spline planner runs inside `compute_control`. On hardware this is a missed control deadline; the planner should run in a background thread (keep flying the old path until the new one is ready).
- **Sim-to-real**: the network was trained on the identified `so_rpy_rotor_drag` model with randomized mass, wind, force noise and gate shifts. Radio and estimator delay are not modelled (the MPCC report proposes forward prediction over the delay; the same applies here).
- The thrust estimate starts at 0 when the motors are unlocked, like for MPCC.

12 Reproducing

```
pixi shell
python scripts/sim.py --config circle.toml --controller nn_tracker_controller.py
python scripts/distill.py dagger --iterations 8 --episodes0 500 --episodes 400  # ~2 h
python scripts/distill.py evaluate --controller nn --suite configs
python scripts/distill.py evaluate --controller nn --suite random --n 48
python scripts/distill.py train --seed 1 --out ece484_fly/control/models/ens/seed1.npz  # x5
python scripts/rl_finetune.py --init ece484_fly/control/models/final/single64_dagger.npz
python scripts/diagnose_mpcc.py --n 60 --variants base,shift
```

Default model: the ensemble `ece484_fly/control/models/final/ensemble_member{1..5}.npz`. Another model can be selected in the config:

```
[controller] model = "ece484_fly/control/models/final/single128.npz"
```

(a comma-separated list of files is an ensemble).

References

[1] S. Ross, G. Gordon, D. Bagnell, "A reduction of imitation learning and structured prediction to no-regret online learning," *AISTATS*, 2011.

[2] M. Laskey, J. Lee, R. Fox, A. Dragan, K. Goldberg, "DART: Noise injection for robust imitation learning," *CoRL*, 2017.

[3] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov, "Proximal policy optimization algorithms," arXiv:1707.06347, 2017.