2019-01-19

Neuroflight: next generation flight control firmware

Neuroflight: Next Generation Flight Control Firmware

William Koch, Renato Mancuso, Azer Bestavros
Boston University
Boston, MA 02215
{wfkoch, rmancuso, best}@bu.edu

***Abstract*—Little innovation has been made to low-level attitude flight control used by unmanned aerial vehicles, which still predominantly uses the classical PID controller. In this work we introduce Neuroflight, the first open source neuro-flight controller firmware. We present our toolchain for training a neural network in simulation and compiling it to run on embedded hardware. Challenges faced jumping from simulation to reality are discussed along with our solutions. Our evaluation shows the neural network can execute at over 2.67kHz on an Arm Cortex-M7 processor and flight tests demonstrate a quadcopter running Neuroflight can achieve stable flight and execute aerobatic maneuvers.**

I. Introduction

Recently there has been explosive growth in user-level applications developed for unmanned aerial vehicles (UAVs). However little innovation has been made to the UAV's low-level attitude flight controller which still predominantly uses the classical PID controller. Although PID control has proven to be sufficient for a variety of applications, it falls short in dynamic flight conditions and environments (*e.g.* in the presence of wind, payload changes and voltage sag). In these cases, more sophisticated control strategies are necessary, that are able to adapt and learn. The use of neural networks (NNs) for flight control (*i.e.* neuro-flight control) has been actively researched for decades to overcome limitations in other control algorithms such as PID control. However the vast majority of research has focused on developing autonomous neuro-flight controller autopilots capable of tracking trajectories [1], [2], [3], [4], [5], [6], [7], [8]. An autopilot consists of an outer loop and an inner loop. The outer loop is responsible for generating attitude[1] and thrust command inputs to follow a specific trajectory. The inner loop is responsible for maintaining stable flight and for reaching the attitude set points over time through direct manipulation of the aircraft's actuators. Unlike the outer loop, the inner attitude control loop is mandatory for flight and can be used exclusively for manually piloting a UAV. Previous work does not address situations in which the neuro-flight controller autopilot can be overridden, which is essential to be used in practice. In this work we explore the adoption of neuro-flight control as an alternative to PID for inner loop flight control (*i.e.* attitude control). However in order to fully understand the performance implications of using NNs for flight control it is critical to study attitude control independently from trajectory planning.

[1]Defined as the orientation of the aircraft in terms of its angular velocity for each roll, pitch, and yaw axis.

In the spring of 2018 we released an OpenAI gym environment [9] called GymFC-v1 [10]. Via GymFC-v1 it is possible to train NNs attitude control of a quadcopter in simulation using reinforcement learning (RL). Neuro-flight controllers trained with Proximal Policy Optimization (PPO) [11] were shown to exceed the performance of a PID controller. Nonetheless the attitude neuro-flight controllers were not validated in the real world, thus it remains unknown if the trained networks are capable of flight.

In this work we make the following contributions. (1) We introduce Neuroflight, the first open source neuro-flight controller firmware for multi-rotors and fixed wing aircraft. The NN embedded in Neuroflight replaces attitude control and motor mixing commonly found in traditional flight control firmwares (Section III). (2) To train neuro-flight controllers capable of stable flight in the real world we released GymFC-v2, an update addressing several challenges in making the transition from simulation to reality (Section IV). (3) We propose a toolchain for compiling a trained NN to run on embedded hardware. To our knowledge this is the first work that consolidates a neuro-flight attitude controller on a microcontroller, rather than a multi-purpose onboard computer, thus allowing deployment on lightweight micro-UAVs (Section V). (4) Lastly, we provide an evaluation showing the NN can execute at over 2.67kHz on an Arm Cortex-M7 processor and flight tests demonstrate that a quadcopter running Neuroflight can achieve stable flight and execute aerobatic maneuvers such as rolls, flips, and the Split-S (Section VI). Source code for the project can be found at https://github.com/wil3/neuroflight and videos of our test flights can be viewed at https://www.youtube.com/playlist?list=PLqSAhwMPhV6tJJ1yCUIh0GcIVEOfyBD_S.

The goal of this work is to provide the community with a stable platform to innovate and advance development of neuro-flight control design for UAVs, and to take a step towards making neuro-flight controllers mainstream. In the future we hope to establish NN powered attitude control as a convenient alternative to classic PID control for UAVs operating in harsh environments or that require particularly competitive set point tracking performance (*e.g.* drone racing).

II. Background and Related Work

Since the dawn of fly-by-wire, flight control algorithms have continued to advance to meet increasing performance demands [12], [13], [14]. In recent years a significant amount of research has investigated the use of NNs for flight control which has advantages over classical control methods thanks to their ability to *learn* and *plan*.

Various efforts have demonstrated stable flight of a quadcopter through mathematical models using neuro-flight controllers to track trajectories. Online learning methods [2], [3] can learn quadcopter dynamics in real-time. Yet this requires an initial learning period and flight performance behavior can be unpredictable for rare occurring events. Offline supervised learning [1] can construct pre-trained neuro-flight controllers capable of immediate flight. However realistic data can be expensive to obtain and inaccuracies from the true aircraft can result in suboptimal control policies. RL is an alternative to supervised learning for offline learning. It is ideal for sequential tasks in continuous environments, such as control and thus an attractive option for training neuro-flight controllers. RL consists of an agent (*i.e.* NN) interacting with an environment to learn a task. At discrete time steps the agent performs an action (*e.g.* writes control signals to aircraft actuators) in the environment. In return the agent receives the current state of the environment (obtained from various aircraft sensors which typically becomes the input to the NN) and a numerical reward representing the action's performance. The agent's objective is to maximize its rewards.

Over time there has been a number of successes transferring controllers trained with RL to a UAVs onboard computer to autonomously track trajectories in the real world. Flight has been achieved for both helicopters [4], [5], [6] and quadcopters [7], [8]. Unfortunately none of these works have published any code thereby making it difficult to reproduce results and to build on top their research. Furthermore evaluations are only in respect to the accuracy of position therefore it is still unknown how well attitude is controlled.

In previous work [10] we proposed an RL environment, GymFC-v1, for developing attitude neuro-flight controllers which exceed accuracy of a PID controller in regards to angular velocity error. The GymFC-v1 environment uses the Gazebo simulator [15], a high fidelity physics simulator, which contains a digital replica, or *digital twin*, of the aircraft, fixed about its center of mass to the simulation world one meter above the ground allowing the aircraft to freely rotate in any direction. The angular velocity $\Omega(t) = [\Omega_\phi(t), \Omega_\theta(t), \Omega_\psi(t)]$ for each roll, pitch, and yaw axis of the aircraft is controlled by writing pulse width modulation (PWM) values to the aircraft actuators. The agent is trained using episodic tasks (*i.e.* a task that has a terminal state). At the beginning of an episodic task a desired angular velocity $\Omega^*(t)$ is randomly sampled. The goal of the agent is to achieve this velocity in a finite amount of time starting from still. At each time step an action $a(t) = [a_0(t), \ldots, a_{N-1}(t)]$ is provided by the agent where N is the number of aircraft actuators to be controlled (*e.g.* $N = 4$ for a quadcopter) and $a_i(t) \in [1000, 2000]$ represents the PWM value. The agent is returned the state $x(t) = (e(t), \omega(t))$ where $e(t) = \Omega^*(t) - \Omega(t)$ is the angular velocity error and $\omega(t) = [\omega_0(t), \ldots, \omega_{N-1}(t)]$ is the angular velocity of each actuator (*e.g.* for a quadcopter the RPM of the motor). Additionally a negative reward r is returned representing the angular velocity error. However evaluations were preformed in simulation thus it was unknown if neuro-flight controllers trained by GymFC-v1 could control a quadcopter in the real world.

In this work we pick up where GymFC-v1 left off. We explain in Section IV how without several modifications a NN trained with GymFC-v1 will not be able to achieve stable flight. With these modifications addressed in GymFC-v2 we were able to generate attitude neuro-flight controllers capable of high precision flight in the real world.

III. Neuroflight Overview

Neuroflight is a fork of Betaflight version 3.3.3 [16], a high performance flight controller firmware used extensively in first-person-view (FPV) multicopter racing. Internally Betaflight uses a two-degree-of-freedom PID controller (not to be confused with rotational degrees-of-freedom) for attitude control and includes other enhancements such as gain scheduling for increased stability when battery voltage is low and throttle is high. Betaflight runs on a wide variety of flight controller hardware based on the Arm Cortex-M family of microcontrollers. Flight control tasks are scheduled using a non-preemptive cooperative scheduler. The main PID controller task consists of multiple subtasks, including: (1) reading the remote control (RC) command for the desired angular velocity, (2) reading and filtering the angular velocity from the onboard gyroscope sensor, (3) evaluating the PID controller, (4) applying motor mixing to the PID output to account for asymmetries in the motor locations (see [10] for further details on mixing), and (5) writing the motor control signals to the electronic speed controller (ESC).

Neuroflight replaces Betaflight's PID controller task with a neuro-flight controller task. This task uses a single NN for attitude control and motor mixing. The architecture of Neuroflight decouples the NN from the rest of the firmware allowing the NN to be trained and compiled independently. The compiled NN is then later linked into Neuroflight to produce a firmware image for the target flight controller hardware.

To Neuroflight, the NN appears to be a generic function $y(t) = f(x(t))$. The input $x(t) = [e(t), \Delta e(t)]$ where $\Delta e(t) = e(t) - e(t-1)$. The output $y(t) = [y_0, \ldots, y_{N-1}]$ where N is the number of aircraft actuators to be controlled and $y_i \in [0, 1]$ is the control signal representing the percent power to be applied to the $i-th$ actuator. This output representation is protocol agnostic and is not compatible with NNs trained with GymFC-v1 whose output is the PWM to be applied to the actuator. PWM is seldomly used in high performance flight control firmware and has been replaced by digital protocols such as DShot for improved accuracy and speed [16].

At time t, the NN inputs are resolved; $\Omega^*(t)$ is read from the RX serial port which is either connected to a radio receiver in the case of manual flight or an onboard companion computer operating as an autopilot in the case of autonomous flight, and $\Omega(t)$ is read from the gyroscope sensor. The NN is then evaluated to obtain the control signal outputs $y(t)$. However the NN has no concept of thrust ($\mathbf{T}$), therefore to achieve translational movement the thrust command must be mixed into the NN output to produce the final control signal output to the ESC, $u(t)$. The logic of throttle mixing is to uniformly apply additional power across all actuators proportional to the available range in the NN output, while giving priority to achieving $\Omega^*(t)$. If any output value is over saturated (*i.e.* $\exists y_i(t) : y_i(t) \geq 1$) no additional throttle will be added. The input throttle value is scaled depending on the available output range to obtain the actual throttle value $\widehat{\mathbf{T}}(t) = \mathbf{T}(t)\left(1 - \max_i\{y_i(t)\}\right)$ where the function $\max$ returns the max value from the NN output. The readjusted throttle value is then proportionally added to each NN output to form the final control signal output $u_i(t) = \widehat{\mathbf{T}}(t) + y_i(t)$.

IV. GymFC-v2

In this section we discuss the enhancements made to GymFC-v2. These changes primarily consist of a new state representation and reward system. GymFC-v2 reinforces stable flight behavior through our reward system defined as $r = r_e + r_y + r_\Delta$. The agent is penalized for its angular velocity error, similar to GymFC-v1, along each axis with $r_e = -(e_\phi^2 + e_\theta^2 + e_\psi^2)$. However we have identified the remaining two variables in the reward system as critical for transferability to the real world and achieving stable flight. Both rewards are a function of the agents control output. First r_y rewards the agent for minimizing the control output, and next, r_Δ rewards the agent for minimizing oscillations.

Motor Velocity to Delta Angular Velocity Error GymFC-v1 returns the state $(e(t), \omega(t))$ to the agent at each time step. However not all UAVs have the sensors to measure motor velocity $\omega(t)$ as this typically involves digital ESC protocols. Even an aircraft with compatible hardware, the inclusion of the motor velocity as an input to the NN introduces additional challenges as it forces the development of an accurate propulsion system model for the digital twin. A mismatch between the physical propulsion system (*i.e.* motor/propeller combination) and the digital twin will result in the inability to achieve stable flight. Developing an accurate motor models is time-consuming and expensive. Specialized equipment is required to measure power consumption, torque, and thrust. To address these issues we investigated alternative environment states that did not rely on the motor model. Through experimentation we found that we could reduce the entire state to just angular velocity errors, by replacing $\omega(t)$ with the error differences $\Delta e(t)$. To identify the performance impact we trained a NN in an environment with $\omega(t)$ and compared this to a NN trained in an environment with $\Delta e(t)$. Both NNs were trained with PPO using hyperparameters from [10] for 10 million steps. After training, each NN was validated against 10 never before seen random target angular velocities. Results show the NN trained in an environment with $x(t) = (e(t), \Delta e(t))$ experienced on average 45.07% *less* error with only an increase of 3.41% in its control signal outputs. Furthermore we found using a history of error as input was also satisfactory however we opted to use the former as it did not require a variable size state to be maintained.

By modifying the environment state we discovered we also needed to modify the RL task. Training using episodic tasks, in which the aircraft is at rest and must reach an angular velocity never exposes the agent to scenarios in which the quadcopter must return to still from some random angular velocity. With the new state input consisting of the previous state, this is a significant difference from GymFC-v1 which only uses the current state. A continuous task is constructed to mimic real flight, continually issuing commands[2]. This task randomly samples a command and sets the target angular velocity to this command for a random amount of time. This command is then followed by an idle (*i.e.* $\Omega^* = [0, 0, 0]$) command to return the aircraft to still for a random amount of time. This is repeated until a max simulation time is reached.

Minimizing Output Oscillations In the real world high frequency oscillations in the control output can damage motors. Rapid switching of the control output causes the ESC to rapidly change the angular velocity of the motor drawing excessive current into the motor windings. The increase in current causes high temperatures which can lead to the insulation of the motor wires to fail. Once the motor wires are exposed they will produce a short and "burn out" the motor.

The reward system used by GymFC-v1 is strictly a function of the angular velocity error. This is inadequate in developing neuro-flight controllers that can be used in the real world. Essentially this produces controllers that closely resemble the behavior of an over-tuned PID controller. The controller is stuck in a state in which it is always correcting itself, leading to output oscillation.

In order to construct networks that produce smooth control signal outputs, the control signal output must be introduced into the reward system. This turned out to be quite challenging. Ultimately we were able to construct NNs outputting stable control outputs with the inclusion of the reward $r_\Delta = \beta \sum_{i=0}^{N-1} \max\{0, \Delta y_{\max} - (\Delta y_i)^2\}$ which is only applied if the absolute angular velocity error for every axis is less than some threshold (*i.e.* the error band). This allows the agent to be signaled by r_e to reach the target without the influence from this reward. Maximizing r_Δ will drive the agent's change in output to zero when in the error band. To derive r_Δ, the change in the control output y_i from the previous simulation step is squared to magnify the effect. This is then subtracted from a constant $\Delta y_{\max}$ defining an upper bound for the change in the control output. The `max` function then forces a positive reward, therefore if $(\Delta y_i)^2$ exceeds the limit no reward will be given. The rewards for each control output $N-1$ are summed

[2]Technically this is still considered an episodic task since the simulation time is finite. However in the real world flight time is typically finite as well.

and then scaled by a constant β, where $\beta > 0$. Using the same training and validation procedure previously discussed, we found a NN trained in GymFC-v2 compared to GymFC-v1 resulted in a 87.95% decrease in Δy.

Minimizing Control Signal Output Values Recall from Section II, that the GymFC-v1 environment fixes the aircraft to the simulation world about its center of mass, allowing it to only perform rotational movements. Due to this constrain the agent can achieve Ω^* with a number of different control signal outputs (*e.g.* when $\Omega^* = [0, 0, 0]$ this can be achieved as long as $y_0 \equiv y_1 \equiv y_2 \equiv y_3$). However this poses a significant problem when transferred to the real world as an aircraft is not fixed about its center of mass. Any additional power to the motors will result in an unexpected change in translational movement. This is immediately evident when arming the quadcopter which should remain idle until RC commands are received. At idle, the power output (typically 4% of the throttle value) must not result in any translational movement. Another byproduct of inefficient control signals is a decreased throttle range (Section III). Therefore it is desirable to have the NN control signals minimized while still maintaining the desired angular velocity. In order to teach the agent to minimize control outputs we introduce the reward function $r_y = \alpha\,(1 - \bar{y})$ providing the agent a positive reward as the output decreases. Since $y_i \leq 1$ we first compute the average output $\bar{y}$. Next $1 - \bar{y}$ is calculated as a positive reward for low output usage which is scaled by a constant α, where $\alpha > 0$. NNs trained using this reward experience on average a 90.56% decrease in their control signal output.

V. Toolchain

In this section we introduce our toolchain for building the Neuroflight firmware. Neuroflight is based on the philosophy that each flight control firmware should be personalized for the target aircraft to achieve maximum performance. To train a NN optimal attitude control of an aircraft, a digital representation (*i.e.* a *digital twin*) of the aircraft must be constructed to be used in simulation. This work begins to address how digital twin fidelity affects flight performance, however it is still an open question we will address in future work. The toolchain displayed in Fig. 1 consists of three stages and takes as input a digital twin and outputs a Neuroflight firmware unique to the digital twin. In the remainder of this section we will discuss each stage in detail.

Training The training stage takes as input a digital twin of an aircraft and outputs a NN trained attitude control of the digital twin capable of achieving stable flight in the real world. Our toolchain can support any RL library that interfaces with OpenAI environment APIs and allows for the NN state to be saved as a Tensorflow graph. Currently our toolchain uses RL algorithms provided by OpenAI baselines [17] which has been modified to save the NN state. In Tensorflow, the saved state of a NN is known as a checkpoint and consists of three files describing the structure and values in the graph. Once training has completed, the checkpoint is provides as input to Stage 2: Optimization.

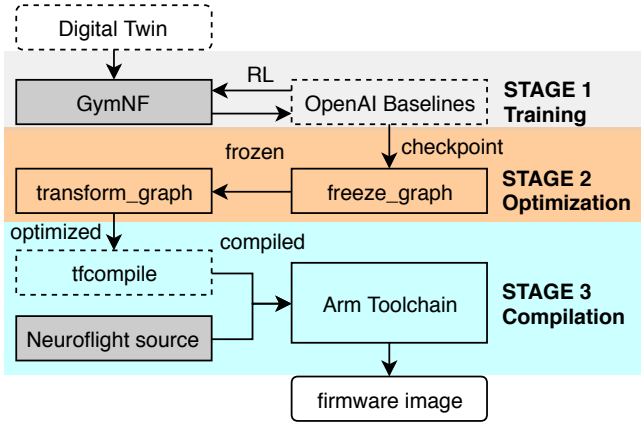


Fig. 1: Overview of the Neuroflight toolchain. Our main contributions are in the gray boxes while boxes with dashed borders indicate minor modifications to existing software.

Optimization The optimization stage is an intermediate stage between training and compilation that prepares the NN graph to be run on hardware. The optimization stage (and compilation stage) require a number of Tensorflow tools which can all be found in the Tensorflow repository [18]. The first step in the optimization stage is to *freeze* the graph. Freezing the graph accomplishes two tasks: (1) condenses the three checkpoint files into a single Protobuf file by replacing variables with their equivalent constant values (*e.g.* numerical weight values) and (2) extracts the subgraph containing the trained NN by trimming unused nodes and operations that were only used during training. Freezing is done with Tensorflow's `freeze_graph.py` tool which takes as input the checkpoint and the output node of the graph so the tool can identify and extract the subgraph.

Unfortunately the Tensorflow input and output nodes are not documented by RL libraries (OpenAI baselines [17], Stable baselines [19], TensorForce [20]) and in most cases it is not trivial to identify them. We reverse engineered the graph produced by OpenAI baselines (specifically the PPO1 implementation) using a combination of tools and cross referencing with the source code. A Tensorflow graph can be visually inspected using Tensorflow's Tensorboard tool. OpenAI baselines does not support Tensorboard thus we created a script to convert a checkpoint to a Probobuf file and then used Tensorflow's `import_pb_to_tensorboard.py` tool to view the graph in Tensorboard. Additionally we used Tensorflow's `summarize_graph` tool to summarize the inputs and outputs of the graph. Ultimately we identified the input node to be "pi/ob", and the output to be "pi/pol/final/BiasAdd".

Once the graph is freezed, it is optimized to run on hardware by running the Tensorflow `transform_graph` tool. Optimization provided by this tool allows graphs to execute faster and reduce its overall footprint by further removing unnecessary nodes. The optimized frozen ProtoBuf file is provided as input to Stage 3: Compilation.

Compilation A significant challenge was developing a

method to integrate a trained NN into Neuroflight to be able to run on the limited resources provided by a microcontroller. The most powerful of the microcontrollers supported by Betaflight consists of 1MB of flash memory and a Cortex-M7 processor with a clock speed of 216MHz [21]. Recently there has been an increase in interest for running NNs on embedded devices but few solutions have been proposed. We found Tensorflow's tool `tfcompile` to work best for our toolchain. `tfcompile` provides ahead-of-time (AOT) compilation of Tensorflow graphs into executable code primarily motivated as a method to execute graphs on mobile devices. Normally executing graphs requires the Tensorflow runtime which is far too heavy for a microcontroller. Compiling graphs using `tfcompile` does not use the Tensoflow runtime which results in a self contained executable and a reduced footprint.

Tensorflow uses the Bazel [22] build system and expects you will be using the `tfcompile` Bazel macro in your project. Neuroflight on the other hand is using `make` with the GNU Arm Embedded Toolchain. Thus it was necessary for us to integrate `tfcompile` into the toolchain by calling the `tfcompile` binary directly. When invoked, an object file representing the compiled graph and an accompanying header file is produced. Examining the header file we identified three additional Tensorflow dependencies that must be included in Neuroflight (typically this is automatically included if using the Bazel build system): the AOT runtime (`runtime.o`), an interface to run the compiled functions (`xla_compiled_cpu_function.o`), and running options (`executable_run_options.o`) for a total of 24.86 kB. In Section VI we will analyze the size of the generated object file for the specific neuro-flight controller.

To perform fast floating point calculations Neuroflight must be compiled with Arm's hard-float application binary interface (ABI). Betaflight core, inherited by Neuroflight already defines the proper compilation flags in the Makefile however it is required that the entire firmware must be compiled with the same ABI meaning the Tensorflow graph must also be compiled with the same ABI. Yet `tfcompile` does not currently allow for setting arbitrary compilation flags which required us to modify the code. Under the hood, `tfcompile` uses LLVM for code generation. We were able to enable hard floating points through the ABIType attribute in the llvm::TargetOptions class.

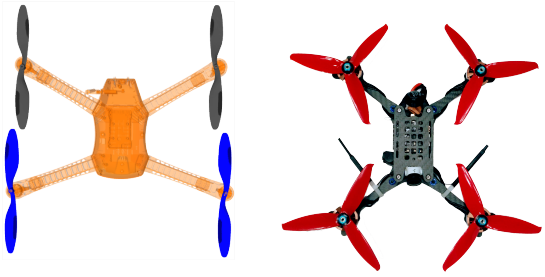
VI. Evaluation

In this section we evaluate Neuroflight controlling a high performance FPV racing quadcopter called NF1 and show it is capable not only of stable flight but also the ability to execute advanced aerobatic maneuvers. Images of NF1 and its entire build log have been published to RotorBuilds [23].

Firmware Construction We used the Iris quadcopter model included with the Gazebo simulator with modifications to the motor model for our digital twin. The digital twin motor model used by Gazebo is quite simple. Each control signal is multiplied by a maximum rotor velocity constant to derive the target rotor velocity while each rotor is associated with a PID controller to achieve this target rotor velocity. We obtained an estimated maximum 3,500 RPMs for our propulsion system from Miniquad Test Bench [24] to update the maximum rotor velocity constant. We also modified the rotor PID controller (P=0.01, I=1.0) to achieve a similar throttle ramp.

	Iris	NF1
Weight	1282g	432g
Wheelbase	550mm	212mm
Propeller	1047x2	5152x3
Motor	2830 850K_v	2204 2522K_v
Battery	3-cell 3.5Ah LiPo	4-cell 1.5Ah LiPo
Flight Controller	F4	F7

TABLE I: Comparison between Iris and NF1 specifications.



(a) Iris (b) NF1

Fig. 2: Iris simulated quadcopter compared to the NF1 real quadcopter

NF1 is in stark contrast with the Iris quadcopter model used by GymFC-v1 which is advertised for autonomous flight and imaging [25]. We have provided a visual comparison in Fig. 2 and a comparison between the aircraft specifications in Table I. In this table, weight includes the battery, while the wheelbase is the motor to motor diagonal distance. Propeller specifications are in the format "LLPPxB" where LL is the propeller length in inches, PP is the pitch in inches and B is the number of blades. Brushless motor sizes are in the format "WWHH" where WW and HH is the stator width and height respectively. The motors K_v value is the motor velocity constant and is defined as the inverse of the motors back-EMF constant which roughly indicates the RPMs per volt on an unloaded motor [26]. Flight controllers are classified by the version of the embedded Arm Cortex-M processor prefixed by the letter 'F' (*e.g.* F4 flight controller uses a Cortex-M4).

Our NN architecture consisted of 2 hidden layers with 32 nodes each using hyperbolic tangent activation functions. We trained the NN with the OpenAI Baseline version 0.1.4 implementation of PPO1 due to its previous success [10]. The reward system hyperparameters used were $\alpha = 300$, $\beta = 0.5$, and $\Delta y_{max} = 100^2$. We used the following PPO hyperparameters found by random search: a horizon of 500, an Adam stepsize set to 1e-4 linearly decayed through training, 5 epochs with minibatch sizes of 32, 0.99 discount and a Generalized Advantage Estimation (GAE) parameter of 0.95. The optimization stage reduced the frozen Tensorflow graph by 16% to a size of 12kB. The graph was compiled with Tensorflow version 1.8.0-rc1 and the firmware was compiled for

	Control Algorithm		Flight Control Task	
	Disarmed	Armed	Disarmed	Armed
Neuroflight	197 ± 0.03	200 ± 0.08	235 ± 0.08	281 ± 1.02
Betaflight	10 ± 0.02	11 ± 0.02	52 ± 0.06	97 ± 1.02

TABLE II: Timing analysis of the control algorithm and flight control task. All values are in microseconds.

the MATEKF722 target corresponding to the manufacturer and model of our flight controller MATEKSYS Flight Controller F722-STD. The final size of the firmware image is 913kB.

Timing Analysis Running a flight control task with a fast control rate reduces write latency to the ESC resulting in higher precision flight. However the latency of the ESC protocol places a limit on the control rate. Thus it is critical to analyze the execution time of the neuro-flight control task so the optimal control rate of the task can be determined. It is also important to identify which ESC protocol will provide the best performance. We collect timing data for Neuroflight and compare this to Betaflight for when the quadcopter is disarmed and also armed under load. We instrumented the firmware to calculate the timing measurement and wrote the results to an unused serial port on the flight control board. Connecting to the serial port on the flight control board via an FTDI adapter we are able to log the data on an external PC. We recorded 5,000 measurements and report the mean with a 95% confidence interval in Table II. Results show the neuro-flight control task's average execution time to be 281 ± 1.02 μs which allows the NN subtask to execute at 2.67kHz with 8kHz gyro updates which is far faster than what is required to achieve stable flight (for comparison, commercial quadcopters using the PWM ESC protocol have a max rate of 500Hz). Although the NN can execute faster, the NN subtask frequency is a division of the gyro update (in this case with denominator of three). This control rate is more than four times faster than the PWM ESC protocol (500Hz) therefore we configure Neuroflight to use the ESC protocol DShot600 which has a max frequency of 37.5kHz [27]. Given the simplicity of the PID algorithm it came as no surprise that it is significantly faster than the NN. However increasing the control rate too much can introduce additional noise [27]. As microcontrollers continue to increase in speed we will be able to keep increasing neuro-flight controller control rates to be on par with PID control.

Flight Evaluation To test the performance of Neuroflight we had an experienced drone racing pilot conduct test flights for us. Neuroflight supports real-time logging during flight allowing us to collect gyro and RC command data to analyze how well the neuro-flight controller is able to track the desired angular velocity. We asked the pilot to fly a mix of basic maneuvers such as loops and figure eights and advanced maneuvers such as rolls, flips, dives and the Split-S. To execute a Split-S the pilot inverts the quadcopter and descends in a half loop dive, exiting the loop so they are flying in the opposite horizontal direction. Once we collected the flight logs we played the desired angular rates back to the NN in the GymFC-v2 environment to evaluate the performance in simulation. Comparison between the simulated and real world performance is illustrated in Fig. 3 while specific maneuvers that occur during this test flight are annotated. Flight in the real world had an average absolute error $|\bar{e}|_{\text{real}} = [15.17, 21.05, 11.26]$ for the roll, pitch and yaw axis in degrees/s respectively while the GymFC-v2 simulation had an average absolute error $|\bar{e}|_{\text{sim}} = [2.88, 1.52, 4.07]$.

The increase in error is expected because the digital twin does not perfectly model the real system. The increased error on the pitch axis appears to be due to the differences in frame shape between the digital twin and real quadcopter, which are both asymmetrical but in relation to different axis. This discrepancy may have resulted in pitch control lagging in the real world as more torque and power is required to pitch in our real quadcopter. A more accurate digital twin model can boost accuracy. Furthermore, during this particular flight wind gusts exceeded 30mph, while in the simulation world there are no external disturbances acting upon the aircraft. In the future we plan to deploy an array of sensors to measure wind speed so we can correlate wind gusts with excessive error. As shown in the video, stable flight can be maintained demonstrating the transferability of a NN trained with our approach.

VII. Future Work and Conclusion

In this work we introduced Neuroflight, the first open-source neuro-flight control firmware for remote piloting multi-copters and fixed wing aircraft and its accompanying toolchain. There are three main directions we plan to pursue in future work: digital twin development, adaptive and predictive control, and continuous learning. The economic costs associated with developing neuro-flight control will foreshadow its future, whether it could be mainstream or for special purpose applications. In future work we will continue to investigate how the fidelity of a digital twin affects flight performance in an effort to reduce costs during development. With a stable platform in place we can now begin to harness the NN's true potential. We will enhance the digital twin to aid in adaptive control to account for excessive sensor noise, voltage sag, change in flight dynamics due to high throttle input, payload changes, external disturbances such as wind, and propulsion system failure. Our current approach trains NNs exclusively using offline learning. However to reduce the performance gap between the simulated and real world it is more likely a hybrid architecture will be necessary to provide continuous learning. Given the payload restrictions of micro-UAVs and weight associated with hardware necessary for online learning we will investigate methods to off-load learning to the cloud. We believe Neuroflight is a major milestone in neuro-flight control and will provide a foundation for next generation flight control firmwares.

References

[1] J. F. Shepherd III and K. Tumer, "Robust neuro-control for a micro quadrotor," in *Proceedings of the 12th annual conference on Genetic and evolutionary computation*. ACM, 2010, pp. 1131–1138.

[2] C. Nicol, C. Macnab, and A. Ramirez-Serrano, "Robust neural network control of a quadrotor helicopter," in *Electrical and Computer Engineering, 2008. CCECE 2008. Canadian Conference on*. IEEE, 2008, pp. 001 233–001 238.

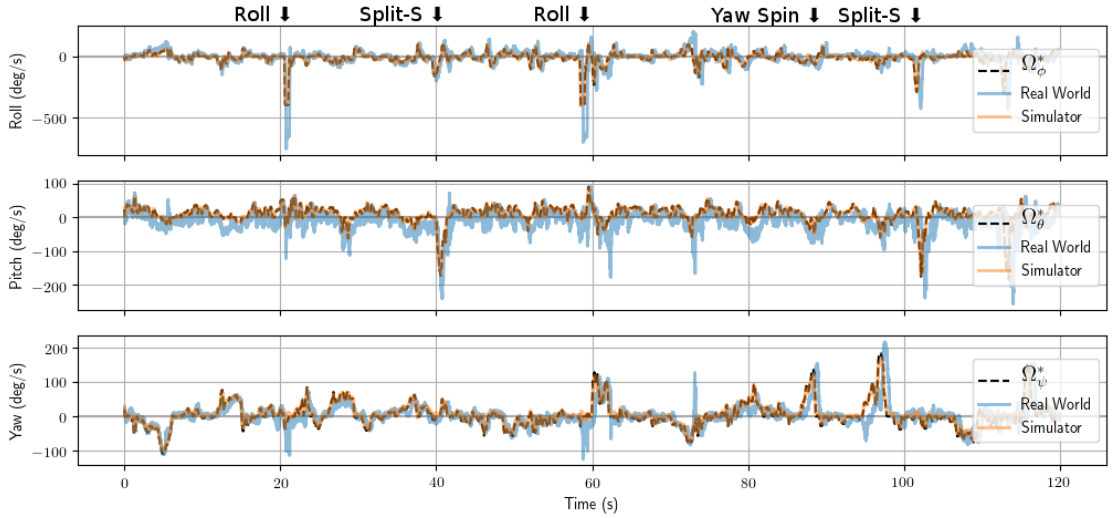


Fig. 3: Flight test log demonstrating Neuroflight tracking a desired angular velocity in the real world compared to in simulation. Maneuvers during this flight are annotated.

[3] T. Dierks and S. Jagannathan, "Output feedback control of a quadrotor uav using neural networks," *IEEE transactions on neural networks*, vol. 21, no. 1, pp. 50–66, 2010.

[4] J. A. Bagnell and J. G. Schneider, "Autonomous helicopter control using reinforcement learning policy search methods," in *Robotics and Automation, 2001. Proceedings 2001 ICRA. IEEE International Conference on*, vol. 2. IEEE, 2001, pp. 1615–1620.

[5] H. J. Kim, M. I. Jordan, S. Sastry, and A. Y. Ng, "Autonomous helicopter flight via reinforcement learning," in *Advances in neural information processing systems*, 2004, pp. 799–806.

[6] P. Abbeel, A. Coates, M. Quigley, and A. Y. Ng, "An application of reinforcement learning to aerobatic helicopter flight," in *Advances in neural information processing systems*, 2007, pp. 1–8.

[7] J. Hwangbo, I. Sa, R. Siegwart, and M. Hutter, "Control of a quadrotor with reinforcement learning," *IEEE Robotics and Automation Letters*, vol. 2, no. 4, pp. 2096–2103, 2017.

[8] S. R. B. dos Santos, S. N. Givigi, and C. L. N. Júnior, "An experimental validation of reinforcement learning applied to the position control of uavs," in *2012 IEEE International Conference on Systems, Man, and Cybernetics (SMC)*. IEEE, 2012, pp. 2796–2802.

[9] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[10] W. Koch, R. Mancuso, R. West, and A. Bestavros, "Reinforcement learning for uav attitude control," 2018.

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

[12] D. J. Leith and W. E. Leithead, "Survey of gain-scheduling analysis and design," *International journal of control*, vol. 73, no. 11, pp. 1001–1025, 2000.

[13] N. Hovakimyan, C. Cao, E. Kharisov, E. Xargay, and I. M. Gregory, "L1 adaptive control for safety-critical systems," *IEEE Control Systems*, vol. 31, no. 5, pp. 54–104, 2011.

[14] D. Lee, H. J. Kim, and S. Sastry, "Feedback linearization vs. adaptive sliding mode control for a quadrotor helicopter," *International Journal of control, Automation and systems*, vol. 7, no. 3, pp. 419–428, 2009.

[15] N. Koenig and A. Howard, "Design and use paradigms for gazebo, an open-source multi-robot simulator," in *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, vol. 3. IEEE, 2004, pp. 2149–2154.

[16] "BetaFlight," https://github.com/betaflight/betaflight, 2018, accessed: 2018-11-25.

[17] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, "Openai baselines," https://github.com/openai/baselines, 2017.

[18] "Tensorflow:An Open Source Machine Learning Framework for Everyone," https://github.com/tensorflow/tensorflow/, 2018, accessed: 2018-11-25.

[19] A. Hill, A. Raffin, R. Traore, P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, and Y. Wu, "Stable baselines," https://github.com/hill-a/stable-baselines, 2018.

[20] M. Schaarschmidt, A. Kuhnle, and K. Fricke, "Tensorforce: A tensorflow library for applied reinforcement learning," Web page, 2017. [Online]. Available: https://github.com/reinforceio/tensorforce

[21] "STM32F745VG," https://www.st.com/en/microcontrollers/stm32f745vg.html, 2018, accessed: 2018-11-25.

[22] "Bazel - a fast, scalable, multi-language and extensible build system," https://bazel.build/, 2018, accessed: 2018-11-25.

[23] "NF1: Neuroflight Test Aircraft 1," https://rotorbuilds.com/build/15163, 2018, accessed: 2018-11-25.

[24] "Motor Data Explorer," https://www.miniquadtestbench.com/motor-explorer.html, 2018, accessed: 2018-11-25.

[25] "Iris Quadcopter," 2018. [Online]. Available: http://www.arducopter.co.uk/iris-quadcopter-uav.html

[26] "Brushless motor constant explained," http://learningrc.com/motor-kv/, 2015.

[27] O. Liang, "Looptime and Flight Controller," https://oscarliang.com/best-looptime-flight-controller/, 2018, accessed: 2018-11-25.