Enabling Intelligent Onboard Guidance, Navigation, and Control Using Near-Term Flight Hardware.

Callum Wilson[1, a]* **and Annalisa Riccardi**[1, b]

[1] *Department of Mechanical and Aerospace Engineering, University of Strathclyde, 75 Montrose Street, Glasgow*

G1 1XJ, United Kingdom.
[a] *callum.j.wilson@strath.ac.uk.*
[b] *annalisa.riccardi@strath.ac.uk.*
* *Corresponding author*

Abstract

Future space missions will require various technological advancements to meet more stringent and challenging requirements. Next generation guidance, navigation, and control systems must safely operate autonomously in hazardous and uncertain environments. While the focus of these developments is often on flight software, spacecraft hardware also creates computational limitations for the algorithms which run onboard. Here we look at the feasibility of implementing intelligent control onboard a spacecraft. Intelligent control methods combine theories from automatic control, artificial intelligence, and operations research to derive control systems capable of handling substantial uncertainties. This has clear benefits for operating in unknown space environments. However, most modern intelligent systems require substantial computational power which is not available onboard spacecraft. Recent advancements in single board computers have created much physically lighter and less power-intensive processors which are suitable for spaceflight and purpose built for machine learning. In this study we implement a reinforcement learning based controller on NVIDIA Jetson Nano hardware. We apply this controller to a powered descent guidance problem using a simulated Mars landing environment. The initial offline approach used to derive the controller has two steps. First, optimal trajectories and guidance laws are calculated using nominal environment conditions. These are then used to initialise a reinforcement learning agent which learns a control policy that copes with environmental uncertainties. In this case the control policy is parameterised as a neural network which can also update its weights online from real time observations. Online updates use a novel method of weight updates called Extreme Q-Learning Machine to tune the output weights of the neural network in operation. We show that this control system can be deployed on hardware which is sufficiently light, in terms of power and mass, to be used onboard spacecraft. This demonstrates the potential for intelligent controllers to be used on flight suitable hardware.

Keywords: Intelligent Control, Reinforcement Learning, Edge Artificial Intelligence

1 Introduction

Designing effective spacecraft Guidance Navigation and Control (GNC) systems poses several challenges. These control systems must deal with substantial uncertainties inherent to space missions which is difficult for many conventional methods to handle. One class of methods which can be used for controlling uncertain environments is Intelligent Control (IC). These methods use theories and architectures from the field of Artificial Intelligence (AI), combined with automatic control and operations research, to devise autonomous control systems which handle uncertainties in a system's environment, goals, or even in the control system itself[1]. This has clear benefits for spacecraft GNC, however, one factor which limits the use of IC onboard spacecraft is its computational cost. Computing systems onboard spacecraft face stringent power budgets, while many AI architectures are computationally intensive.

New advances in edge hardware for AI have increased the possibility for spacecraft onboard intelligence. Several manufacturers now produce high performance Graphics Processing Units (GPUs) optimised for running AI algorithms which are computationally and physically lightweight. For example, companies such as NVIDIA [2] and Intel [3] are investing substantial resources in developing edge AI capabilities. Due to the availability of this hardware, applications of onboard intelligence are now growing, with most being related to image processing. One recent example uses flight tested hardware to create segmentation maps of floods with the goal of spacecraft being able to downlink these maps in real time to enable quicker disaster response[4]. Applications of onboard intelligence to GNC are more limited and so here we aim to use onboard intelligence to directly train a controller.

Reinforcement Learning (RL) is a class of AI methods which has gained interest for many different applications. These methods learn how to control a system through interaction and optimise their behaviour with respect to a reward function specified by the designer[5]. Recent breakthroughs in AI applied to games have used RL methods, which demonstrates their capabilities on a variety of difficult problems. Most notably, work by DeepMind produced AlphaGo and AlphaZero which overcame the task - previously thought impossible - of beating the world's best humans at the game of Go[6]. In most cases, a RL agent learns how to control a system through repeated, simulated interactions and then does not update its control policy in operation. While this approach is still effective, it cannot be considered 'intelligent' control in the sense that it does not update online based on new information. Our work aims to demonstrate the possibility for including online updates with an RL agent.

We apply RL to a three degree-of-freedom Martian spacecraft powered descent problem with environmental uncertainties. Spacecraft landing on extra-terrestrial bodies is an important area of research as further missions to the Moon and Mars are planned [7]. These will have stricter performance requirements than previous missions which motivates the use of novel technologies such as IC. Previous work applying RL to this problem includes Gaudet et al [8] which uses the RL method Proximal Policy Optimisation (PPO)[9] to learn over long training periods. This work builds on previous work that improved the training time for this problem using discrete action spaces [10] and by incorporating optimal control demonstrations[11].

The primary contribution of this work is the demonstration of online updates of a pretrained RL agent performed on near-term flight hardware. Online updates use a novel update mechanism called Extreme Q-Learning Machine (EQLM)[12]. This uses theories from Extreme Learning Machines (ELMs)[13] to update Neural Network (NN) parameters without using conventional gradient-based methods. EQLM is capable of training a neural network without the need of iterative tuning making it suitable for an online learning environment. To demonstrate the feasibility of using this approach onboard spacecraft, we run our agent on the NVIDIA Jetson Nano 2GB developer kit [14]. This small, single board computer incorporates a NVIDIA Maxwell GPU that is designed for edge AI applications. The NVIDIA Jetson system has existing flight heritage onboard a recent Lockheed mission that demonstrated various onboard AI applications[15]. With this work, we aim to show that intelligent control is feasible onboard near-term flight hardware for spacecraft GNC.

2 Methods

The approach we use here can be divided into three main parts. The first two parts occur offline for initialising the agent and the final part occurs online. First an agent is trained using conventional gradient descent methods to determine initial weights for the Q-network. We use the conventional gradient-based updates for this part instead of EQLM since the Q-networks have multiple layers, but ELM updates only single layers. The Q-network weights learned from this part are then used to initialise offline the matrix used for EQLM updates. Finally, the agent updates its output weights online. Here we describe this methodology with a brief introduction to the underlying theories of Q-learning.

2.1 Q-Networks

RL processes consist of an agent which takes actions, a on an environment and observes states, s and rewards, r. The agent's goal is to maximise its cumulative reward. Various approaches exist to solving this problem [5] and our method is based on a popular algorithm called Q-learning [16]. This aims to learn an optimal policy by learning action-values, denoted Q - that is the expected cumulative reward following an action in a given state. Building on this, more recently the Deep Q-Networks (DQN) algorithm became popular for its performance in the Atari benchmark environments [17]. This algorithm parameterises the action-value estimates using a NN which allows it to scale to large problems.

The DQN algorithm has several key features. First, it uses experience replay [18], where it stores its observations in a replay memory and samples from this when performing network updates. While training, DQN uses an ϵ-greedy policy to explore states by occasionally taking random actions. The parameter ϵ defines the probability of taking a random action and this value is decreased as the agent learns more with less need to explore. Finally it makes use of a target network, which is a second NN that has the same structure as the main Q-network. This network's parameters remain fixed for a period of time before taking the values of the main network's parameters. The target network is used for calculating targets for updating with the aim of decoupling the action-value updates and estimation.

The main network's parameters are denoted with θ and the target network parameters θ^T. Updates use experiences sample from the replay memory, which consist of a state, action, observed state, and observed reward, denoted $(s_j, a_j, r_{j+1}, s_{j+1})$. The target action value $\mathbf{t}_j$ is then calculated as shown:

$$\mathbf{t}_j = \begin{cases} r_j, & \text{if } s_{j+1} \text{ is terminal} \\ r_j + \gamma \max_a Q_{\theta^T}(s_{j+1}, a), & \text{otherwise} \end{cases} \tag{1}$$

where γ is the discount factor that controls the extent to which long term rewards are considered ($0 \leq \gamma < 1$). The temporal-difference error, $\mathbf{e}_j$ can then simply be calculated as the difference between the estimated action value and target action value:

$$\mathbf{e}_j = Q_\theta(s_j, a_j) - \mathbf{t}_j \tag{2}$$

The temporal difference error can then be used to update the parameters of the main network via gradient descent. This is the approach used in the initial part of our method. The remainder of this section describes our method's three main parts.

2.2 Offline Pretraining

In this part of the method we train an agent using DQN with optimal control demonstrations. For a more detailed description of the approach used for initialisation, readers are referred to the authors previous work [11].

The agent is initially trained as shown schematically in Figure 1 using demonstrations derived by optimisation procedures. These optimal control demonstrations from various initial conditions are stored in a separate demonstration replay memory. Experiences from both this replay memory and the agent's own replay memory are used to update the agent. The output of this part is a pretrained Q-network with optimised parameters.

For the problem scale we consider here, a multilayer Q-Network is required. EQLM is used to update single layer networks with random initial weights and biases. Training a Q-Network initially using gradient descent allows EQLM updates to be performed on just the output layer and instead of random initial weights, the other network parameters are tuned to give a useful representation to the final layer. Following pretraining, all the Q-Network parameters are fixed except for the output weights which will be updated online.

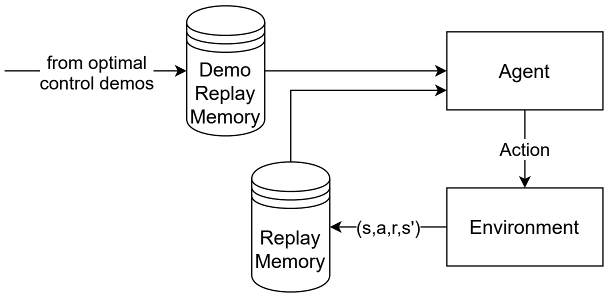

Figure 1: Schematic illustration of agent pretraining using optimal control demonstrations.

2.3 Offline EQLM Initialisation

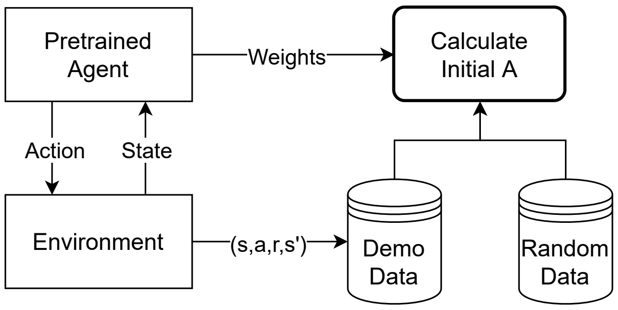

Figure 2: Schematic illustration of initialising matrix for EQLM updates.

Prior to applying EQLM online, it is first necessary to initialise variables for performing updates. Here we describe the process for calculating initial values for the matrix $A^\dagger$ which is used for performing online updates with EQLM. This process is illustrated in Figure 2. The input to the network is the environment state, denoted $\mathbf{s}$. The output of the ith hidden layer, denoted $\mathbf{y}_i$, is:

$$\mathbf{y}_i = (f_i \circ f_{i-1} \circ \cdots \circ f_1)(\mathbf{s}) \tag{3}$$

where $f_i()$ is a nonlinear function of the previous layer's output resulting in each output being a composite function of all previous layers. For an initial minibatch of k_0 inputs $(\mathbf{s}_1, \ldots, \mathbf{s}_{k_0})$, we define the matrix $\mathbf{H}$ as the output of the final hidden layer (denoted $\mathbf{y}$ for simplicity) for each of the inputs as shown:

$$\mathbf{H} = \begin{bmatrix} \mathbf{y}|_{\mathbf{s}=\mathbf{s}_1}^T \\ \vdots \\ \mathbf{y}|_{\mathbf{s}=\mathbf{s}_{k_0}}^T \end{bmatrix}_{k_0 \times \tilde{N}} \tag{4}$$

where $\tilde{N}$ denotes the number of nodes in the final hidden layer. Each row of $\mathbf{H}$ contains the final hidden layer output $\mathbf{y}$ for each of the inputs $(\mathbf{s}_1, \ldots, \mathbf{s}_{k_0})$. The output weights of the network are denoted with β. Matrix $\mathbf{T}$ is then defined as the corresponding target action-values for the inputs $(\mathbf{s}_1, \ldots, \mathbf{s}_{k_0})$ as shown in equation 5. These are calculated using equation 1.

$$\mathbf{T} = \begin{bmatrix} \mathbf{t}|_{\mathbf{s}=\mathbf{s}_1}^T \\ \vdots \\ \mathbf{t}|_{\mathbf{s}=\mathbf{s}_{k_0}}^T \end{bmatrix}_{k_0 \times m} \tag{5}$$

where m denotes the number of discrete actions. For a set of output weights which perfectly represents the targets $\mathbf{T}$ for a given $\mathbf{H}$, the Q-network model can be written as shown in equation 6:

$$\mathbf{H}\beta = \mathbf{T} \tag{6}$$

As stated previously, following pretraining only the output weights β are updated. In the case where EQLM is used without pretraining, network parameters are randomly assigned and the output weights are initialised as shown:

$$A_{t=0}^{\dagger} = \left[\frac{I}{\bar{\gamma}} + \mathbf{H}^T\mathbf{H}\right]^{\dagger} \tag{7}$$

$$\beta_{t=0} = A_{t=0}^{\dagger}\mathbf{H}^T\mathbf{T} \tag{8}$$

However, following pretraining we can instead use the optimised set of output weights to initialise $A_{t=0}^{\dagger}$ by rearranging equation 8 as shown:

$$A_{t=0}^{\dagger} = \beta_{t=0}\left[\mathbf{H}^T\mathbf{T}\right]^{\dagger} \tag{9}$$

where $\beta_{t=0}$ is the optimised output weights. $\mathbf{H}$ and $\mathbf{T}$ are both calculated for a large batch of experiences to perform this step. This initialisation is the final part of the process which occurs offline.

2.4 Online updating

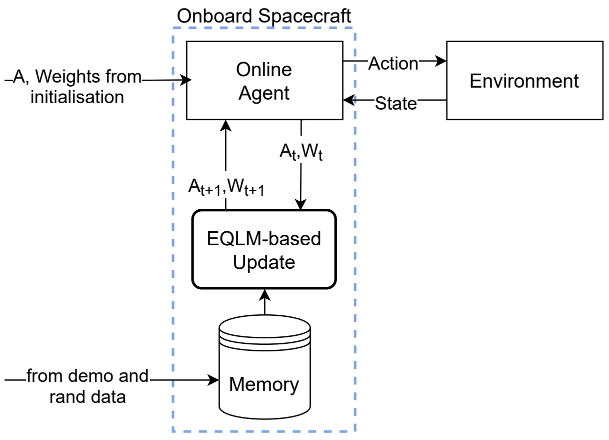

Figure 3: Schematic illustration of an onboard agent with online updates.

With the weights and matrix for updating weights initialised, the agent is now able to update its weights online. In practise, this part would occur onboard the spacecraft. This is highlighted in Figure 3. As with the initialisation part, the online updates use data from agent demonstrations and random actions. In addition to these replay memories, the experiences observed online by the agent also make up the data used to update. These experiences are all fed to the agent in minibatches, whose size is again denoted k. If the agent has already updated from N experiences and samples a new batch of k experiences $(\mathbf{s}_N, \ldots, \mathbf{s}_{N+k})$ for updating, the new incremental matrices $\mathbf{H}_{IC}$ and $\mathbf{T}_{IC}$ can be defined as follows:

$$\mathbf{H}_{IC} = \begin{bmatrix} f(\mathbf{y}|_{\mathbf{s}=\mathbf{s}_N}) \\ \vdots \\ f(\mathbf{y}|_{\mathbf{s}=\mathbf{s}_{N+k}}) \end{bmatrix}_{k \times \tilde{N}} \tag{10}$$

$$\mathbf{T}_{IC} = \begin{bmatrix} \mathbf{t}|_{\mathbf{s}=\mathbf{s}_N}^T \\ \vdots \\ \mathbf{t}|_{\mathbf{s}=\mathbf{s}_{N+k}}^T \end{bmatrix}_{k \times m} \tag{11}$$

Using these matrices, the online EQLM updates are performed with the following equations:

$$K_t = I - A_t^{\dagger}\mathbf{H}_{IC}^T\left(\mathbf{H}_{IC}A_t^{\dagger}\mathbf{H}_{IC}^T + I_{k\times k}\right)^{\dagger}\mathbf{H}_{IC} \tag{12}$$

$$\beta_{t+1} = K_t\beta_t + K_tA_t^{\dagger}\mathbf{H}_{IC}^T\mathbf{T}_{IC} \tag{13}$$

$$A_{t+1}^{\dagger} = K_tA_t^{\dagger} \tag{14}$$

where K_t is another matrix used to calculate weight updates. These updates occur after a certain number of timesteps, n_{step} in the environment. At each update, the agent's previous n_{step} experiences are all used to update, along with $k - n_{step}$ points sampled from the demonstration and random action replay memories.

3 Application to Powered Descent

This section details how the method described previously is applied to the spacecraft lander powered descent problem. Here we introduce the mathematical models and parameters describing the environment and detail the experiments carried out to test this application.

3.1 Environment

The environment we consider is a three degree-of-freedom powered descent for Mars. For this case with no rotations, the spacecraft is considered to have thrusters aligned with its body-frame axes. The equations of motion of the spacecraft are shown in equations 15 to 17:

$$\frac{d}{dt}(\mathbf{x}) = \dot{\mathbf{x}} \tag{15}$$

$$\frac{d}{dt}(\dot{\mathbf{x}}) = \frac{\mathbf{F}_{thrust} + \mathbf{F}_{env}}{m} + \mathbf{g} \tag{16}$$

$$\frac{d}{dt}(m) = -\frac{\Sigma \text{abs}\left(\mathbf{F}_{thrust}\right)}{I_{sp} \cdot g_0} \tag{17}$$

where $\mathbf{x} = \{x_i, x_j, x_k\}\ m$ is the spacecraft's position w.r.t the desired landing location, $\mathbf{F}_{thrust} = \{F_i, F_j, F_k\}\ N$ is the force exerted by the thrusters on the spacecraft, $\mathbf{F}_{env}$ is the disturbance forces from the environment, m is the spacecraft total mass, $\mathbf{g} = \{0, 0, -3.72\}\ N/kg$ is the acceleration due to gravity for Mars, $I_{sp} = 210s$ is the specific thrust of each thruster, and $g_0 = 9.81N/kg$ is the reference acceleration due to gravity.

Possible thrusts are discretised within the ranges $-F_i^{max} \leq F_i \leq F_i^{max}$, $-F_j^{max} \leq F_j \leq F_j^{max}$, and $0 \leq F_k \leq F_k^{max}$. Maximum thrust magnitudes are $F_i^{max} = F_j^{max} = 10kN$ and $F_k^{max} = 13kN$. The number of discrete actions in each direction is 7 in both the i- and j-directions and 3 in the k-direction, giving an action space size of 147 discrete actions. Environmental forces are randomly sampled every 5 timesteps from a normal distribution with mean $0N$ and standard deviation $100N$ and linearly interpolated between samples. The control system's sampling time is $0.2s$.

Appropriate definition of the reward function is crucial to ensure the agent learns a desirable policy. This problem benefits from a shaped reward function which effectively guides the agent towards more optimal actions since it is otherwise difficult for the agent to learn to find the landing site. The reward function we use here is the same as from previous work [11] which is adapated from the reward function used by Gaudet et al [8]. This function is shown in equation 18:

$$\begin{aligned} r =& \alpha \left\|\dot{\mathbf{x}} - \dot{\mathbf{x}}_{targ}\right\| + \beta \left\| \frac{\mathbf{F}_{thrust}}{\mathbf{F}_{thrust}^{max}} \right\| + \eta \\ &+ \kappa \left(r_z < 0.01\ and\ \|\mathbf{x}\| < x_{lim}\ and\ \|\dot{\mathbf{x}}\| < \dot{x}_{lim}\right) \end{aligned} \tag{18}$$

$\dot{\mathbf{x}}_{targ}$ is a target velocity that the agent is rewarded for following. The shape of this target velocity is defined by equations 19 to 24:

$$\dot{\mathbf{x}}_{targ} = -v_0 \left(\frac{\hat{\mathbf{r}}}{\|\hat{\mathbf{r}}\|}\right) \left(1 - exp\left(-\frac{t_{go}}{\tau}\right)\right) \tag{19}$$

$$v_0 = 70 \tag{20}$$

$$t_{go} = \frac{\|\hat{\mathbf{r}}\|}{\|\hat{\mathbf{v}}\|} \tag{21}$$

$$\hat{\mathbf{r}} = \begin{cases} \mathbf{x} - [0\ 0\ 15], & \text{if } x_3 > 15 \\ [0\ 0\ x_3], & \text{otherwise} \end{cases} \tag{22}$$

$$\hat{\mathbf{v}} = \begin{cases} \dot{\mathbf{x}} - [0\ 0\ -2], & \text{if } x_3 > 15 \\ \dot{\mathbf{x}} - [0\ 0\ -1], & \text{otherwise} \end{cases} \tag{23}$$

$$\tau = \begin{cases} 20, & \text{if } x_3 > 15 \\ 100, & \text{otherwise} \end{cases} \tag{24}$$

This motivates the agent to follow a velocity pointing towards $15m$ above the desired landing zone which decreases as it approaches this point. Then over the final $15m$ of descent the target velocity is entirely normal to the surface in the k-direction. The constants α, β, η, and κ in equation 18 weight different parts of the reward function. The α term is a negative reward which increases as the difference between the spacecraft's velocity and target velocity increases. The β term is also a negative reward for the control effort normalised w.r.t the maximum thrust. η is a positive constant which motivates the agent to keep advancing in the environment. Finally, κ is the reward gained for a successful landing within the limits of $x_{lim} = 5m$ and $\dot{x}_{lim} = 2m/s$. The values of the constant coefficients in the reward function are $\alpha = -0.01$, $\beta = -0.05$, $\eta = 0.01$, and $\kappa = 10$.

The input to the network $\mathbf{s}$ consists of the lander's position, velocity, and mass. To avoid saturating the hidden layer outputs in the network, inputs are scaled by a constant factor $\mathbf{s}_{scale}$ as shown:

$$\mathbf{s} = \{\mathbf{x},\ \dot{\mathbf{x}},\ m\} \cdot \mathbf{s}_{scale} \tag{25}$$

with the values for scaling each variable defined as follows:

$$\mathbf{s}_{scale} = \{0.01,\ 0.01,\ 0.01,\ 0.1,\ 0.1,\ 0.1,\ 0.005\} \tag{26}$$

3.2 Pretraining and Initialisation

Using the process detailed in [11], we first pretrain an agent using gradient descent methods and optimal demonstrations for 10,000 episodes. Due to the stochastic nature of the algorithm and environment, each training run yields variable performance and so several runs are performed with different random seeds. We perform 32 training runs in total to get 32 sets of pretrained agent weights. All of these agents are then initialised 5 times, again with different random seeds, and run for a fixed number of EQLM updates to test their performance after updating. The pretrained agent used for online updating is selected based on its performance across the 5 runs of these episodes. For all the experiments shown here, initial conditions are selected randomly at the start of each episode in the ranges shown in Table 1.

Table 1: Range of initial conditions for training and testing agents

State Variable	Min.	Max.
i-position	$0.4km$	$1.1km$
j-position	$-1.1km$	$1.1km$
k-position	$2.4km$	$2.6km$
i-velocity	$-75ms^{-1}$	$-5ms^{-1}$
j-velocity	$-35ms^{-1}$	$35ms^{-1}$
k-velocity	$-95ms^{-1}$	$-65ms^{-1}$
mass	$2000kg$	$2000kg$

The hyperparameters used for pretraining are shown in Table 2. The network has three hidden layers with $tanh$ activation functions and is updated using the RMSProp al-gorithm. ϵ_0 and n_ϵ are parameters which control the rate of exploration of the agent - how often it takes random actions to explore the environment. γ is the discount factor, k is the number of experiences sampled from the agent's own replay memory for each update and k_{demo} is the number of experiences sampled from the optimal control demonstrations, giving a total minibatch size of $k + k_{demo} = 200$. n_T is the number of timesteps between target network updates.

Table 2: Hyperparameters for agent pretraining (from [11])

Parameter	Value
Hidden units	(300,200,300)
learning rate	1×10^{-5}
ϵ_0	0.6
n_ϵ	4000
γ	0.926
k	100
k_{demo}	100
n_T	65

Following pretraining, the next step is to initialise the agent for online EQLM updates. As with the pretraining run, initialising $A^\dagger_{t=0}$ involves a stochastic process of sampling data from the demonstration and random replay memories. Therefore, we also perform this initialisation 32 times with different random seeds and assess the performance as before. The initial minibatch size k_0 used to initialise $A^\dagger_{t=0}$ is fixed at 2000 for all experiments with 1000 experiences sampled from the agent demonstrations and 1000 from random actions. With the network architecture defined in pretraining, the only other hyperparameter for EQLM updates is the discount factor γ. This is fixed at $\gamma = 0.9$ in all cases - both for initialisation and online updates.

While the initial minibatch size could be fixed, the minibatch size for online updating has a greater effect on the agent's performance. Moreover, larger minibatch sizes would increase the time taken to update weights which is undesirable for online updating. This motivates the careful tuning of this minibatch size to minimize it while achieving the best performance. As well as the overall minibatch size, the number of steps per update must also be chosen. This affects not only the frequency of updates but also the number of observed experiences included in each update. To tune these parameters, we consider three performance criteria to minimise: norm of final position in the i- and j-directions, norm of final velocity in the i- and j-directions, and absolute final velocity in the k-direction. For each pair of parameters (minibatch size and steps per update), we run 200 episodes and calculate the performance in terms of those three criteria. From this, we can create a Pareto front of non-dominated solutions and select the set of parameters which appears most often in those solutions. The range of values tested for the number of steps per update is $2, 4, 8, 16, 32, 64, 128$ and for minibatch size is $40, 80, 120, 160, 200$, excluding pairs of parameters where the steps per update is greater than the minibatch size.

With the pretrained weights determined, initial $A^{\dagger}$ matrix calculated, and steps per update and minibatch size selected it possible to run the online EQLM updates. First we add an additional training step of running EQLM updates offline for 500 episodes with the goal of better tuning the output weights and matrix $A^{\dagger}$ prior to running online. When running online updates, each episode begins with the same set of output weights and matrix $A^{\dagger}$, whereas for the offline updates these are updated over all the episodes. We use the same values for steps per update and minibatch size as for the online updates in the offline update runs.

3.3 Experiments on hardware

The hardware used for testing online updates is the NVIDIA Jetson Nano 2GB developer kit. It is equipped with a 64-bit Quad-core ARM A57 CPU at 1.43 GHz and a 128-core NVIDIA Maxwell GPU. The onboard memory for the developer kit is 2 GB 64-bit LPDDR4. The methods detailed previously are implemented in the Tensorflow Python library which makes it possible to use GPU acceleration. Testing episodes that are run on the hardware use the same initial set of weights and matrix $A^{\dagger}$ at the start of the episode, which are then updated throughout the episode. Initial conditions for the episodes are the same as in training as shown in Table 1.

4 Results and Discussion

Results are shown for the offline pretraining and initialisation and for the online updates run on the Jetson Nano hardware. For the cases where multiple seeded runs were carried out to determine best seeds, for brevity we only show results of the best performing run.

4.1 Pretraining and Initialisation

Figure 4 shows the learning curve of the best pretrained agent out of 32 runs. Cumulative reward refers to the sum of all rewards over a single episode. These are

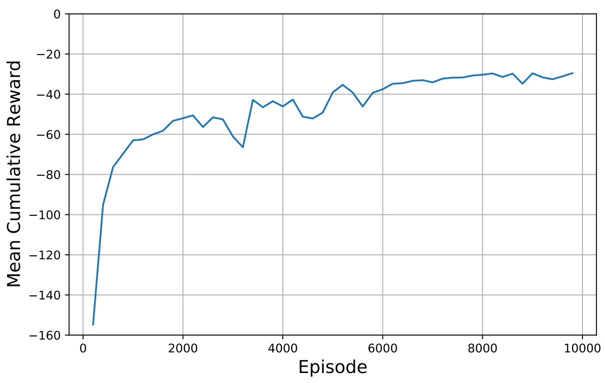

Figure 4: Learning curve for the best performing pretrained agent. Rewards displayed are averaged over 200 episodes.

then averaged across episodes to more clearly show the learning curve. The mean cumulative reward increases most rapidly over the first 1000 episodes before increasing more gradually over the rest of the learning period. Over the last 100 episodes it is able to successfully land frequently with a mean cumulative reward of -28.5.

As stated previously, pretrained agents were evaluated based on performance over 500 episodes with EQLM update and each pretrained agent ran using 5 different random seeds for initialising the $A^{\dagger}$ matrix. Performance for each of these seeds for the best pretrained agent are summarised in Table 3. This table also shows a summary of the performance from the best performing of the 32 $A^{\dagger}$ matrix initialisation seeds, which all used the best pretrained agent.

These results highlight the importance of selecting a random seed for initialising $A^{\dagger}$ since the performance of the best pretrained agent varies considerably across seeds. This is shown most clearly in the k-velocity, which has median and standard deviation of -0.93 and 0.61 for seed 4, but these are -1.79 and 6.96 for seed 3. The high standard deviation in k-velocity in seed 3 shows the agent is frequently crashing the spacecraft at high velocity, which is obviously very undesirable. Contrast this with the performance of the best $A^{\dagger}$ seed, which has a median and standard deviation of only -0.88 and 0.54. It is interesting to note that median fuel consumption is broadly similar across all seeds, which shows all seeds perform similarly with regards to fuel economy but vary more substantially in how often they successfully land.

Table 3: Median and standard deviation of values for terminal states and fuel consumption from 500 online update episodes with the best random seeds for initialisation. Results for the best pretrained agent are shown for each of its 5 runs.

	Norm ij-position (m)		Norm ij-velocity (m/s)		k-velocity (m/s)		Fuel (kg)	
	median	SD	median	SD	median	SD	median	SD
Pretrained - seed 1	9.35	7.00	1.08	1.05	-0.92	5.05	439.1	27.4
Pretrained - seed 2	9.58	6.82	0.90	1.51	-1.05	1.03	436.1	39.5
Pretrained - seed 3	14.59	12.12	1.83	3.92	-1.79	6.96	432.8	62.4
Pretrained - seed 4	11.63	6.88	0.95	0.62	-0.93	0.61	438.1	33.6
Pretrained - seed 5	9.12	5.45	1.01	1.00	-0.74	0.65	438.3	28.9
Best $A^\dagger$ seed	7.35	5.89	1.03	0.59	-0.88	0.54	438.9	26.2

Table 4: Comparison of two best performing hyperparameter values in terms of terminal state values.

	(80, 4)		(160, 16)	
	median	SD	median	SD
ij-position (m)	7.77	5.74	8.02	7.37
ij-velocity (m/s)	0.95	0.63	1.08	1.80
k-velocity (m/s)	0.86	0.58	0.88	0.96

4.2 Hyperparameter Selection

For the ranges of number of steps per update and minibatch size stated previously, we found the non-dominated solutions in terms of norm ij-position, norm ij-velocity, and absolute k-velocity for all of the 6200 episodes across the 31 pairs of hyperparameters. This gave 23 Pareto-optimal points. Of these, two pairs of hyperparameters produced 4 episodes each making them the most frequently optimal. These were (minibatch 80, update steps 4) and (minibatch 160, update steps 16). To select one of these sets of hyperparameters, we compared their performance across all of their episodes. These are summarised in Table 4. Since the pair (80,4) shows lower median and standard deviations across all performance criteria, these were selected as the hyperparameters to use.

These hyperparameters are used for all further EQLM updates. The number of steps per update is 4, meaning the agent updates its weights every 4 timesteps and uses the previous 4 experiences. For the minibatch size of 80, this means that the agent also samples 38 experiences from the agent demonstrations and 38 experiences from the random actions for each update.

4.3 Online Updates

The reaminder of the results come from experiments run on the Jetson Nano hardware. Three different agents were run on the hardware to test their performance:

- *Pretrained agent* - using the fixed weights of the best pretrained agent without any online updating.
- *Offline updated agent* - using the best pretrained agent and best $A^\dagger$ plus 500 episodes of offline updates without any online updating.
- *Online updated agent* - using the best pretrained agent and best $A^\dagger$ plus 500 episodes of offline updates and with online updating.

The distributions of final states for each of these agents are shown in Figures 5, 6, and 7. For both Figure 5 showing positions and Figure 6 showing velocities, the majority of points are clustered near the origin as desired. The distribution of points for the pretrained and online updated agents are both very similar, while the offline updated agent has a more off-centre distribution. This suggests that the online updates stear the performance more towards that of the pretrained agent following offline updates. For all agents the majority of terminal k-velocities have a magnitude less than $2m/s$, however some episodes show higher landing speeds of more than $3m/s$. Similarly, the terminal i- and j-velocities show several outliers - most notably in the offline updated agent. This shows further improvements to the robustness of these agents to the varying initial conditions and disturbances are necessary. Nevertheless, in most cases each agent is able to successfully land close to the desired location.

Table 5 summarises each agent's performance over the 500 episodes. In most cases each agent performs sim-

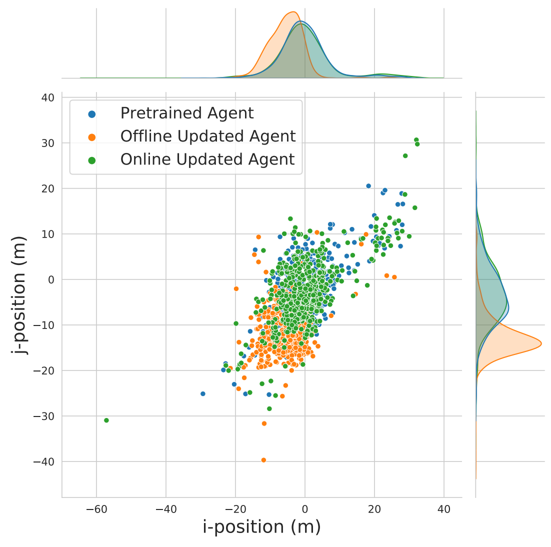

Figure 5: Distributions of terminal i- and j-positions for each of the agents over 500 episodes.

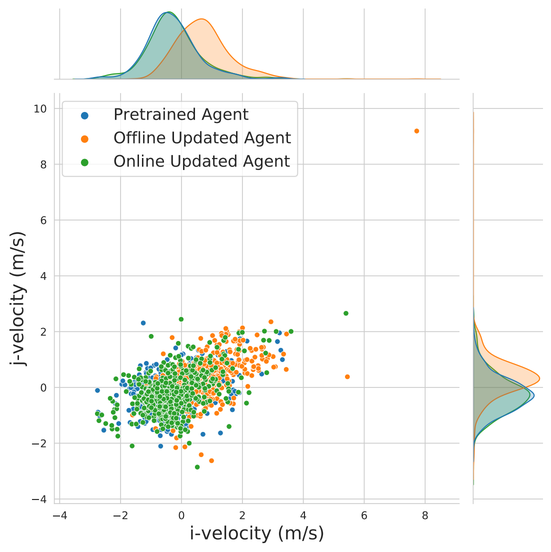

Figure 6: Distributions of terminal i- and j-velocities for each of the agents over 500 episodes.

ilarly, however the offline u pdated a gent h as a notably greater variability in the ij-velocity. This agent also shows the highest median fuel consumption, whereas the difference between the pretrained and online updated agents' fuel consumption is only 2kg.

Figures 8 to 10 show one example trajectory obtained using the online updating agent. The velocities over the trajectory in Figure 9 demonstrate the effect of the target velocity on the agent's policy as the spacecraft initially accelerates quickly towards its target velocity before the velocity decreases towards zero. Figure 10 also clearly shows the discrete thrust magnitudes in each direction. The initial conditions for this trajectory were $\mathbf{x}_0 = \{0.53, -0.89, 2.45\}\,km$, $\dot{\mathbf{x}}_0 = \{-15.6, -30.7, -76.2\}\,m/s$, and $m_0 = 2000kg$. From this state, the agent reaches a terminal state of $\mathbf{x} = \{-0.089, -9.011, 0.024\}\,m$ and $\dot{\mathbf{x}} = \{-0.71, -1.12, -0.15\}\,m/s$ with a fuel consumption of $455.5kg$.

The final results we show are the time taken to update the agent when running on the Jetson Nano. This is important to review since the updates cannot be so slow to prevent their use online using real hardware. Figure 11 shows a histogram of times taken to update the agent over 500 update steps. The time taken to update includes adding the most recent observation to the experience replay, sampling from all experience replays, and calculating and assigning the new weights. Most updates last between 0.05 and 0.10s which is less than the control system sampling

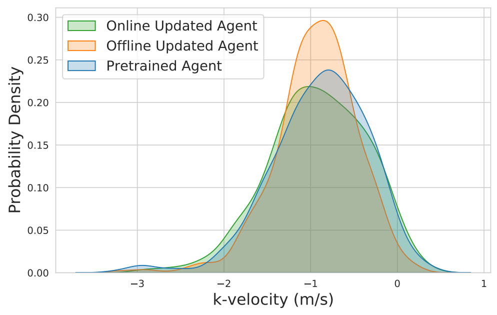

Figure 7: Distributions of terminal k-velocities for each of the agents over 500 episodes.

time of 0.2s as desired. However, there are 2 occasions where updating takes longer than the sampling time with the longest lasting 0.276s. This still allows updates to be made during intervals of 4 timesteps as in our case. The code used to get these timings was also not optimised for time and so these times could certainly be improved.

5 Conclusions

This work demonstrates the potential for intelligent control methods to be implemented on flight-suitable hardware for onboard spacecraft control. Using a reinforcement learning based approach, we were able to train an agent to solve a powered descent problem subject to environmental uncertainties and perform weight updates on-

Table 5: Median and standard deviation of values for terminal states and fuel consumption from 500 episodes run on Jetson Nano hardware for three different configurations of agent.

	Norm ij-position (m)		Norm ij-velocity (m/s)		k-velocity (m/s)		Fuel (kg)	
	median	SD	median	SD	median	SD	median	SD
Pretrained	7.39	6.01	0.92	0.58	-0.84	0.56	437.2	21.5
Offline updated	14.85	4.03	0.91	0.91	-0.90	0.47	449.7	23.1
Online updated	7.21	7.37	0.92	0.66	-0.91	0.56	435.7	22.8

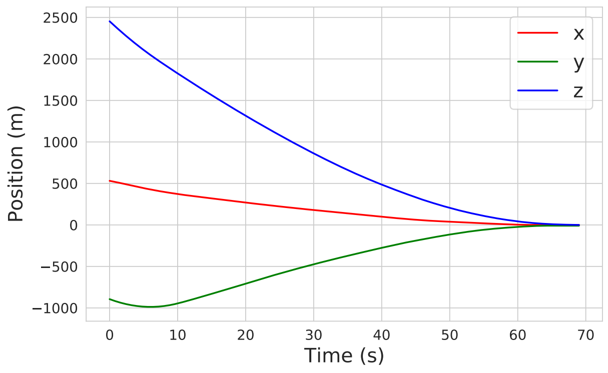

Figure 8: Spacecraft positions over a sample trajectory.

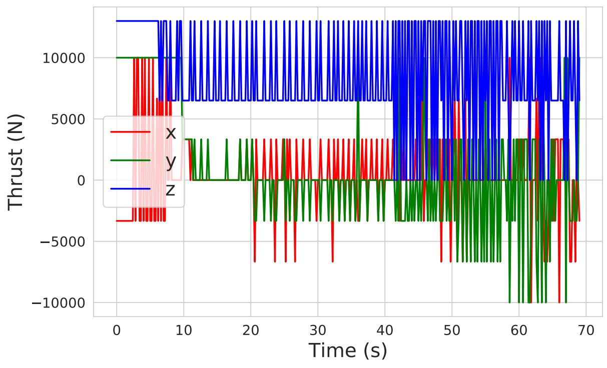

Figure 10: Spacecraft thrusts over a sample trajectory.

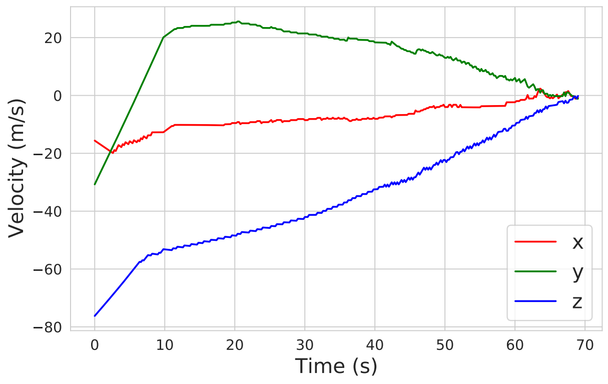

Figure 9: Spacecraft velocities over a sample trajectory.

line with new information. The fast update times for this approach show its suitability for implementation onboard a spacecraft.

The approach used here relied heavily on testing across random seeds to optimise performance. While this can find effective configurations for the agent, this can be a tedious and unreliable process. A more formal approach to improving the agent's performance which is agnostic to random seeds would be beneficial. Furthermore, this approach required offline experience replay to be stored onboard the spacecraft. This could be replaced with a system model which estimates future states and uses these to update the control system, which removes the need for additional onboard memory.

References

[1] C. Wilson, F. Marchetti, M. Di Carlo, A. Riccardi, and E. Minisci, "Classifying Intelligence in Machines: A Taxonomy of Intelligent Control," *Robotics*, vol. 9, no. 3, p. 64, Sep. 2020.

[2] T. Estes, "Deploying and Accelerating AI at the Edge with the NVIDIA EGX Platform," Jul. 2021. [Online]. Available: https://developer.nvidia.com/blog/deploying-and-accelerating-ai-at-the-edge-with-the-nvidia-egx-platform/

[3] J. Wu, "Edge AI Is The Future, Intel And Udacity Are Teaming Up To Train Developers," *Forbes*, Apr. 2020.

[4] G. Mateo-Garcia, J. Veitch-Michaelis, L. Smith, S. V. Oprea, G. Schumann, Y. Gal, A. G. Baydin, and D. Backes, "Towards global flood mapping onboard low cost satellites with machine learning," *Scientific Reports*, vol. 11, no. 1, p. 7249, Mar. 2021.

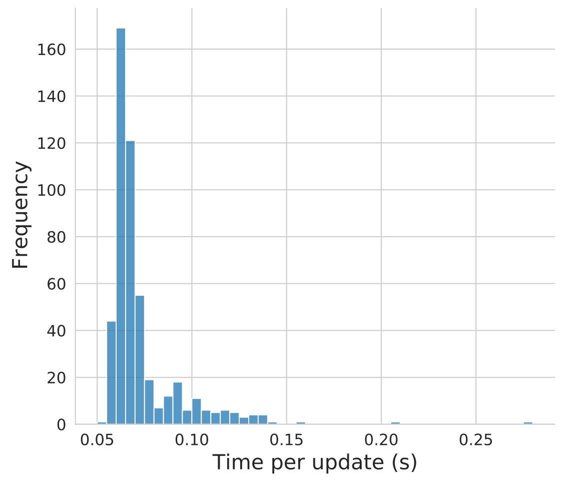

Figure 11: Histogram of time to update the agent's weights online.

[5] R. S. Sutton and A. G. Barto, *Reinforcement learning: an introduction*, second edition ed., ser. Adaptive computation and machine learning series. Cambridge, Massachusetts: The MIT Press, 2018.

[6] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, Oct. 2017.

[7] M. B. Quadrelli, L. J. Wood, J. E. Riedel, M. C. McHenry, M. Aung, L. A. Cangahuala, R. A. Volpe, P. M. Beauchamp, and J. A. Cutts, "Guidance, Navigation, and Control Technology Assessment for Future Planetary Science Missions," *Journal of Guidance, Control, and Dynamics*, vol. 38, no. 7, pp. 1165–1186, Jul. 2015.

[8] B. Gaudet, R. Linares, and R. Furfaro, "Deep reinforcement learning for six degree-of-freedom planetary landing," *Advances in Space Research*, vol. 65, no. 7, pp. 1723–1741, Apr. 2020.

[9] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," *arXiv:1707.06347 [cs]*, Aug. 2017, arXiv: 1707.06347.

[10] C. Wilson and A. Riccardi, "Improving the efficiency of reinforcement learning for a spacecraft powered descent with Q-learning," *Optimization and Engineering*, Oct. 2021. [Online]. Available: https://link.springer.com/10.1007/s11081-021-09687-z

[11] ——, "Leveraging Optimal Control Demonstrations in Reinforcement Learning for Powered Descent," in *2021 8th International Conference on Astrodynamics Tools and Techniques (ICATT)*, Noordwijk, The Netherlands, 2021.

[12] C. Wilson, A. Riccardi, and E. Minisci, "A Novel Update Mechanism for Q-Networks Based On Extreme Learning Machines," in *2020 International Joint Conference on Neural Networks (IJCNN)*. Glasgow, United Kingdom: IEEE, Jul. 2020, pp. 1–7.

[13] G.-B. Huang, Q.-Y. Zhu, and C.-K. Siew, "Extreme learning machine: Theory and applications," *Neurocomputing*, vol. 70, no. 1, pp. 489–501, Dec. 2006.

[14] S. Hariharapura Sheshadri and D. Franklin, "Introducing the Ultimate Starter AI Computer, the NVIDIA Jetson Nano 2GB Developer Kit," Oct. 2020. [Online]. Available: https://developer.nvidia.com/blog/ultimate-starter-ai-computer-jetson-nano-2gb-developer-kit/

[15] "Lockheed Martin and University of Southern California Build Smart CubeSats, La Jument," *Media - Lockheed Martin*, Aug. 2020. [Online]. Available: https://news.lockheedmartin.com/news-releases?item=128962

[16] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3-4, pp. 279–292, May 1992.

[17] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015.

[18] W. Fedus, P. Ramachandran, R. Agarwal, Y. Bengio, H. Larochelle, M. Rowland, and W. Dabney, "Revisiting Fundamentals of Experience Replay," in *International Conference on Machine Learning*. PMLR, Nov. 2020, pp. 3061–3071.