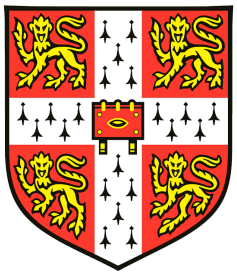
Superior Colliculus-Inspired Reinforcement Learning in Continuous and Discrete Action Spaces



Luke Robert Johnston

Department of Engineering
University of Cambridge
Supervisor: Prof. Guillaume Hennequin

This thesis is submitted for the degree of
Master of Philosophy

Queens' College

January 2024

This thesis is dedicated to my partner AK, without whose support this work would not have been possible

Declaration

This thesis is the result of my own work and includes nothing which is the outcome of work done in collaboration except as declared in the Preface and specified in the text. I further state that no substantial part of my thesis has already been submitted, or, is being concurrently submitted for any such degree, diploma or other qualification at the University of Cambridge or any other University or similar institution except as declared in the Preface and specified in the text. It does not exceed the prescribed 15,000 word limit - excluding figures, footnotes, bibliography, and appendices - of the Engineering Degree Committee.

Luke Johnston
December 2023

Acknowledgements

First and foremost I would like to thank my supervisor Prof. Guillaume Hennequin for his sound advice and support in the development of this work. I began this project with no apriori neuroscience or software development background and so I appreciate his patience in bringing me up to speed. I would additionally like to thank the rest of the Hennequin lab for their warmth and advice throughout the year, along with the wider Computational and Biological Learning lab for providing such a stimulating research environment. I would also like to thank Prof. Marco Tripodi for his welcome advice on this project as well as hospitality during visits to the the MRC-LMB, helping to tie together experimental work from his own lab with our theoretical simulations and proposing potential avenues of further collaboration. Finally, I would like to thank Dr Amma Kyei Mensah for the scholarship which in part enabled this work to be performed, as well as Queens' alumnus Kenneth Perry for the additional grant enabling my attendance at COSYNE 2023 in Montreal.

Abstract

All vision-dependent animals exhibit a range of oculomotor activity during naturalistic behaviours such as navigation and foraging. Foremost among these is the *saccade* – a rapid, ballistic movement of the eyes between points of fixation, the initiation and control of which is believed to be governed by a midbrain region known as the superior colliculus (SC). Links between localised activity in the SC and stereotyped saccadic movements have long been identified, with research indicating the presence of a distinct spatial code based on aligned topographic sensory 'maps' within the SC's layer-based structure. However, the exact form of sensorimotor transform governing the path from visual stimulus to resultant oculomotor action within these layers is yet to be fully characterised. This work represents a small step towards this goal, considering a set of reinforcement learning (RL) simulations for tasks which parallel those believed to be governed by the SC. By analysing the post-training behavioural statistics and network dynamics of our RL agents we aim to provide insight into this complex sensorimotor transform, identifying potential links between our results and open questions in the SC literature. While the various constraints imposed on our models limit the extent to which we can draw direct links to the physiological implementation of sensorimotor transform in the SC, the techniques employed in our simulations serve as a foundation for further work of this nature, representing part of an emerging array of techniques recruited to investigate this perplexing neural structure.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

1.1 Motivation

Of all tasks performed by the central nervous system, few have garnered as much scientific attention as that of the sensorimotor transform. The translation of sensory stimuli to motor commands is a crucial process for all animals and as such represents a complex neural computation, recruiting a large number of interconnected brain regions optimised for this purpose [1]. This work considers in detail one such example of this transform occurring within a midbrain structure known as the superior colliculus (SC). The SC is crucial for controlling rapid reorienting movements toward or away from salient stimuli. These include rapid visual refoveations (saccades, or *gaze shifts*), head rotations, and occasionally other bodily reorienting movements [2–4]. Such movements are important for all vision-dependent animals across a range of behaviours, including foraging, navigation, and the detection and evasion of predators [5, 6]. The SC is uniquely optimised to perform this task, with a layer-based physiology encoding both sensory and motor information in aligned topographical 'maps', allowing direct communication between these domains within a single structure.

Despite the large body of experimental evidence directly linking site-specific activity in the SC with corresponding directed saccades [7–9], the mechanisms by which sensory information is integrated to produce these movements are yet to be fully characterised. One common technique employed to study sensorimotor control in the brain considers the analysis of models trained to perform tasks analogous to those governed by a neural region of interest [10, 11]. Examination of the resultant behaviour and neural dynamics of these models can often provide valuable insights into the biological implementations of corresponding ethological tasks. To this extent, this work considers the training of reinforcement learning (RL) agents to perform tasks analogous to those governed by the SC. The subsequent analyses and discussions aim to provide potential insight into some of the open questions related to the implementation of sensorimotor transform within this perplexing structure.

1.2 Literature Review: Superior Colliculus

The superior colliculus (SC) is a paired structure (i.e. split across both hemispheres) situated on the surface of the midbrain - the uppermost division of the brainstem, located beneath the cortex [5, 12]. Together with the inferior colliculus (IC) just below, these two structures compose the tectum, a region responsible for processing sensory information to produce rapid motor responses toward or away from stimuli of interest [13]. These motor responses consist of reorienting movements of the eyes (saccades),

head, and in some animals, rest of the body. While both superior and inferior colliculi are involved with producing commands for such movements, the SC has traditionally been subject to more extensive study. This is largely due to the sensory modalities associated with each - while the IC is involved mainly with auditory processing [14], the SC is considered multimodal in nature [6]. While predominantly receiving visual input, it additionally integrates somatosensory and auditory information within its structure.

The history of SC research is relatively brief in the context of neuroscience, originating in the 19th century from observations of eye movements in response to tectum stimulation [7]. Thus ensued a period of debate regarding the exact function and necessity of this area, with conflicting experimental findings. Some studies suggested a vital role of this area for gaze reorientations [2, 15], while others cast doubt on its necessity, noting that saccades could be evoked from other brain areas without requiring SC activity [16, 17][1]. In the decades since, the ability to obtain single-unit recordings and perform *in vivo* electrical microstimulation studies have been invaluable in allowing us to build a clearer picture of the role of the SC, and begin to characterise links between the anatomical structure and function of this region.

The SC is a laminar structure - it can be anatomically and functionally divided into superficial layers (SCs) and deep layers (SCd), the latter often further subdivided into an additional intermediate layer (SCi). The superficial layer is considered purely sensory in nature, and receives a range of visual information. It is composed of the stratum opticum (SO) which receives visual information directly from the retina (via the optic tract), stratum griseum superficiale (SGS) which receives inputs from both optic tract and visual cortex[2], and the stratum zonale (SZ), associated with projections to the intermediate and deep layers to modulate the resultant motor signals produced. The intermediate and deep layers have both sensory and motor function, featuring a complex interplay between the two. The intermediate layers can be considered as mediating interactions between the sensory stimuli processed at the superficial layers and motor commands generated in the deep layers. Here signals are received from additional modalities including auditory and somatosensory, in addition to visual. Integration of this information is thought to produce corresponding motor signals, to be processed by the deep layers. The stratum griseum intermediatum (SGI) is concerned primarily with this integration process, and the stratum album intermediatum (SAI) with the transformation of this information into motor signals. Finally, the deep layers are mainly involved with the motor outputs themselves. Both the stratum griseum profundum (SGP) and stratum album profundum (SAP) further integrate across modalities, finalising the motor commands to be projected to motor-execution brain regions responsible for implementing the desired action. The SGP projects mainly to the brainstem and spinal cord, and SAP mainly to thalamus and cortex [19].

One aspect of the superior colliculus that is vital to establish when characterising links between its structure and function is that of the spatial code implemented within its layers. As previously mentioned these layers are considered to implement topographic maps - directly encoding spatial location of both sensory stimuli and motor activity via neurons whose response fields tile egocentric (body-centred) sensorimotor space. Specifically, it is believed the sensory layers contain such a retinotopic map of visual space, with the intermediate layers similarly implementing maps of auditory and somatosensory space. Furthermore the deep layers feature a motor map, activity across which encoding the location of forthcoming movements (discussed in further detail later in this chapter). Importantly, the multimodal sensory maps in the superficial and intermediate layers are considered to be topograhically 'aligned'

[1] A 1966 review [18] summarised them as 'dispensable structures for conjugate gaze'.

[2] While retinal projections via the optic tract refer to 'unprocessed' visual information, information from visual cortex has been processed to some degree (for example, performing feature detection).

[20–22], serving two key purposes in optimising the SC for its task[3]. Firstly, deep SC neurons receiving inputs across multiple sensory modalities exhibit 'multisensory facilitation' [24], where the response evoked in reaction to multiple relevant stimuli can be larger than the sum of their individual responses. These responses are most significant for synchronous, spatially co-located stimuli - as would be the case in important ethological tasks such as hunting or predator avoidance. Additionally, aligned sensory maps in the superficial layer and motor maps in the deep layer allow for an efficient transformation from sensory cues to corresponding motor commands, where a stimulus representation in the sensory map is thought to enable the rapid triggering of a motor action to the corresponding location [20].

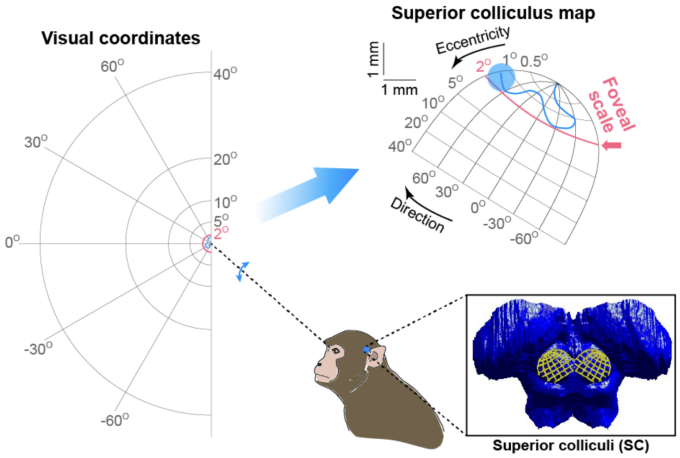


Figure 1.1: **Retinotopic organisation in the SC.** This schematic displays how egocentric visual angles are transformed to spatial location on the SC's surface ('map'), modelled by a parametric transformation based on empirical relations [25]. We observe a clear dominance of foveal (centre of gaze) information (red line), mapping to a large portion of the rostral (anterior) SC. This relative over-representation of foveal information reflects the SC's frequent role in producing small, high accuracy 'microsaccades', and is also consistent with the SC's role in maintaining and adjusting fixation [26, 27]. Reused from Hafed et al. (2021) [28].

The prevailing overview of sensorimotor transformation in the superior colliculus function is therefore that stimuli across sensory modalities, each topographically represented, are transformed via representation on a common motor map to a motor command which results in a subsequent change of gaze direction. Of course, this summary represents a highly simplified overview which does not clarify the mechanisms and physiological processes involved in this transformation. The remainder of this review discusses ongoing research which aims to characterise this complex process, as well as highlight some of the associated questions that remain. In the interests of concision and relevance to this work, many key aspects of SC research will not be addressed. Some notable examples include a precise treatment of brain regions believed to project to and receive connections from the SC, interactions between the SC and frontal eye field (FEF)[4] in mediating saccades and maintaining fixation, and the range of cell types recruited within the SC to produce its characteristic behaviours. Instead, the following sections focus on research relating to the broad mechanistic implementation of sensorimotor transform within the SC, which constitutes the central theme of this work.

To begin with, one vital and particularly complex aspect of this process is the manner in which sensory

[3]While alignment of the sensory maps is considered to be important in transforming sensory signals to resultant activity on the motor map (which in turn encodes the intended movement), it is unclear if the motor map itself is aligned with the sensory maps. Some microstimulation studies suggest this may be the case [23], but there is insufficient evidence to definitively confirm this alignment.

[4]The FEF in the frontal cortex is considered to play a similar role to the SC, also mediating saccades, and has complex interactions with the SC that are not yet understood.

information across modalities is transformed into a common, topographic representation. As visual, auditory and somatosensory targets can all trigger saccades, signals provided by each modality must undergo a coordinate transformation to enable their representation within a shared reference frame. For example, the somatosensory system uses a body-centred reference frame, whereas visual and auditory information is obtained with respect to the positions of the ears and eyes respectively. The SC must subsequently map this information - obtained across a range of egocentric frames - to aligned topographic maps, considered to reflect the egocentric visual reference frame [29, 30]. This is a complex transformation - especially considering the continual, independent movement of each constituent coordinate frame - and as such is yet to be fully characterised. One ongoing branch of research attempts to obtain insight into this process by considering how these maps might be aligned during an animal's development [31]. Finally, it is not clear to what extent, if any, the SC is able to maintain any allocentric (environment-centred) representations of information, in addition to that represented on its egocentric maps.

An area that represents a substantial proportion of ongoing research regarding sensorimotor transform in the SC considers the form of population code implemented on the motor map within its deep layers. Much experimental evidence suggests activity encoding movement to be highly localised on this map, spatially encoding (egocentric) endpoint of the proceeding saccade. Such activity is represented as a locus or 'mound' - considered invariant in size and intensity with respect to the encoded saccade [25] - however it is not fully understood however how wider population activity is transformed into this locus and how it is subsequently decoded. Though some theories exist based on probabilistic coding strategies [32, 33], a 'winner-takes-all' scheme [34, 35] is generally considered the most popular candidate. In this form of coding local excitatory units compete for activation, with the 'winner' suppressing other units via long range inhibitory feedback, such that only a small group of neurons remain active. Though much evidence exists for such a continuous attractor network in the SC [36, 37], some experimental data also casts doubt on this theory, with some studies suggesting such necessary inhibitory connections may not exist in the deep motor layers [38–40]. Additionally, experiments measuring SC activity during presentation of two concurrent stimuli [38, 39] - producing a 'averaging saccade' landing somewhere between the two stimuli - suggest that activity representing such a saccade is encoded at motor locations corresponding to both stimuli, further casting doubt on such a singular representation [41].

While such mechanisms may contribute to the selection of a spatially localised neural population which encodes the upcoming saccade, there is debate regarding how such a spatial code is decoded into the temporal signal signal interpreted by downstream motor areas. Theories surrounding this process can be broadly split into two competing models: vector averaging [42–45] and vector summation [46–49]. The former, in its simplest form, considers the SC to specify (egocentric) coordinates of the upcoming saccade via an activity-weighted average of each 'movement vector' represented by a neuron's location in the motor map[5]. An additional constant is also sometimes included in the normalising term to better incorporate experimental data. The latter similarly considers a vector summation of SC activity to model the saccade. Early 'static' models of both schemes therefore reflect a simple form of population coding, where SC motor activity is read out and decoded by downstream motor areas to implement the encoded saccade [8, 23, 50, 51]. Dynamic, temporal properties of the saccade are then considered to be governed by feedback mechanisms contained within the downstream motor areas responsible for producing the saccade. In such models, the SC is not situated directly in the motor feedback loop governing the saccade.

Accumulating modern experimental evidence, however, appears to suggest that the level of activity in

[5]Sometimes referred to as taking the 'population vector average'.

the SC additionally influences saccade kinematics. This casts doubt on the accuracy of the previous static decoding models, which are unable to incorporate this temporal aspect. Such experiments showed that although site of SC microstimulation predominantly determines movement location, the statistics of this stimulation additionally modulate the resulting motor response [52–55]. Specifically, both duration and magnitude of microstimulation were shown to affect achievement of the theoretical maximum saccade amplitude (as specified by the spatial location of activity), with frequency of stimulation also found to affect velocity and latency of movement. While microstimulation studies cannot paint a full picture, they provide convincing evidence for the existence of a *spatiotemporal* code in the deep motor layers of the SC, where both spatial and temporal activity dynamically govern saccade characteristics. The exact form of this code is still the subject of much debate[6]. One popular theory - known as the *dual coding hypothesis* [44] - succinctly theorises that spatial location of maximal discharge determines the saccade vector, whereas activity rate governs speed of movement.

To this extent, modern models of saccadic activity attempt to better reflect experimental data by explicitly incorporating the temporal aspect SC coding. In contrast to the static, read-out based models described above, such models consider the temporal activity of the SC to dynamically modulate the ongoing saccade, via a range of potential mechanisms. In the case of vector averaging, models have been developed in which the level of neural activity modulates the gain within the wider motor feedback loop [44, 45][7]. A dynamic model of the vector summation theory has also been developed. In these models - termed 'dynamic ensemble coding' - the integral of activity over time describes momentary intended saccade trajectory and hence dynamically governs movement kinematics [48, 49]. Finally, models have been proposed which situate the SC directly in the motor loop, considering the possibility the SC may dynamically encode motor error of the ongoing saccade [57, 58]. A further discussion of these models, including a more involved mathematical treatment, is given in Appendix A.

Finally, there is also debate regarding the anatomical form of the sensory and motor maps within the SC. One theory that has gained traction in recent decades proposes these maps to be modular in nature - that is, composed of discrete modules each associated with a unique spatial location, arranged in a honeycomb structure. These modules are columnar in nature and extend across layers, reminiscent of similar structures found in cortex [59]. While evidence has long existed for such organisation of sensory maps in the superficial layers [60], recent research shows such a structure to be similarly mirrored in the deep motor layers [33]. This structural organisation gives rise to notions of the SC as an 'addressable and modularly organised spatial-motor register'. Under such a hypothesis each module can be considered a 'functional orienting unit', receiving triggering information from a given region in sensory space and containing premotor neurons for the generation of an angular movement to the corresponding egocentric location. Supporting evidence of such modularity consists of both histological enzyme-staining techniques [61, 62], as well more recently genetic characterisation [33] (Figure 1.2). Such evidence has led some researchers to consider the SC as a 'global spatial indexing system' [33], and further speculate regarding the SC's ability to direct covert responses in addition to full, overt motor movements - i.e. 'activating' a module in such a way to simulate its downstream (i.e. environmental) effect. Further investigation of this theory is needed, but it could potentially tie into theories regarding the SC's involvement in motor planning [63] via internal movement simulation.

To conclude, it is clear further work is needed to fully characterise the complex process of sensorimotor transform within the layers of the superior colliculus. Some pertinent questions highlighted above

[6]The challenge of deciphering this code - and converting it into the purely temporal signal interpreted by downstream motor areas - is often termed the 'spatial-to-temporal transform problem' [8]. The related problem of extracting separate signals for the horizontal and vertical burst generator circuits is known as the 'vector decomposition problem' [56].

[7]Specifically that of the brainstem burst generator, acting as a form of differential controller by producing sharp oculomotor movements in response to large differences in motor error.

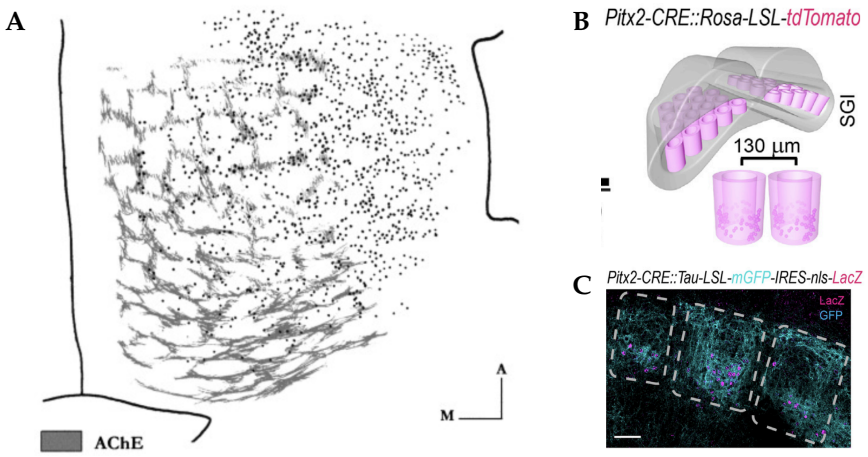


Figure 1.2: **Evidence of a modular structure in the superior colliculus. (A)** Left: Enzyme staining techniques. **(A)** Enzyme staining with acetylcholinesterase (AChE) reveal a 'honeycomb' structure in the cat SC. Such a protein is used to break down the neurotransmitter acetylcholine into acetate and choline, thereby indicating dense synaptic activity. Reused from Chevalier & Mana (2000) [61]. Right: Genetic characterisation. **(B)** Schematic describing the discrete anatomical modules observed to tile the stratum griseum intermediale (SGI) within the deep layers of the SC. These modules were mapped and characterised via genetic profiling techniques utilising fluorescence markers **(C)**, informed by the genetic expression patterns of the *Pitx2* gene in modular neurons. Reused from Masullo et al. (2019) [33].

include the form of spatiotemporal code within the SC maps; the role of feedback in achieving saccadic motor control, and the role of a possibly modular anatomical structure within the sensory and motor maps. Through the subsequent analysis of simulation data in the proceeding sections of this report, we will return to these questions and attempt to offer some relevant discussion and commentary, including informed suggestions for relevant potential future work.

1.3 Literature Review: Reinforcement Learning

In this work, the methods used to train our models are taken largely from a branch of machine learning known as reinforcement learning (RL). Considered as one of the three main machine learning paradigms[8], reinforcement learning involves training an agent to take actions in its environment which maximise cumulative reward. Desirable actions are 'reinforced' vi this reward signal, and the agent learns to adjust its decision-making policy accordingly. Such a paradigm has immediate ties to neuroscience when compared to the ethological behaviour of animals - for example considering abstract 'reward' to be a desirable quantity such as food[9]. It is no surprise, therefore, that there is a long history in computational neuroscience of developing and analysing artificial reinforcement learning models ('artificial agents') and drawing subsequent links to the neural dynamics governing biological learning and behaviour.

A famous case study which exemplifies this relationship involves an RL algorithm known as temporal difference (TD) learning and its links to learning dynamics implemented by the brain. TD learning is a type of model-free RL method (it does not explicitly require a model of the environment) used to estimate the value of given states or actions [64]. TD learning aims to predict future rewards, and continually update these predictions, based on the difference between expected and received rewards -

[8]The other two being supervised and unsupervised learning.

[9]Indeed when training animals to perform tasks in experimental neuroscience, a food or juice-based reward is commonly used to encourage desired behaviour.

a quantity known as the reward prediction error (RPE), or TD error. In the subsequent decade after this model was proposed, data from monkey experiments appeared to suggest dopamine neurons in the midbrain encoded exactly such a quantity [65]. Specifically, the phasic firing of these neurons appeared to correlate with the level of reward prediction error predicted by TD learning under comparable conditions. Computational frameworks have since been developed to explicitly model this observation, mechanistically linking the two and suggesting that the brain potentially uses a form of this algorithm as a reward-based learning process [66].

In recent years, an active area of RL research that has facilitated many neuroscience based discoveries is that of meta reinforcement learning (meta-RL). While traditional reinforcement learning considers models trained to optimally perform a single task, meta-RL examines agents trained simultaneously across multiple tasks, considering to what extent such agents can 'learn to learn'. By modelling these agents with complex, biologically inspired neural networks and exposing them to a range of training tasks, such agents often obtain the fascinating emergent ability to rapidly adapt to unseen, novel tasks. Such a paradigm has arguably strong ties to ethological animal behaviour, which often requires rapid, context-dependent learning and decision making.

These agents are typically based on models known as Recurrent Neural Networks (RNNs), a powerful class of neural network models. Such networks don't process information strictly in a feedforward manner but also in loop-like interactions across the network, directly mimicking patterns seen in neural systems where such recurrent connectivity has long been observed.[10] Architecturally, these networks are characterised by an internal state which evolves over time, enabling strong task performance by acting as a form of memory. One critical problem with these models, however, is that the recurrent nature of their activity can complicate the training process [68].

A class of RNN's that have received much attention in meta-RL - both due to their ability to circumvent this problem, as well as generally strong performance - are known as Long Short-Term Memory networks (LSTMs) [69]. These networks are characterised by inclusion of a 'gating' mechanism which dynamically controls the flow of information within each LSTM cell, governing to what extent new information is incorporated into the current state of the network and enabling the learning of long sequences of dependencies. This work utilises a variant of these networks known as the Gated Recurrent Unit (GRU) [70] which utilises a similar system, noted for its tradeoff between simplicity and performance (see Chapter 2) [71]. As with the broader concept of recurrent connectivity, the gating mechanisms key to the success of these networks have a strong degree of neural analogy.[11]

Having briefly described the networks underpinning recent advances in RL, we can discuss some recent discoveries within this field and their neuroscience implications. One key showcase of the neuroscience implications of meta-RL demonstrated LSTM-based agents developing the ability to 'learn to learn' across a range of tasks [72]. Such work provides evidence these networks can capture meta-learning processes within their internal dynamics to enable rapid adaptation, and offers a model to potentially understand similar capabilities in biological systems. Similar links were made in work proposing a

[10] Discoveries of recurrent connectivity in the brain have been made across a range of physical scales. At the microscale, such connectivity is implemented in synaptic feedback loops, where neurons form recurrent loops with nearby neurons (or even themselves [67]). At mesoscale (between brain regions) a famous example is the thalamocortical loop, where the thalamus - considered a 'relay station' for sensory information - forms a circular pathway with the cortex. After processing in sensory areas of cortex, information is then sent back to the thalamus for relay to other cortical areas. Finally at macroscale, there exist large scale networks such as the default mode network - comprising multiple recurrently connected brain regions responsible for a range of internal thinking patterns. Functional loops involving attention (such as the frontal eye field and visual cortex for visual attention), and memory (most famously the hippocampal circuits) are also widely considered dependent on recurrent interaction.

[11] For example, inhibitory neurons classically release neurotransmitters to inhibit firing of nearby neurons with the purpose of selectively suppressing and prioritising signals, much like a gating system. Additionally, synaptic plasticity (the strengthening or weakening of synapses over time via long term potentiation or depression) could be considered to represent slowly modulating 'gates' in the brain.

meta-RL algorithm called RL^2 [73], explicitly separating long and short timescale learning via use of an RNN with a 'slow' outer loop and 'fast' inner loop, governed by the model weights and hidden state respectively. Such a model once again demonstrates generalisation across RL algorithms to enable rapid adaptation, drawing interesting parallels to similar hierarchical learning mechanisms that could exist in the brain. Other work considers more direct links, proposing the human prefrontal cortex (PFC) to operate as a self-contained meta-RL system [74], drawing links with neuroanatomy to argue the PFC implements flexible task-specific adaptation similarly to the models developed. Links have also been made to the exploration strategies of meta-RL systems and their biological counterparts [75], highlighting how agents learn structured exploration strategies parallelling those of humans. Other work considers links between artificial and biological learning strategies through the lens of cognitive psychology [76], suggesting both model-free ('fast') and model-based ('slow') learning strategies can coexist in biological and artificial agents, with parallels between their respective implementations.

It is therefore clear computational modelling based on techniques from meta-RL has much to offer the field of neuroscience. While the complex RNN models deployed in this field can never truly capture the intricacies of neural systems, analysis of the emergent behaviour of these models has yet elucidated compelling links between biological and artificial agents, with clear potential for further discovery.

1.4 Overview

Having established the context in which this work sits, we can formalise the broad goal of this project as the application of reinforcement learning to tasks inspired by the role of the superior colliculus, to draw inferences which could provide insight in the many research questions surrounding this structure. Chapter 2 features a brief overview of the methodologies used for our simulations. Chapter 3 considers simulations involving an agent with continuous angular action space, trained to solve a task involving successive exploration and exploitation of rewarded locations. We subsequently examine how the agent's *meta-learning* capability to adapt to a more complex version of the task. Chapter 4 considers a more neurophysiologically inspired model where our agent performs discrete, saccadic movements, with the additional option of simulating (planning) an action via a pre-trained predictive model. It is then trained to solve a task involving tracking a moving target through space. Subsequent analysis considers how the agent's visual input is transformed to a policy over actions, and how the target is represented within this policy. We also consider how planned movements are utilised to improve task performance. Finally, Chapter 5 summarises our findings, discussing the implications of these results with respect to our understanding of the SC and proposing future work based on further development of our simulations.

Chapter 2

Methodology

This chapter briefly describes the methods used to train and analyse the models employed in the simulations of this work.

2.1 Recurrent Neural Networks

Recurrent Neural Networks (RNNs) are typically the model of choice employed to simulate neural dynamics and can be broadly split into two categories [77]: rate networks, representing neural activity with a continuous variable[1]

$$\mathbf{r}(t), \;\; \mathbf{r} \in \mathbb{R}, \;\; t \geq 0 \tag{2.1}$$

and spike-based networks, which consider a timeseries of discrete spiking events as

$$\mathbf{s}(t), \;\; \mathbf{s} \in \{0, 1\}, \;\; t \geq 0 \tag{2.2}$$

Each utilise different computational techniques and have different modelling use-cases. This work considers models built on the former, given there is little evidence of an explicit spike timing-based code in the SC. Additionally these models are much simpler to work with, due to their greater efficiency and improved analytical tractability [78, 79]. In practice, rate-based RNNs are typically described using a discrete-time update equation

$$\mathbf{h}_t = f(\mathbf{h}_{t-1}, \mathbf{x}_t; \theta), \;\; t \in \{1, 2, ...\} \tag{2.3}$$

where $\mathbf{h}$ is the vectorised hidden state[2], $\mathbf{x}$ is the network's input, and θ are the network parameters. The output of the network - often conceptualised as the 'read-out' - is then computed from the evolving hidden state as

$$\mathbf{y}_t = \mathbf{V}\mathbf{h}_t + \mathbf{b} \tag{2.4}$$

[1] Such a variable is often used to denote neural firing rate per unit time, but can at times also be considered to represent synaptic efficiency (strength) or membrane potential of a given neuron.

[2] The hidden state is an abstract quantity, often considered to represent the 'memory' of the network. Inputs and outputs of the network $\mathbf{x}$ and $\mathbf{y}$ are then considered more analagous to the neural firing rate variable from eq. (2.1).

2.1.1 Vanilla RNNs

In a *vanilla* rate-based RNN - 'vanilla' denoting a default, unmodified model - evolution of the hidden state can be described in continuous time by the differential equation

$$\tau \frac{d\mathbf{h}(t)}{dt} = -\mathbf{h}(t) + \phi(\mathbf{W}_h \mathbf{x}(t) + \mathbf{U}_h \mathbf{h}(t) + \mathbf{b}_h), \;\; t \geq 0 \tag{2.5}$$

where $\mathbf{W}_h$ is the input (or 'read-in') matrix, $\mathbf{U}_h$ governs recurrent activity, $\mathbf{b}_h$ is a bias term, and ϕ is the network's activation function, typically $\tanh()$ or a sigmoid. The output of the RNN is computed via a read-out, such as that in eq. (2.4).

When simulating the dynamics evoked by these equations, we implement a discrete-time approximation to obtain an update equation of the form in eq. (2.3). Using Euler's method, we consider a small timestep Δt and multiply through eq. (2.5) to obtain

$$\mathbf{h}(t + \Delta t) = \mathbf{h}(t) + \frac{\Delta t}{\tau}(-\mathbf{h}(t) + \phi(\mathbf{W}\mathbf{h}(t) + \mathbf{U}\mathbf{x}(t) + \mathbf{b})) \tag{2.6}$$

$$= (1 - \frac{\Delta t}{\tau})\mathbf{h}(t) + \frac{\Delta t}{\tau}\phi(\mathbf{W}\mathbf{h}(t) + \mathbf{U}\mathbf{x}(t) + \mathbf{b}) \tag{2.7}$$

where $\frac{\Delta t}{\tau}$ represents the effective 'update rate' of the discrete time system, governing the magnitude of transition between timesteps. In practise we typically consider the following simplified form with this term implicitly absorbed into our update rule

$$\mathbf{h}_t = \phi(\mathbf{W}\mathbf{x}_t + \mathbf{U}\mathbf{h}_{t-1} + \mathbf{b}_h) \tag{2.8}$$

These dynamics are then simulated for an arbitrary length of time, with readouts taken as necessary.

2.1.2 Gated Recurrent Unit

The Gated Recurrent Unit (GRU) is a class of rate-based RNN that builds on the simple structure described above via the concept of gating [80]. Information now flows through two input channels, known as update and reset gates, producing intermediate variables $\mathbf{z}$ and $\mathbf{r}$ respectively which govern to what extent new information is combined with the unit's existing hidden state

$$\mathbf{z}_t = \sigma(\mathbf{W}_z \mathbf{x}_t + \mathbf{U}_z \mathbf{h}_{t-1} + \mathbf{b}_z) \tag{2.9}$$

$$\mathbf{r}_t = \sigma(\mathbf{W}_r \mathbf{x}_t + \mathbf{U}_r \mathbf{h}_{t-1} + \mathbf{b}_r) \tag{2.10}$$

where σ denotes a sigmoid activation. A candidate hidden state and final output hidden state are then calculated as

$$\hat{\mathbf{h}}_t = \phi(\mathbf{W}_h \mathbf{x}_t + \mathbf{U}_h(\mathbf{r}_t \odot \mathbf{h}_{t-1}) + \mathbf{b}_h) \tag{2.11}$$

$$\mathbf{h}_t = (1 - \mathbf{z}_t) \odot \mathbf{h}_{t-1} + \mathbf{z}_t \odot \hat{\mathbf{h}}_t \tag{2.12}$$

where ϕ is a $\tanh()$ activation and $\odot$ denotes a Hadamard product. The intuition for the reset gate can be considered as determining how to combine the input $\mathbf{x}_t$ with the unit's 'memory' $\mathbf{h}_{t-1}$, and the update gate as deciding the relative proportions of memory and new information carried forward - where $\mathbf{z}_t = 0$ corresponds to erasing all memory, and $\mathbf{z}_t = 1$ corresponds to ignoring new information entirely. As detailed previously, such a structure enables the learning of long sequences of dependencies and hence strong performance in our RL tasks of interest. Figure 2.1 describes this process diagrammatically.

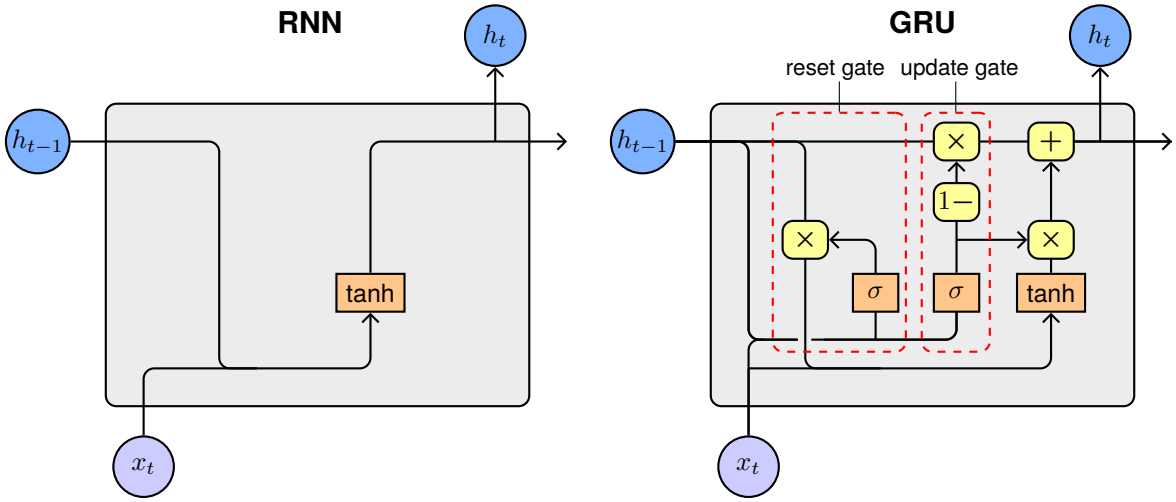


Figure 2.1: **Single unit for a vanilla RNN and GRU.** Left: Vanilla RNN unit. Input $\mathbf{x}_t$ is combined with previous hidden state $\mathbf{h}_{t-1}$ then passed through an activation function (in this case $\tanh()$) to produce new hidden state $\mathbf{h}_t$, as described in eq. (2.8). Right: GRU unit. Input $\mathbf{x}_t$ now flows through the update and reset gates to produce $\mathbf{z}_t$ and $\mathbf{r}_t$ respectively. $\mathbf{x}_t$, $\mathbf{z}_t$ and $\mathbf{x}_t$ are then combined with previous hidden state $\mathbf{h}_{t-1}$ to produce $\mathbf{h}_t$ as described by the operations in the diagram and eqs. (2.9-2.12).

2.2 Network Optimisation

The goal of network optimisation is then to find the set of model parameters $\theta = \{..., \mathbf{W}_k, \mathbf{U}_k, \mathbf{b}_k, ...\}$ that optimises task performance. This procedure is typically performed via the minimisation of some loss $\mathcal{L}(\theta)$, where formally

$$\theta^* = \underset{\theta}{\operatorname{argmin}}\, \mathcal{L}(\theta) \tag{2.13}$$

and θ^* are the optimal parameters we seek. Given the high dimensionality of the parameter space, this is a complex task requiring sophisticated optimisation techniques.

2.2.1 Gradient Descent

This objective minimisation is typically accomplished with gradient descent, a popular technique which follows the gradient of steepest descent in parameter space, implementing an iterative update rule of the form

$$\theta_{t+1} = f(\theta_t, \nabla_\theta \mathcal{L}(\theta_t)) \tag{2.14}$$

where in the simplest case $f(\cdot) = \theta_t - \eta \nabla \mathcal{L}(\theta_t)$, and η is a constant. In this work we use Adam [81], a popular gradient-based optimiser due to its efficiency and stability. It modifies the previous simple update rule to incorporate both first and second moments of the gradient via m_t and v_t respectively, where

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\nabla_\theta \mathcal{L}(\theta_t), \quad v_t = \beta_2 v_{t-1} + (1-\beta_2)\left(\nabla_\theta \mathcal{L}(\theta_t)\right)^2 \tag{2.15}$$

and β are set to high values[3] to prevent strong deviations from the current trajectory. The update rule incorporates bias-corrected versions of these terms, providing adaptive learning rates for each parameter as

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t + \epsilon}}, \quad \text{where } \hat{m}_t = \frac{m_t}{1-\beta_1}, \; \hat{v}_t = \frac{v_t}{1-\beta_2} \tag{2.16}$$

[3]By default, $(\beta_1, \beta_2) = (0.9, 0.999)$.

where ϵ is a small constant for numerical stability. Such adaptive learning rates are particularly useful for dealing with the often sparse gradients provided by the sequence data processed by RNNs. The momentum provided by m_t and v_t are also useful for dealing with the vanishing and exploding gradient problems, providing efficiency and stability when encountering flat and steep areas of the loss landscape respectively [82, 83].

2.2.2 Automatic Differentiation

Given the dependence of gradient-based optimisation on $\nabla\mathcal{L}(\theta_t)$, it is essential we can efficiently and accurately compute this quantity. Automatic differentiation (AD) provides a set of techniques to elegantly calculate our jacobian at code level, provided there exists a set of differentiable relationships between our function inputs and outputs. By leveraging the chain rule to propagate derivatives between nodes on the computational graph formed by our loss function, AD provides access to the function jacobian in constant time with respect to the function evaluation itself [84][4]. AD can operate in both forward and reverse modes, though is typically employed in reverse-mode given its greater efficiency in the common case of multi-input single-output functions [86][5]. As a brief mathematical treatment of reverse-mode AD, consider some function $g : \mathbb{R}^n \to \mathbb{R}$ which consists of a stacked series of L linear transformations each followed by a non-linear activation function. The forward pass through the l^{th} layer is

$$\mathbf{z}^{(l)} = \mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)} \tag{2.17}$$

$$\mathbf{a}^{(l)} = \sigma(\mathbf{z}^{(l)}) \tag{2.18}$$

with loss evaluated as

$$\mathcal{L}(\theta) = \text{LossFunction}\left(\mathbf{a}^{(L)}, \mathbf{y}^*; \theta\right) \tag{2.19}$$

where $\mathbf{y}^*$ is the ground truth for which our loss is calculated against[6]. We then begin the reverse pass, calculating derivatives for the final layer

$$\delta^{(L)} = \frac{\partial\mathcal{L}}{\partial\mathbf{a}^{(L)}} \odot \sigma'(\mathbf{z}^{(L)}) \tag{2.20}$$

where $\delta^{(l)}$ represents the derivative of the loss with respect to $\mathbf{z}^{(l)}$, the pre-activation values at the l^{th} layer. The corresponding parameter gradients are then

$$\frac{\partial\mathcal{L}}{\partial\mathbf{W}^{(L)}} = \delta^{(L)}(\mathbf{a}^{(L-1)})^\top \tag{2.21}$$

$$\frac{\partial\mathcal{L}}{\partial\mathbf{b}^{(L)}} = \delta^{(L)} \tag{2.22}$$

Applying the chain rule, we can propagate these gradients backwards to an arbitrary layer

$$\delta^{(l)} = (\mathbf{W}^{(l+1)})^\top\delta^{(l+1)} \odot \sigma'\left(\mathbf{z}^{(l)}\right) \tag{2.23}$$

where intermediate values $\mathbf{z}^{(l)}$ and $\mathbf{a}^{(l)}$ are typically stored in memory during the forward pass for efficiency. After propagating back to our input layer, we have assembled the total jacobian $\nabla\mathcal{L}(\theta)$ for use with our gradient-based optimiser.

[4] In other words for a function evaluation of time complexity $\mathcal{O}(M)$, jacobian evaluation is similarly $\mathcal{O}(M)$, slower typically by only a small constant. This is often termed the 'cheap gradient principle' [85].

[5] For some gradient-requiring function $f : \mathbb{R}^n \to \mathbb{R}^m$, forward mode is more efficient for $n < m$, but backwards mode more efficient for $n > m$. Typically m is very large and $n = 1$, and so reverse-mode is employed far more often.

[6] In the case $\mathbf{y}$ is available we denote such a paradigm *supervised learning*, and where the loss is formed solely from the input data itself, *unsupervised learning*.

2.2.3 Backpropagation Through Time

Backpropagation refers to the algorithm which combines reverse-mode AD with an optimisation step to iteratively update network parameters until some convergence criteria is met[7]. To apply backpropagation to RNNs however, a more involved approach must be taken due to the temporal dependencies between the weights and activations across the network, including the loops formed by recurrent terms. Backpropagation Through Time (BPTT) adapts the backpropagation algorithm to this temporal context by considering an 'unrolled' version of an RNN such that it resembles a feedforward network (Figure 2.2) [88, 89]. During the backwards pass the calculation of gradients then take into account the influence

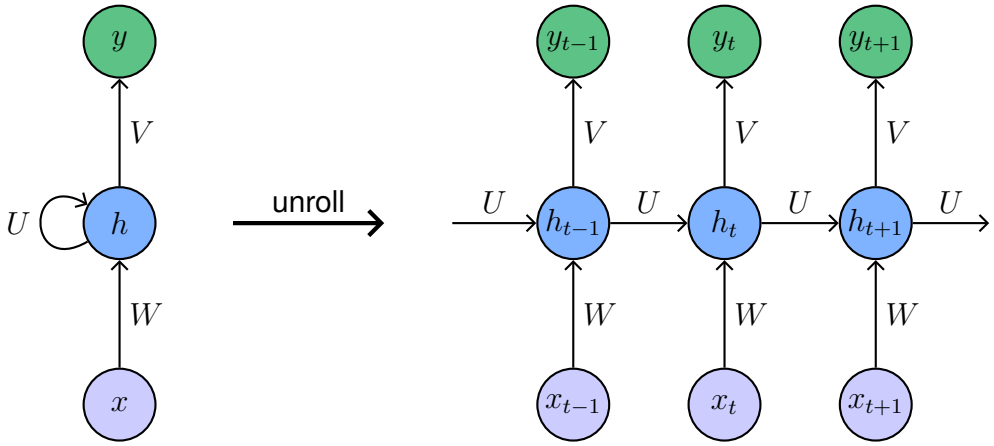


Figure 2.2: **RNN unrolled to a feedforward network.** By 'unrolling' the recurrent element of our network across time, we observe how an RNN can be framed as a feedforward network - now with a shared weights across each layer. The backpropagation algorithm can then be extended to this temporal context - known as *backpropagation through time* - as explained in the main text and Algorithm 1.

of each activation at subsequent (causal) timesteps; such gradients are accumulated over all timesteps to perform a single update of the shared weights. A brief overview of this algorithm is described below.

Algorithm 1: Backpropagation Through Time (BPTT)

input : Input data $\mathbf{x}$, initialised network parameters θ_0
parameters: Network parameters θ, hyperparameters (episode length T, number of epochs N, learning rate η, etc.)

$\theta \leftarrow \theta_0$ % Initialise network parameters
while $i < N$ ***or*** *not converged* **do**
  % Forward pass through time
  for $t = 1 : T$ **do**
    $\mathbf{h}_t \leftarrow f(\mathbf{h}_{t-1}, \mathbf{x}_t; \theta)$ % Update hidden state with RNN equation
    $\mathbf{y}_t \leftarrow g(\mathbf{h}_t; \theta)$ % Readout output
  % Compute total loss
  $\mathcal{L}(\theta) \leftarrow \sum_{t=1}^{T} \text{LossFunction}(\mathbf{y}_t, \mathbf{y}_t^*; \theta)$ % Sum loss over timesteps
  % Backward pass through time
  $\delta\theta \leftarrow 0$
  for $t' = T : 1$ **do**
    $\delta\theta \leftarrow \delta\theta + \nabla\mathcal{L}(\theta)\big|_{t=t'}$ % Accumulate gradients w.r.t θ across timesteps
  $\theta \leftarrow g(\theta, \delta\theta)$ % Update θ with gradient-based optimiser of choice
return θ^* % Optimised network parameters

[7]Conceptualised in the 80's, this technique is now considered a cornerstone of artificial neural network training, and is often credited in part for the rapid growth of the field of machine learning [87].

2.3 Policy Gradient Methods

While minimising some loss $\mathcal{L}(\theta)$ to optimise our network is often a straightforward process in the supervised case, the previously discussed techniques are typically not directly transferrable to reinforcement learning. This motivates the use of alternative methods to optimise performance of the RL agents trained in this work - a popular choice for this are *policy gradient* methods.

2.3.1 Motivation

In RL we seek to *maximise* cumulative reward of our agent with the objective[8]

$$J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_t^T r(\mathbf{s}_t, \mathbf{a}_t) \right] = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[R(\tau) \right] \tag{2.24}$$

where *trajectories* $\tau = (\mathbf{s}_1, \mathbf{o}_1, \mathbf{a}_1, r_1, \mathbf{s}_2, \mathbf{o}_2, \mathbf{a}_2, r_2, ..., \mathbf{s}_T, \mathbf{o}_T, \mathbf{a}_T, r_T)$ are sampled from some distribution

$$p_\theta(\tau) = p(\mathbf{s}_1) \prod_{t=1}^{T} \pi_\theta(\mathbf{a}_t|\mathbf{s}_t) p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t) \tag{2.25}$$

and $\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$ denotes the agent's *policy* (in the simplest case of full observability), $p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$ is the *state transition* probability, and $R(\tau)$ is shorthand for the total reward obtained over such a trajectory. The relationship between trajectory variables is often described graphically within the *RL loop* (Figure 2.3A). In this work we are interested in the *partially observed* variant of this model where our agent obtains incomplete state information from the environment via some state observation $\mathbf{o}_t$, hence we include this variable as part of our sampled trajectory.

The need for alternative methods to optimise this RL objective can be graphically illustrated by framing the process describing our partially observed trajectory as a variant of the Markov Decision Process (MDP) known as a Partially Observed Markov Decision Process (POMDP) (Figure 2.3B), given certain assumptions made to preserve the Markov property. Examining this MDP through the lens of a computational graph, the inherent unpredictability of the process becomes evident: the agent's policy is typically stochastic (actions are sampled), and state transitions are also often probabilistic. There is therefore no guarantee the relationships between nodes are known, let alone deterministic and differentiable, and as such direct backpropagation through this graph is typically not possible. Additionally, the sparse and often delayed nature of rewards obtained can make the training signal difficult to interpret[9]. To optimise our agent's parameters we therefore turn to a separate class of algorithms known as policy gradient methods, which instead seek to directly optimise the agent's policy.

2.3.2 Vanilla Policy Gradients

The RL objective above can be estimated with Monte Carlo methods, where we sample a large number of trajectories and compute their rewards for a given policy π_θ, with the accuracy of our estimation improving with the number of sampled trajectories. Policy gradients methods seek to estimate $\nabla J(\theta)$ with a similarly sampling-based approach. We start by expressing our objective as a integral over trajectories[10]

$$J(\theta) = \int p_\theta(\tau) R(\tau) d\tau \tag{2.26}$$

[8]Despite the fact the reward function considered in our work is a function of state only, in the following section we consider reward as a function of both state and action, in keeping with the literature.

[9]This is often referred to as the *credit assignment problem*.

[10]Note such an analytic expression assumes a hypothetical continuous trajectory-space.

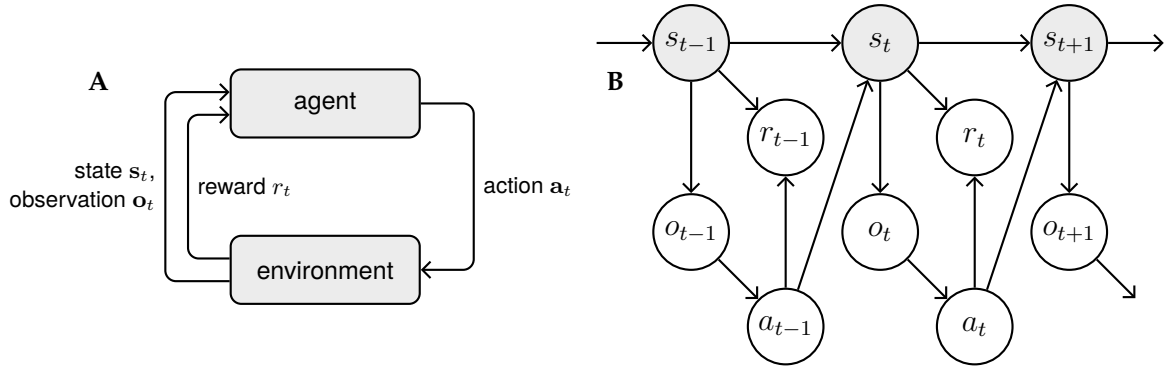


Figure 2.3: **RL loop and corresponding POMDP. (A)** *RL loop* for our partially observed paradigm, displaying the broad relationships between action $\mathbf{a}_t$, state $\mathbf{s}_t$, observation $\mathbf{o}_t$ and reward r_t. In the fully observed case, the agent's *policy* $\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$ denotes the network-parameterised distribution over actions $\mathbf{a}_t$ given full state observation $\mathbf{s}_t$. Reward received is then given by $r(\mathbf{s}_t, \mathbf{a}_t)$, and subsequent state encountered by the state transition probability $p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)$. Note such policy and state transition terms assume the Markov property - that past states and actions have no bearing on the current policy or state transition given the current state and action - as is often justified in the case of full observability. In our case of *partial* observability however - where state observations $\mathbf{o}_t$ replace full state information - such a property is typically not assumed, as in our work. Instead, decisions made at each timestep are considered to be conditioned on the entire *history* of past observations, actions and rewards denoted $H_t = (\mathbf{o}_{1:t}, \mathbf{a}_{1:t-1}, r_{1:t-1})$. All information from this history relevant to the agent's decisions is considered to be represented within the rolling hidden state $\mathbf{h}_t$ of our agent's recurrent network, such that our agent's policy at each timestep can be described $\pi_\theta(\mathbf{a}_t|\mathbf{h}_t)$. **(B)** The Partially Observed Markov Decision Process (POMDP) corresponding to this loop. To justify the treatment of our partially observed paradigm as a MDP, we must extend the definition of a Markov Process to enable our agent to make decisions conditional on past information while retaining the Markov property. This is typically done via introduction of the concept of a *belief state* $b(\mathbf{s}_t)$. This is an implicit distribution (i.e. it is never explicitly computed) over the agent's theorised possible states formed from the history of trial information H_t. This serves as a way to encapsulate any relevant information contained in the sequence of past observations, actions and rewards into a single quantity that is updated at each step. Therefore conditioned on the current belief state $b(\mathbf{s}_t)$ (not shown), our partially observed decision process preserves the Markov property. While the belief state is a useful concept in describing our process as a POMDP, in practise it is simpler to consider our agent's policy as conditioned on rolling hidden state of the network i.e. $\pi_\theta(\mathbf{a}_t|\mathbf{h}_t)$, given this is how decisions are made at code level (both trial history H_t and belief state $b(\mathbf{s}_t)$ can equally be considered to be captured within this state).

where our gradient term is then

$$\nabla_\theta J(\theta) = \int \nabla_\theta p_\theta(\tau) R(\tau) d\tau \tag{2.27}$$

The following identity, known as the *log-trick*, is then crucial in allowing us to rearrange this gradient expression in a sampleable form

$$p_\theta(\tau)\nabla_\theta \log p_\theta(\tau) = p_\theta(\tau)\frac{\nabla_\theta p_\theta(\tau)}{p_\theta(\tau)} = \nabla_\theta p_\theta(\tau) \tag{2.28}$$

Substituting this in we obtain a gradient term that appears amenable to sample-based estimation

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\nabla_\theta \log p_\theta(\tau) R(\tau)\right] \tag{2.29}$$

Carefully examining each term in this equation and noting the Markov assumption we make for our policy at each timestep[11], we show our gradient can be estimated with Monte Carlo methods via an

[11]That is, conditioned on current hidden state $\mathbf{h}_t$, our agent's policy is independent of past states and actions.

estimator which reduces to a function of the policy directly, of form

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{h}_{i,t}) \right) \left(\sum_{t=1}^{T} r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \right) \tag{2.30}$$

where in our partially observed paradigm the policy is conditioned on network hidden state $\mathbf{h}_t$, capturing current observation $\mathbf{o}_t$ in addition to any other relevant past information from the trial history $H_t = (\mathbf{o}_{1:t}, \mathbf{a}_{1:t-1}, r_{1:t-1})$. The reward itself remains a function of the full unobserved state. This gradient term can then be calculated at code level by performing backpropagation on the *pseudo-objective*[12] given by this expression without the ∇_θ symbol. A full derivation of this formula is given in Appendix B.3.

This expression is an unbiased estimator for the gradients we require, representing a sample-based maximum likelihood gradient weighted by the reward obtained across a given trajectory. In practise, the significant variance of this estimator makes it infeasible to implement for reasonable N, and so we make two key modifications to reduce this:

- **Causality:** Noting that actions taken at time t' cannot affect rewards obtained at $t < t'$, we modify our reward summation to take this into account thereby removing unnecessary variance-inducing terms. Expanding the first set of brackets and removing acausal reward terms, we obtain

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{h}_{i,t}) \left(\sum_{t'=t}^{T} r(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'}) \right) \tag{2.31}$$

  where cumulative reward from time t' is referred to as *reward to go*, often denoted $\hat{Q}(\mathbf{s}_{i,t}, \mathbf{a}_{i,t})$. The resulting estimator is identical in expectation, with lower variance based on the terms removed.

- **Baselines:** It can be shown subtracting a constant from the reward does not change our estimator in expectation, but can reduce variance [90]. We also recognise that when weighting our maximum likelihood gradient term by the sum of rewards this quantity should intuitively be centred for stability, balancing increases and decreases in sampled trajectory probability based on their relative success. The modified estimator for some *baseline* b is then

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{h}_{i,t}) \left(\sum_{t'=t}^{T} (r(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'}) - b) \right) \tag{2.32}$$

  where a common choice for such a baseline is average reward $b = \frac{1}{NT} \sum_{i=1}^{N} \sum_{t=1}^{T} r_{i,t}$

Use of such an estimator to iteratively sample trajectories, estimate the gradient term, and update parameters is referred to as the REINFORCE algorithm [91, 92]. We now briefly detail some advanced variants of policy gradient algorithms as implemented within this work.

2.3.3 Advantage Actor Critic

Advantage Actor Critic (A2C) is a policy gradient method that introduces a modified gradient weighting term to further improve performance, via the concept of an *actor-critic* architecture [93]. Noting our baseline can be any function that doesn't depend on action - and is hence free to vary with state without introducing bias - we first return to the fully observed case and consider a maximum likelihood gradient

[12] This pseudo-objective, typically denoted $\tilde{J}(\theta)$, is so called to distinguish it from our original, theoretical RL objective $J(\theta)$ in eq. (2.24).

weighting term at time t of form

$$\sum_{t'=t}^{T} r(\mathbf{s}_{i,t'}, \mathbf{a}_{i,t'}) - B(\mathbf{s}'_t) \tag{2.33}$$

where $B(\mathbf{s}_t)$ is a function of the state that is chosen to reduce the variance of our estimator. A common choice for this is the *state-value function* $V(\mathbf{s}_t)$[13]

$$V(\mathbf{s}_t) = \sum_{t'=t}^{T} \mathbb{E}_{\mathbf{a}_t \sim \pi_\theta(\mathbf{a}_t|\mathbf{s}_t)} \left[r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) | \mathbf{s}_t, \mathbf{a}_t \right] \tag{2.34}$$

which represents the expected future reward ('value') associated solely with the state $\mathbf{s}_t$, intuitively representing a strong choice for our baseline. We then further introduce the concept of an *advantage function*, defined as

$$A(\mathbf{s}_t, \mathbf{a}_t) = Q(\mathbf{s}_t, \mathbf{a}_t) - V(\mathbf{s}_t) \tag{2.35}$$

where $Q(\mathbf{s}_t, \mathbf{a}_t)$ is the *state-action value function*, defined as

$$Q(\mathbf{s}_t, \mathbf{a}_t) = \mathbb{E}_{\pi_\theta} \left[\hat{Q}(\mathbf{s}_t, \mathbf{a}_t) \right] = \sum_{t'=t}^{T} \mathbb{E}_{\mathbf{s}_t, \mathbf{a}_t \sim \pi_\theta(\cdot)} \left[r(\mathbf{s}_{t'}, \mathbf{a}_{t'}) | \mathbf{s}_t, \mathbf{a}_t \right] \tag{2.36}$$

This is effectively an idealised version of our previous single sample 'reward-to-go' estimate $\hat{Q}(\mathbf{s}_t, \mathbf{a}_t)$, representing the expected future reward associated with a specific *action* $\mathbf{a}_t$ taken in state $\mathbf{s}_t$. Note that both the value and state-value functions, and hence the advantage function, are idealised quantities which are not directly obtainable by the agent - the advantage function is therefore a *theoretical* quantity which describes how much better an action taken in a given state is, in expectation, than the average action taken in that state. This represents a strong choice to replace our previous gradient weighting term, given its intuitive measure of action quality and the low variance associated with the expectation. Implementing this advantage, the corresponding gradient estimator takes the form

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t}) A(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \tag{2.37}$$

To use an estimator of this form in practise, we must estimate the advantage across sampled trajectories. This is typically performed via a combination of sample-based estimation and functional approximation of the state value function $\hat{V}(\mathbf{s}_t)$. To obtain a sample-based expression for the advantage, we first consider that state-value and value functions are related as

$$Q(\mathbf{s}_t, \mathbf{a}_t) = r(\mathbf{s}_t, \mathbf{a}_t) + \mathbb{E}_{\mathbf{s}_{t+1} \sim p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)} \left[V(\mathbf{s}_{t+1}) \right] \tag{2.38}$$

and therefore make the approximation

$$Q(\mathbf{s}_t, \mathbf{a}_t) \approx r(\mathbf{s}_t, \mathbf{a}_t) + V(\mathbf{s}_{t+1}) \tag{2.39}$$

Considering a functional approximation of the state value $\hat{V}(\mathbf{s}_t)$, we similarly obtain a single sample estimate for the state-action value function

$$\hat{Q}(\mathbf{s}_t, \mathbf{a}_t) \approx r(\mathbf{s}_t, \mathbf{a}_t) + \hat{V}(\mathbf{s}_{t+1}) \tag{2.40}$$

[13]Note expectations in the following equations are considered over the agent's *policy*, emphasising the fact these quantities are defined by considering all possible actions sampled from the agent's current policy, as opposed to considering all possible trajectories.

and can finally estimate our advantage as

$$\hat{A}(\mathbf{s}_t, \mathbf{a}_t) \approx r(\mathbf{s}_t, \mathbf{a}_t) + \hat{V}(\mathbf{s}_{t+1}) - \hat{V}(\mathbf{s}_t) \tag{2.41}$$

Note that so far, we have considered the fully observed case for simplicity. In our partially observed case the state-value approximation is now parameterised by hidden state $\mathbf{h}_t$, and we have

$$\hat{A}(\mathbf{s}_t, \mathbf{h}_t, \mathbf{a}_t) \approx r(\mathbf{s}_t, \mathbf{a}_t) + \hat{V}(\mathbf{h}_{t+1}) - \hat{V}(\mathbf{h}_t) \tag{2.42}$$

where reward received remains a deterministic function of the unobserved state $\mathbf{s}_t$, but our state value estimate is obtained as a function of our agent's network hidden state $\mathbf{h}_t$.

Such an estimation of the advantage is an example of the Advantage Actor-Critic (A2C) policy gradient method, where the *actor* is our policy $\pi(\mathbf{a}_t|\mathbf{h}_t)$ which takes actions in the environment to maximise reward, and the *critic* $\hat{V}(\mathbf{h}_t)$ then evaluates the quality of actions taken via the advantage term $\hat{A}(\mathbf{s}_t, \mathbf{h}_t, \mathbf{a}_t)$, rewarding decisions that had better outcomes than expected under the current policy by incorporating elements of temporal difference learning. Reducing the number of sampled rewards required to obtain our advantage estimate via function approximation is known as *bootstrapping*. Such methods benefit from reduced variance, with the caveat of the introduction of some bias based on the accuracy of $\hat{V}(\mathbf{h}_t)$. This tradeoff between bias and variance is governed by the number of samples used in our bootstrapped estimate, where a general (n-step bootstrapped) form of this function approximation is known as a *generalised advantage function* (GAE), as clarified in the following section.

Finally, we implement an additional loss to optimise our value estimation function[14] as

$$\mathcal{L}_{\text{critic}}(\theta) = \mathbb{E}_{\mathbf{s}_t, \mathbf{a}_t \sim \pi_\theta(\mathbf{a}_t|\mathbf{s}_t)} \left[(r(\mathbf{s}_t, \mathbf{a}_t) + \hat{V}(\mathbf{h}_{t+1}) - \hat{V}(\mathbf{h}_t))^2 \right] \tag{2.43}$$

which takes a similar bootstrapped form. This can be considered a *self-supervised* loss which we *minimise* alongside our primary (actor) objective.

2.3.4 Proximal Policy Optimization

For training the RL agents in this work we use a modified version of the A2C method known as Proximal Policy Optimization (PPO) [94]. The key improvements of PPO over A2C are largely twofold, improving both the stability and efficiency of our policy updates. To explain this we must first introduce the concept of importance sampling. Considering the ratio between some previous policy $\pi_{\theta_{\text{old}}}$ and the current policy π_θ, we define

$$l(\theta)_{i,t} = \frac{\pi_\theta(\mathbf{a}_{i,t}|\mathbf{h}_{i,t})}{\pi_{\theta_{\text{old}}}(\mathbf{a}_{i,t}|\mathbf{h}_{i,t})} \tag{2.44}$$

This ratio is typically considered within a process known as importance sampling, where we have some *off-policy* data from another policy (here an older version $\pi_{\theta_{\text{old}}}$) which we wish to use to improve our current policy. We accomplish this by scaling our update under the old policy by $l(\theta)$[15]

$$\nabla_\theta J_{\text{IS}}(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} l(\theta)_{i,t} \nabla_\theta \log \pi_{\theta_{\text{old}}}(\mathbf{a}_{i,t}|\mathbf{h}_{i,t}) \hat{A}(\mathbf{s}_{i,t}, \mathbf{h}_{i,t}, \mathbf{a}_{i,t}) \tag{2.45}$$

[14] In this case, our critic loss is equivalent to the square of the advantages.

[15] Note this estimator is no longer unbiased, as we consider only the importance weight of the current timestep (as opposed to the product of weights up to that timestep). This makes the implicit assumption the marginal probability of the current state is equal under the old and new policies, which is done to avoid the issue of importance weights becoming degenerate exponentially in T.

Such off-policy learning has obvious benefits for sample efficiency but importantly also gives a measure of policy update stability, where ratios deviating from 1 indicate increasingly unstable updates. The main modification of PPO is therefore to implement this ratio as part of its 'clipped surrogate objective', of form

$$J_{\mathrm{PPO}}(\theta) = \mathbb{E}_{\tau\sim\pi_\theta(\tau)}\left[\min\left(l(\theta)\hat{A}(\tau), \mathrm{clip}\left(l(\theta), 1-\epsilon, 1+\epsilon\right)\hat{A}(\tau)\right)\right] \tag{2.46}$$

Such an objective introduces a clipping parameter ϵ[16] which is applied element-wise to the probability ratio. This prevents large, potentially unstable updates by enforcing the updated policy remains close to the previous policy, effectively introducing a form of *trust region*[17]. While at first glance our log probability term has 'disappeared' it remains implicit in the ratio $l(\theta)$, which can be considered the exponentiated difference between log probabilities under both policies $l(\theta) = \exp(\log \pi_\theta - \log \pi_{\theta_{\mathrm{old}}})$. By explicitly incorporating off-policy information in our objective term such an algorithm also benefits from improved sample efficiency, enabling multiple mini-batch updates to our objective for each sampled batch of trajectories. We also now consider an alternative form of advantage function, termed the *generalized advantage estimate* [96], of form

$$\hat{A}_t^{\mathrm{GAE}} = \sum_{t=0}^{T-1}(\gamma\lambda)^t\delta_t, \quad \delta_t = r_t + \gamma V(\mathbf{h}_{t+1}) - V(\mathbf{h}_t) \tag{2.47}$$

where δ_t is the *temporal difference error* at time t. We therefore effectively extend our single sample bootstrapped estimate to now also factor in errors at subsequent timesteps via the decay parameter λ, the tuning of which governs an implicit tradeoff between bias and variance[18]. The verbose form of the PPO gradient estimator implemented is then

$$\nabla_\theta J_{\mathrm{PPO}}(\theta) \approx \frac{1}{B}\sum_{i=1}^{B}\sum_{t=1}^{T}\nabla_\theta \min\left(l(\theta)_{i,t}\hat{A}_{i,t}^{\mathrm{GAE}}, \mathrm{clip}\left(l(\theta)_{i,t}, 1-\epsilon, 1+\epsilon\right)\hat{A}_{i,t}^{\mathrm{GAE}}\right) \tag{2.48}$$

and the critic loss takes a similar bootstrapped form as before. A full description of the PPO algorithm is given below in Algorithm 2. In summary, PPO leverages the reduced variance of an actor-critic architecture with the increased update stability provided by the clipping parameter and improved sample efficiency enabled by off-policy sampling, producing strong performance empirically [97, 98].

[16] This is typically kept low; a common choice is $\epsilon = 0.2$.

[17] The term 'trust region' typically refers to an explicit constraint on policy update magnitude as governed by a KL divergence penalty, initially popularised by the Trust Region Policy Optimization (TRPO) method [95]. The clipping parameter in PPO performs a similar role, in a less involved way.

[18] For $\lambda = 0$ GAE reduces to our previous single sample estimate $\hat{V}$ (high bias, low variance), while for $\lambda = 1$ we considers estimates based on the full trajectory (low bias, high variance).

Algorithm 2: Proximal Policy Optimization (PPO) for POMDP

input : Initial policy parameters θ_0, initial hidden states $\mathbf{h}_0$, initial environment states $\mathbf{s}_0$

parameters: Policy network parameters θ, hyperparameters (clip parameter ϵ, number of epochs E, optimisation steps per epoch O, length of trial T, trajectory batch size N, mini-batch size B, learning rate η, etc.)

$\theta \leftarrow \theta_0$ $\mathbf{h} \leftarrow \mathbf{h}_0$ % Initialise policy network parameters and hidden states

while $e < E$ ***or*** *not converged* **do**

  % Collect data for N trials, length T

  for $i = 1 : N$ **do**

    for $t = 1 : T$ **do**

      % Actor step

      $\mathbf{h}_{i,t} \leftarrow \text{NetworkUpdate}(\mathbf{h}_{i,t-1}, \mathbf{o}_{i,t-1}, \mathbf{a}_{i,t-1}, r_{i,t-1}, ; \theta)$ % Update network hidden state

      $\mathbf{a}_{i,t}, \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{h}_{i,t}) \leftarrow \text{SamplePolicy}(\mathbf{h}_{i,t}; \theta)$ % Sample action from policy

      $\mathbf{o}_{i,t} \leftarrow \text{StateObservation}(\mathbf{s}_{i,t}, \mathbf{a}_{i,t})$ % Obtain state observation

      % Critic step

      $\hat{V}_{i,t} \leftarrow \text{ValueReadout}(\mathbf{h}_{i,t}; \theta)$ % Estimate state value

      $r_{i,t} \leftarrow \text{RewardFunction}(\mathbf{s}_{i,t}, \mathbf{a}_{i,t})$ % Obtain reward

      $\delta_{i,t-1} \leftarrow \text{TDError}(r_{i,t-1}, V_{i,t-1}, V_{i,t})$ % Compute temporal difference error

    $\hat{A}_{i,t} \leftarrow \text{ComputeGAE}(\delta_{i,t}, \delta_{i,t+1}, ..., \delta_{i,T}, \gamma, \lambda)$ % Compute generalised advantage estimate

  % Update policy and value networks

  for $j = 1 : O$ **do**

    % Compute PPO policy gradient estimator

    $\theta_{\text{old}} \leftarrow \theta$ % Set 'old' policy weights to current policy weights

    $l(\theta)_{i,t} \leftarrow \exp(\log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{h}_{i,t}) - \log \pi_{\theta_{\text{old}}}(\mathbf{a}_{i,t}|\mathbf{h}_{i,t}))$ % Compute probability ratio

    $\text{SampleMiniBatch}(B, M)$ % sample B trajectories for mini-batch update

    $\nabla_\theta J_{\text{PPO}}(\theta) \approx \frac{1}{B} \sum_{i=1}^{B} \sum_{t=1}^{T} \nabla_\theta \min \left(l(\theta)_{i,t} \hat{A}_{i,t}, \text{clip}\left(l(\theta)_{i,t}, 1-\epsilon, 1+\epsilon\right) \hat{A}_{i,t} \right)$

    $\theta \leftarrow \text{PolicyUpdate}(\theta, \nabla_\theta J_{\text{PPO}}(\theta), \eta)$ % Update both actor and critic portions of network with mini-batch gradient

return θ^*% Optimised network parameters

2.4 Simulation Environment

2.4.1 Simulation Space

The simulation space employed in this work is consistent across experiments and consists of a two dimensional space that is periodic across both axes. Such a space is akin to the surface of a torus[19] and is chosen to reflect the periodicity of egocentric space from our agent's perspective. Despite the caveat that we lack the specific curvature - such as the notion of poles - of 'true' egocentric space represented by a sphere, a torus nonetheless describes a periodic, egocentric representation of the environment with which to investigate the angular motor behaviour of interest in this work, while simplifying the challenges of working in spherical space[20]. Points on this space are fully defined by two angles (ϕ, θ)

[19]Specifically we describe a *horn torus*, a type of torus with equal minor and major radii.

[20]For example due to discontinuities at the poles and difficulties visualising a 2D projection of this space.

lying on the unit circle S^1, hence we denote our toroidal space via a cartesian product as

$$\mathbb{T}^2 = S^1 \times S^1 = \{(\phi, \theta) \mid 0 \leq \phi, \theta < 2\pi\} \tag{2.49}$$

where θ and ϕ represent longitudinal and latitudinal rotational coordinates respectively. In practice these angles are unconstrained at code level in our simulations, with the periodic constraint applied via carefully chosen periodic objectives and basis functions[21]. We denote such freely varying points $(x, y) \in \mathbb{R}^2$, and define a mapping $f : \mathbb{R}^2 \to \mathbb{T}^2$ by considering these points as members of an equivalence class $[x, y]_\sim$ defined under the relation

$$(x, y) \sim (x + 2\pi n, y + 2\pi m), \quad \forall n, m \in \mathbb{Z} \tag{2.50}$$

Our code level 'angles' therefore live in Euclidean space but are mapped onto the torus as

$$\mathbb{T}^2 = \{[x, y]_\sim \mid x, y \in \mathbb{R}^2\}, \quad \text{i.e.} \quad f : \begin{cases} x \mapsto \phi = x \pmod{2\pi} \\ y \mapsto \theta = y \pmod{2\pi} \end{cases} \tag{2.51}$$

Simulations in this work consider an agent's angular rotations in this space - representing movement on the torus - and interactions with N_x point-defined objects of varying reward. Given our simulations are egocentric in nature the most meaningful quantities are *relative* positions of each object with respect to the agent, however for simplicity at code level we consider static objects $\mathbf{x}_0 = (x_0, y_0)$ and time varying agent position $\mathbf{p}_t = (p_{t,x}, p_{t,y})$, where relative object positions are computed as necessary as $\mathbf{x}_t = \mathbf{x}_0 - \mathbf{p}_t$. Figure 2.4 depicts these quantities in both a toroidal and projected view of our simulation space.

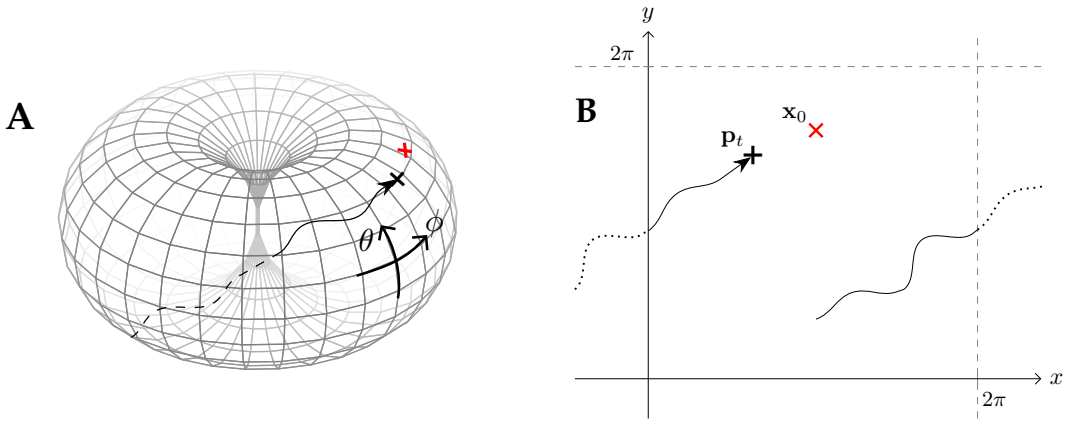


Figure 2.4: **Simulation space. (A)** Toroidal visualisation. In this space our agent's gaze position $\mathbf{p}_t$ (black plus) and single object position $\mathbf{x}_0$ (red cross) are defined by periodic angles (ϕ, θ) that lie on the surface of a torus (the ϕ axis swings around the torus and θ axis - intersecting the major radii, dotted - represents rotation within the minor radii). At code level however we represent these positions with coordinates (x, y) which are mapped to (ϕ, θ) as in eq. (2.50), shown on the Cartesian plane in B. **(B)** Cartesian visualisation. We define points on this plane now with unconstrained Euclidean coordinates (x, y), each associated with a unique position within the dashed lines as described by the equivalence relation in eq. (2.48). The domain within this range could hence be considered an 'unwrapped' version of the angular space in (A), and will be used throughout this work to visualise agent behaviour. Note for simplicity in our simulations we consider 'static' objects $\mathbf{x}_0$ with fixed positions in space, and time varying agent position $\mathbf{p}_t$. Environmental state and agent behaviour are hence defined by the relative positions of each object $\mathbf{x}_t^{(j)} = \mathbf{x}_0^{(j)} - \mathbf{p}_t$, as explored in Section 2.4.2.

This work considers RL agents trained to solve tasks involving navigating this space: taking some action

[21]Enforcing such a constraint is also vital to ensure our space has the correct degree of degeneracy - that all (x, y) points map to a single (ϕ, θ) point in $\mathbb{T}^2$.

$\mathbf{a}_t$ in state $\mathbf{s}_t$ to update position $\mathbf{p}_t$ and hence obtain reward r_t, as denoted in our *RL loop* from Figure 2.3. The following section considers the functions which describe how state observation $\mathbf{o}_t$ and reward r_t are calculated in a given state.

2.4.2 Basis and Objective Functions

One of the main aims of the RL simulations in this work is to investigate the transformation of visual information to motor commands within the SC. We must therefore implement a state-observation mapping in our agents that is reflective of the visual information received at SC-level. Given a significant proportion of the SC's visual information consists of direct retinal projections [5], one simple choice for this state-observation mapping is to tile the agent's egocentric space with basis functions to represent these retinal activations (or more formally, *retinal ganglion cells*). This array is centred at instantaneous agent location $\mathbf{p}_t$ and extends for some range in both x and y directions - where a range of $\pm\pi$ would represent 'full' scope of the space, and a range of $\pm\pi/2$ would hence reflect a 25% observed environment at any time. To assign each retinal activation a scalar value based on the relative distance of each object $\mathbf{x}_t^{(j)}$, a simple choice for such a function is a bivariate von Mises distribution, the circular analogue of a bivariate Gaussian[22]. We define our basis functions via a modified form of this function, adding a small amount of noise to reflect imperfect environmental sampling

$$o_t^{(i)} = f^{(i)}(\mathbf{s}_t|\boldsymbol{\mu}^{(i)}, \kappa) = \sum_{j=1}^{N_x} e^{\kappa\left(\sum\left[\cos(\mathbf{x}_t^{(j)} - \boldsymbol{\mu}^{(i)})\right] - 2\right)} + \epsilon, \quad \epsilon \sim \mathcal{N}\left(0, \sigma^2_{\text{sampling}}\right) \tag{2.52}$$

where $o_t^{(i)}$ is the scalar activation seen by neuron i at time t, summing over contributions from each object $\mathbf{x}_t^{(j)}$. Each of the N_o basis functions is centred with a $\boldsymbol{\mu}^{(i)}$ vector relative to instantaneous agent location, and κ is a scaling constant determining basis function width. Input state $\mathbf{s}_t$ is a set of all relevant environmental information at time t, in this case relative object locations $\mathbf{x}_t^{(j)}$. The sum in the exponent denotes the element-wise sum of the $\cos$ term, i.e. $\sum \cos(\mathbf{x}) = \cos(x) + \cos(y)$. We omit the normalising Bessel function term from the standard von Mises distribution for computational simplicity[23], given our basis functions are not required to conform to the normalisation constraints of probability distributions, and introduce a -2 term in the exponent to bound our pre-noise activations $0 < f \leq 1$[24].

This representation implies a greyscale interpretation of the environment, which does not distinguish between objects. Such a distinction is made in some of our simulations by specifying object colour with RGB tuples, similarly contained within $\mathbf{s}$. To enable our agent to differentiate between such objects we therefore introduce colour channels to our objective by multiplying the activations for each $\mathbf{x}^{(j)}$ by a corresponding normalised vector $\mathbf{c}^{(j)} = (r^{(j)}, g^{(j)}, b^{(j)})$ such that each observation is now a vector $\mathbf{o}^{(i)} \in \mathbb{R}^3$

$$\mathbf{o}_t^{(i)} = [o_{t,r}^{(i)}, o_{t,g}^{(i)}, o_{t,b}^{(i)}] = f_{\text{rgb}}^{(i)}(\mathbf{s}_t|\boldsymbol{\mu}^{(i)}, \kappa) = \sum_{j=1}^{N_x} \left(e^{\kappa\left(\sum\left[\cos(\mathbf{x}_t^{(j)} - \boldsymbol{\mu}^{(i)})\right] - 2\right)} \right) \mathbf{c}^{(j)} + \boldsymbol{\epsilon} \tag{2.53}$$

In total our agent then receives $3N_o$ inputs across 3 independent RGB channels, each considering

[22]Such a distribution can be approximated as Gaussian for small values of $\mathbf{x} - \boldsymbol{\mu}$, effectively considering a distance measure based on the Euclidean norm of angular difference. This is a more appropriate angular distance measure than geodesic distance on the torus $\mathbb{T}^2$, which features distortion between the inner and outer surfaces and therefore non-uniform geodesics for equivalent angular differences. Calculating this quantity is also more involved, and can introduce instability.

[23]Not only are Bessel functions expensive to calculate, their derivatives cannot be expressed with elementary functions which induces problems during backpropagation.

[24]Given each basis function responds only to small angles, we can approximate our exponents as $\cos(x-\mu_x)+\cos(y-\mu_y)-2 \approx 1 - \frac{(x-\mu_x)^2}{2} + 1 - \frac{(y-\mu_y)^2}{2} - 2 = -\frac{(x-\mu_x)^2+(y-\mu_y)^2}{2}$, where we note the Gaussian analogy and observe the effect of the -2 term to elegantly bound pre-noise activations below 1.

contributions from $N_x = 3$ objects. Concatenating all state observations together, our agent is fed at each timestep with the vector of observations

$$\mathbf{o}_t = [o_{t,r}^{(1)}, o_{t,g}^{(1)}, o_{t,b}^{(1)}, ..., o_{t,r}^{(N_o)}, o_{t,g}^{(N_o)}, o_{t,b}^{(N_o)}] \tag{2.54}$$

State reward at each timestep is similarly obtained from a modified von Mises function

$$r_t = r(\mathbf{s}_t|\kappa_r) = \sum_{j=1}^{N_x} r^{(j)} e^{\kappa_r \left(\sum\left[\cos(\mathbf{x}_t^{(j)})\right]-2\right)} \tag{2.55}$$

where we sum over object contributions based on both their relative location $\mathbf{x}_t^{(j)}$ and reward $r^{(j)}$.

With respect to the classical RL paradigm described in Section 2.3.1 consisting of states, observations, actions and rewards, we can therefore briefly define each of these quantities as considered in this work:

- **State**: A vector $\mathbf{s}_t$ comprising concatenated state information, including relative object locations with respect to agent position $\mathbf{x}_t^{(j)}$, their associated scalar rewards $r^{(j)}$, and their respective colours $\mathbf{c}^{(j)}$ (in some simulations)
- **Observation**: The vector $\mathbf{o}_t$ of all concatenated scalar visual basis function inputs our agent receives from the current state
- **Action**: The angular movement vector $\mathbf{a}_t = (a_{t,x}, a_{t,y})$ (or planned movement vector, in some simulations) chosen as a function of state observation and hidden network state
- **Reward**: The scalar value $0 \leq r_t \leq 1$ received from the current state based on proximity to the rewarded object

2.5 Network Analysis

Alongside statistical analysis of our trained agent, we also directly analyse the network dynamics governing this behaviour. For this purpose we use a popular method known as Principal Component Analysis (PCA), including a variant known as Demixed PCA (dPCA) [99, 100].

2.5.1 Principal Component Analysis

Principal Component Analysis (PCA) is a technique used to visualise high dimensional neural data by projecting it into a lower dimensional subspace. To preserve as much information as possible in this low dimensional projection, we chose orthogonal projection bases (principal components) that capture maximal variance in the high dimensional data. For an array of neural timeseries data $\mathbf{X}_{\text{data}} \in \mathbb{R}^{N\times D}$, where N and D denote number of observations and neuron count respectively, PCA projects our data into the space of the top K principal components as $f : \mathbb{R}^{N\times D} \to \mathbb{R}^{N\times K}$, where each PC can be considered to reflect some normalised combination of the D original neuron features[25]. Analysis can then be performed directly in the low dimensional embedding space, where we can visualise and interpret our data. We may for example observe dynamical motifs of interest - such as attractor dynamics [101–103], limit cycles [100, 104], or task event-triggered correlations [105, 106] - which can provide insight into the population level activity of our network.

The classical approach to such an eigenproblem considers first computing the sample covariance matrix

[25]I.e. the length N vectors describing a single neuron's activity across all observations.

of the column-centred data, henceforth denoted $\mathbf{X}$

$$\mathbf{C} = \frac{1}{N-1}\mathbf{X}^\top\mathbf{X}, \quad \mathbf{C} \in \mathbb{R}^{D\times D} \tag{2.56}$$

then the corresponding eigendecomposition, sorting by descending eigenvalues

$$\mathbf{C} = \mathbf{Q}\mathbf{\Lambda}\mathbf{Q}^\top, \quad \mathbf{\Lambda} = \text{diag}\,(\lambda_1, \lambda_2, ..., \lambda_D) \text{ where } \lambda_1 \geq \lambda_2 \geq ... \geq \lambda_D \tag{2.57}$$

We then project $\mathbf{X}$ into the subspace described by the top K eigenvectors, contained in $\mathbf{Q}_K$

$$\mathbf{Z} = \mathbf{X}\mathbf{Q}_K, \quad \mathbf{Z} \in \mathbb{R}^{N\times K} \tag{2.58}$$

where $\mathbf{Z}$ is then our dimension-reduced data. However for large neural populations the $\mathcal{O}(D^3)$ cost of eigendecomposition can be expensive. Therefore for cases where $D > N$ a comparatively cheaper $\mathcal{O}(ND^2)$[26] Singular Value Decomposition (SVD) based method is favoured. Such a method directly computes the singular vectors of interest

$$\mathbf{X} = \mathbf{U}\mathbf{\Sigma}\mathbf{V}^\top, \quad \sigma_i = \sqrt{(N-1)\lambda_i} \tag{2.59}$$

with the PC projection then obtained via the top K right singular vectors

$$\mathbf{Z} = \mathbf{X}\mathbf{V}_K \tag{2.60}$$

While less immediately intuitive, such a method is often preferred due to the lower compute cost and improved numerical stability of the algorithms that implement it [107]. There is no need, for example, for the intermediate step of computing the covariance matrix (a numerically unstable operation in the case of ill-conditioned data).

2.5.2 Demixed Principal Component Analysis

A significant drawback with the above approach is that PCs are chosen agnostic to task parameters of interest - all N observations are treated equally, and as such the resultant PCs cannot reasonably be interpreted with respect to task parameters. In other words, the top K PCs explain variance that is *mixed* across task parameters, as well as capturing uninteresting quantities such as noise. Demixed Principal Component Analysis (dPCA) seeks to improve upon this confound by computing a data projection that explains as much variance as possible while also ensuring the dimensions (PCs) of our projection capture variance induced by a single task parameter, enabling us to intrepret neural activity with respect to these parameters [99, 106]. Such a method is inherently supervised, requiring labelled data and a degree of hyperparameter fitting via cross-validation.

To explain dPCA, we first consider an example task parameter set $S = \{t, s, d\}$, denoting time t, stimulus s and decision d respectively. We start by labelling each point in our dataset using these parameters, then average over each parameter combination to get a single datapoint to form a new data matrix, typically stacked into a vector as $\mathbf{X} \in \mathbb{R}^{TSD}$, with length described by the total number of timesteps, stimuli, and decision values respectively. We then decompose this data into a sum of *marginalised* datasets, each dependent solely on a single set of parameters $\phi \subseteq S$, where importantly we dictate that our original data should be reconstructed as $\mathbf{X} = \sum_\phi \mathbf{X}_\phi$. For a single parameter, such a marginalised average is

[26]The total time cost of covariance-based SVD is $\mathcal{O}(ND^2 + D^3)$ (for the covariance calculation and eigendecomposition respectively), whereas only $\mathcal{O}(ND^2)$ for SVD. Space complexity is comparable in both methods at $\mathcal{O}(D^2)$ for covariance-based and $\mathcal{O}(ND + D^2)$ for SVD-based.

intuitively computed by averaging over all other parameters, e.g.

$$\mathbf{X}_t = \langle \mathbf{X} \rangle_{sd} \tag{2.61}$$

This procedure can be generalised to form marginalised datasets of an arbitrary number of parameters, via a slightly more complex process as detailed in Appendix B.3. For $M = |S|$ task parameters there hence exists 2^M possible marginalisation terms, however we typically consider a simpler decomposition with fewer terms. For example a common decomposition for the example parameters given is

$$\mathbf{X} = \mathbf{X}_t + \mathbf{X}_{st} + \mathbf{X}_{dt} + \mathbf{X}_{sdt} = \sum_\phi \mathbf{X}_\phi \tag{2.62}$$

where stimulus and decision terms are combined with their respective stimulus-time and decision-time interactions terms[27]. After decomposing our data in this fashion, we can consider the algorithm used to find a set of demixed principal components for each marginalisation $\mathbf{X}_\phi$. We first note the standard PCA objective can be formulated as minimising the Frobenius norm[28] of the difference between our original and *reconstructed* data

$$\mathcal{L}_{\text{PCA}} = \|\mathbf{X} - \mathbf{FDX}\|^2, \quad \mathbf{X} \in \mathbb{R}^{N \times D},\ \mathbf{D} \in \mathbb{R}^{K \times N},\ \mathbf{F} \in \mathbb{R}^{N \times K} \tag{2.63}$$

where $\mathbf{D}$ is the *decoder* matrix which performs the PCA projection to our low dimensional subspace (termed the 'compression step'), $\mathbf{F}$ is the *encoder* matrix which then de-compresses our data, returning it to full dimensionality, and K is the number of principal components we seek for each marginalisation. Constraining our principal axes to be normalised and orthogonal we intuitively have $\mathbf{D} = \mathbf{F}^\top$[29].

For dPCA, we modify this loss to consider each marginalisation separately, where we now seek a set of K principal components for *each* marginalisation $\mathbf{X}_\phi$ that minimises reconstruction error for that ϕ

$$\mathcal{L}_{\text{dPCA}} = \sum_\phi \mathcal{L}_\phi = \sum_\phi \left(\|\mathbf{X}_\phi - \mathbf{F}_\phi \mathbf{D}_\phi \mathbf{X}\|^2 + \lambda \|\mathbf{D}_\phi\|^2 \right) \tag{2.64}$$

Each set of principal components $\mathbf{D}_\phi$ then reflects directions of variance attributed solely to ϕ whilst still capturing maximal variance in the original data[30]. An important change compared to our previous objective is that we relax the orthogonality constraint $\mathbf{D}^\top \mathbf{D} = \mathbf{I}$ imposed in PCA, to enable full demixing between marginalisations. In other words $\mathbf{D}_\phi^\top \mathbf{D}_{\phi'}$ is unconstrained for $\phi \neq \phi'$, and the previous relation $\mathbf{D} = \mathbf{F}^\top$ no longer holds (where $\mathbf{D}$ is the matrix formed from all $\mathbf{D}_\phi$). Since our data has been projected onto a non-orthogonal (affine) coordinate system, the optimal reconstruction requires de-compression along a different set of axes. A geometric intuition for this is given in Figure 2.5. The regularisation term λ is introduced to prevent overfitting, and is fitted via cross-validation (Appendix B.3). A brief overview of the algorithm used to find $\mathbf{D}_\phi$ is detailed below in the $\lambda = 0$ case for simplicity (Algorithm 3), where

[27] This is because stimulus and decision are inherently correlated with time. Therefore variation in the data due to each of these parameters alone, as well as due to their parameter-time interaction, can be intuitively pooled to a single term (i.e. $\mathbf{X}_d + \mathbf{X}_{td} \to \mathbf{X}_{td}$).

[28] The Frobenius norm is expressed as $\|\mathbf{X}\|^2 = \sum_i \sum_j x_{i,j}^2$.

[29] However note that $\mathbf{D}$ and $\mathbf{F}$ are not orthogonal matrices as they are not square. For a square, full rank matrix $\mathbf{D}_N$ (selecting all N PCs) we would fulfill this constraint, obtaining $\mathbf{F}_N = \mathbf{D}_N^{-1}$, intuitively performing perfect decompression and obtaining zero loss in eq. (2.61).

[30] This is in contrast to initial proposed methods for dPCA [99] which encouraged demixing via a sparsity penalty on the PCs chosen across all marginalisations, encouraging them to fall along marginalisation-specific axes in σ-space. While elegant, the PCs produced by such a method do not completely demix between task parameters.

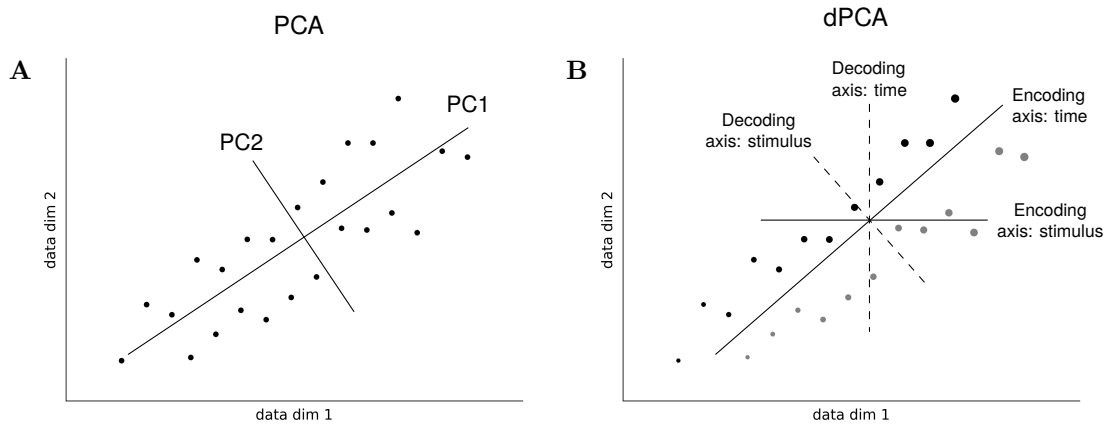


Figure 2.5: **Geometric intuition for PCA and dPCA.** PCA **(A)** chooses principal components which blindly capture maximum variance of our data agnostic to the labels of the data - including the underlying task parameters we wish to investigate - and as such each PC (PC1, PC2) contains variance that is *mixed* across parameters. Such principal components are contained in a decoder (compression) matrix $\mathbf{D}$ whose transpose $\mathbf{F} = \mathbf{D}^\top$ optimally encodes (decompresses) our data back to the input space. For the dPCA **(B)** approach however we now label our data - in this case with time (dot size) and some 'stimulus' task parameter (grey or black, representing a binary stimulus). We then consider separate decoder matrices $\mathbf{D}_\phi$ for each parameter combination ϕ (in this case time, stimulus, and a mixture of the two) that explain variance attributed solely to that parameter, with 1D examples (dotted lines) given for time and stimulus respectively. Each $\mathbf{D}_\phi$ is no longer orthogonal to each other however, and so the total appended matrix of projections $\mathbf{D}$ now represents an affine transform (i.e. introduces a shift component). Hence our encoder (optimal decompression) matrices $\mathbf{F}_\phi$ now represent a different set of axes (solid lines), and a more accurate relation is now $\mathbf{D} \approx \mathbf{F}^+$.

our problem reduces to that of reduced-rank regression, with objective for each marginalisation as

$$\mathbf{A}_\phi = \underset{\mathbf{A}}{\operatorname{argmin}} \|\mathbf{X}_\phi - \mathbf{A}\mathbf{X}\|^2, \quad \text{s.t} \quad \operatorname{rank}(\mathbf{A}) \leq K \tag{2.65}$$

where $\mathbf{A}_\phi = \mathbf{F}_\phi \mathbf{D}_\phi$, and as before K denotes the number of dPCs we seek for each marginalisation[31]. The $\lambda \neq 0$ case is described in Appendix B.3, where λ is first fit with cross-validation before a similar optimisation problem can be formed.

Algorithm 3: Demixed Principal Component Analysis (dPCA) for $\lambda = 0$

input : Original data matrix $\mathbf{X}$, set of marginalised data matrices $\mathbf{X}_\phi$, target rank K of dimension-reduced data

parameters: Regularisation term $\lambda = 0$

% Compute standard regression solution to the unconstrained problem $\mathcal{L} = \|\mathbf{X}_\phi - \mathbf{M}\mathbf{X}\|^2$

$\mathbf{M} \leftarrow \mathbf{X}_\phi \mathbf{X}^+$ % where $\mathbf{X}^+$ is the pseudoinverse computed with SVD, i.e. $\mathbf{X}^+ = \mathbf{V}\mathbf{\Sigma}^+\mathbf{U}^\top$

% Compute $\mathbf{U}_K$, the K-dimensional subspace of $\mathbf{M}\mathbf{X}$ that captures maximum variance

$\mathbf{U}, \mathbf{\Sigma}, \mathbf{V}^\top \leftarrow \mathrm{SVD}(\mathbf{M}\mathbf{X})$, $\mathbf{U}_K \leftarrow \mathbf{U}[:, 1:K]$ % K leading singular vectors

% Project $\mathbf{M}$ to K-space to incorporate rank constraint

$\mathbf{A}_\phi \leftarrow \mathbf{U}_K \mathbf{U}_K^\top \mathbf{M}$ % project to K-space but represent back in full-dimensional space

% Factorise $\mathbf{A}_\phi$ to recover decoder $\mathbf{D}_\phi$ and encoder $\mathbf{F}_\phi$

$\mathbf{D}_\phi \leftarrow \mathbf{U}_K^\top \mathbf{A}_\phi$, $\mathbf{F}_\phi \leftarrow \mathbf{U}_K$

return $\mathbf{D}_\phi$, $\mathbf{F}_\phi$ % decoder $\mathbf{D}_\phi$ (marginalised principal components), encoder $\mathbf{F}_\phi$

[31]Given $\operatorname{rank}(\mathbf{A}) \leq \min(\operatorname{rank}(\mathbf{D}_\phi), \operatorname{rank}(\mathbf{F}_\phi))$ - that is, the rank of a matrix product is always less than or equal to the rank of the multiplied matrices - we have that $\operatorname{rank}(\mathbf{A}) \leq K$.

Chapter 3

Simulations in Continuous Action Space

All vision-dependent animals recruit angular movements of the head and eyes to navigate the egocentric world, with the superior colliculus believed to play a crucial role in the ballistic eye movements - saccades - involved in this process [108]. While macroscopically rapid, these movements are smooth at finer timescales, and are widely believed to be governed by some form of motor control feedback loop [50, 109, 110]. Competing theories exist, however, regarding the exact form of feedback loop involved. While some models suggest SC activity may be statically 'read out' then enacted by a separate downstream motor loop [8, 47], other models consider the SC may dynamically modulate elements of this loop [44, 48], or even situate the SC directly inside it [57, 58] (Appendix A). Given our goal of investigating SC dynamics using RL tasks in egocentric space, a simple paradigm to begin with are tasks loosely inspired by such models where our agent must perform smooth, continual movements through space, now governed by an *RL loop*. Such a paradigm - where our agent employs a *continuous action space* - has many parallels to these models, albeit now incorporating the increased complexity necessary for decision making in RL. This paradigm also benefits from a simplified training process due to the end to end differentiability of our continuous closed loop setting. Under this paradigm, we train an agent to perform angular navigation tasks between objects of interest, subsequently analysing the behaviours developed such that we can comment in more detail on any ties to ethological SC dynamics.

3.1 Task Overview

The task in this chapter considers an agent navigating our egocentric angular space via smooth, continuous movement with the goal of identifying the location of a single rewarded object. Environment state is defined by the locations of 3 randomly placed point objects represented as dots of different colours, only one of which is rewarded via unit scalar reward. The agent's goal in each trial is then, given visual input from its basis functions as described in Section 2.4, to produce a continuously varying output vector to navigate space and discover the rewarded object, subsequently remaining in that location to maximise reward obtained. Figure 3.1 graphically depicts this task and denotes the variables involved.

After training our agent to optimally perform this task - termed the *navigation task* - we then test its ability to generalise this behaviour under a meta-learning paradigm, investigating if the basic navigation skills obtained can extend to more complex scenarios with a small degree of further training. The subsequent task is identical except for the fact the agent is now teleported to a random position in space upon successfully reaching the rewarded object, where it must then successively renavigate back to the previously rewarded location to continue to obtain reward in each trial. This task is termed the

renavigation task, and enables us to test our agent's ability to generalise its reward seeking behaviour to a new contexts by leveraging internal notions of egocentric space.

Post meta-training we then analyse our adapted agent with the goal to provide insight into the following questions, drawing appropriate links to the superior colliculus literature in Section 1.2:

- What behaviours does our agent learn to solve reward-seeking tasks in periodic angular space?
- Can our agent generalise simple navigation behaviours in periodic angular space to more complex renavigation behaviours, by utilising notions of rewarded location in egocentric space?

3.2 Model Overview

The model used in this task is a GRU with $H = 80$ hidden units - a powerful recurrent network enabling our agent to learn the long sequences of dependencies required in this task. Figure 3.1A displays a schematic of this model, drawing parallels to the more general RL loop of Figure 2.3 and the main task variables of state $\mathbf{s}_t$, observation $\mathbf{o}_t$, action $\mathbf{a}_t$ and reward r_t. The input observation $\mathbf{o}_t$ to our network consists of 100 RGB visual basis functions uniformly distributed in a 10×10 grid, tiling egocentric space at equal intervals to form an RGB input vector of length 300 as in eq. (2.49). The choice for uniform coverage is motivated by our task objectives to examine egocentric angular navigation behaviour and notions of relative reward location, which do not explicitly dictate a need for ethological realism, such as a limited observation scope[1].

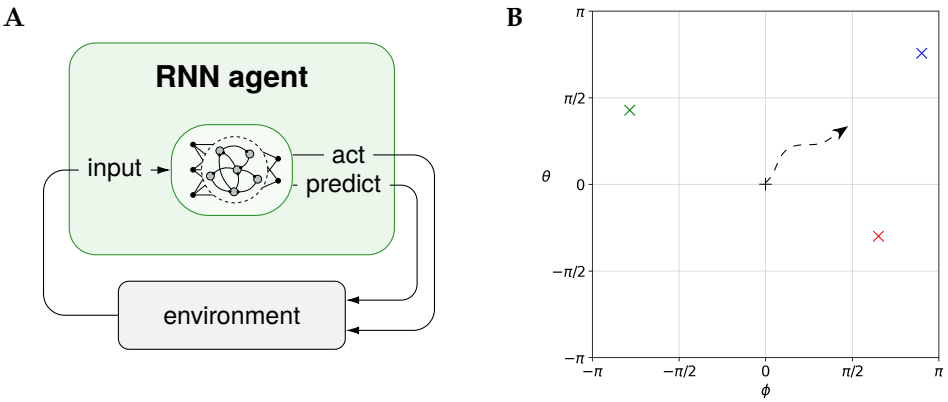


Figure 3.1: **Continuous action space task paradigm. (A)** Agent model structure for the navigation and renavigation tasks. The structure of this model mirrors the RL loop in Figure 2.3, where our agent's input consists of basis function activations $\mathbf{o}_t$ and scalar reward r_t encountered in state $\mathbf{s}_t$, which are integrated within its recurrent GRU network alongside past experiences contained within its hidden state $\mathbf{h}_{t-1}$. Its outputs consist of an action (movement) vector $\mathbf{a}_t = (a_{t,x}, a_{t,y})$ as in eq. (3.1) as well as state predictions $\hat{\mathbf{x}}$ and $\hat{\mathbf{y}}$ predicting relative reward location and identity respectively, each a separate readout from the hidden state. This loop continues for length of the trial. **(B)** Graphical depiction of the paradigm. Agent position $\mathbf{p}_t$ (black plus) is initialised at $(0, 0)$ and object positions $\mathbf{x}_0^{(j)}$ (RGB crosses) are randomly initialised in our periodic angular space with range $-\pi \leq \phi, \theta \leq \pi$ (grid shown for clarity). The agent must then take actions $\mathbf{a}_t$ to successively explore the environment and solve the tasks, as described in the main text. For the navigation task this involves discovering the high reward object then remaining at that location until the trial finishes. For the renavigation task the agent is now teleported upon reaching the target, and so following initial discovery it must leverage its internal representation of relative reward location to successively renavigate back to this target. In these tasks the agent's basis functions from eq. (2.50) uniformly tile the space in a 10×10 grid (not shown) such that it receives information across the entire space at each timestep.

[1]Specifically our objective, and hence the behaviours recruited, become significantly more complex in the case of limited scope, where egocentric notions of space cannot always be built (for example, if a given aperture of space at any point contains no objects).

The action taken in this task at each timestep is an angular velocity vector $\mathbf{a}_t = (a_{t,x}, a_{t,y})$, calculated with a read out from the evolving GRU hidden state as

$$\mathbf{a}_t = \alpha \mathbf{V}\mathbf{h}_t + \boldsymbol{\epsilon}, \quad \epsilon \sim \mathcal{N}(0, \sigma^2_{\text{motor}}) \tag{3.1}$$

where

$$\mathbf{h}_t = \text{GRU}(\mathbf{o}_t, \mathbf{h}_{t-1}; \theta) \tag{3.2}$$

and $\mathbf{V}$ is a readout matrix, α is step size used to keep angular movements of a reasonable magnitude[2], and $\boldsymbol{\epsilon}$ is a small amount of sampled 'motor noise'. $\text{GRU}()$ refers to the equations in Section 2.1.2, where input $\mathbf{x}$ is now the vector of observations $\mathbf{o}_t$. Following this calculated action, agent position is updated as $\mathbf{p}_{t+1} = \mathbf{p}_t + \mathbf{a}_t$ producing new state $\mathbf{s}_{t+1}$ and completing our RL loop.

The incorporation of some stochasticity in this paradigm is crucial to enforce exploration in our agent - here we apply this with a single 'motor noise' injection at the level of action readout, as described in eq. (3.1)[3]. Therefore despite the fact our agent's expected action $\alpha\mathbf{V}\mathbf{h}_t$ is a deterministic readout from the agent's network parameterised by θ, the agent's implicit *policy* π_θ can be considered stochastic due to the additive noise. Formally our agent's policy can hence be considered a Gaussian distribution centred at the readout

$$\pi_\theta(\mathbf{a}_t|\mathbf{s}_t) = \mathcal{N}(\mu_\theta(\mathbf{h}_t), \sigma_{\text{motor}}), \quad \mu_\theta(\mathbf{h}_t) = \alpha\mathbf{V}\mathbf{h}_t \tag{3.3}$$

Finally, with respect to the objectives of our task, we also wish to encourage our agent to build an internal model of egocentric space during each trial. To this extent we train the agent to read out two additional predictive terms from the agent's hidden state at each timestep relating to prediction of the rewarded object, as explained in the following section.

3.3 Training

3.3.1 Navigation Task

To train our agent on the initial navigation task we first form an optimisation objective as a function of the network parameters. Broadly we seek to maximise total reward obtained per trial, however we also introduce two additional, auxiliary loss terms. These govern both identification of the rewarded object and prediction of its location, encouraging the agent to build an internal 'world model' of egocentric space. Our total objective is then

$$J(\theta) = \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[\sum_t^T \left(r(\mathbf{s}_t) + \lambda_1 e^{\sum\left[\cos(\mathbf{x}_t^{\text{reward}} - \hat{\mathbf{x}}_t^{\text{reward}})\right]} - \lambda_2 \sum_j y_j \log \hat{y}_j\right)\right] \tag{3.4}$$

The first term governs reward maximisation, taking the form of a von-Mises based function as in eq. (2.51). The second term forces the agent to predict relative location of the rewarded object $\mathbf{x}^{\text{reward}}$ via a predictive hidden state readout $\hat{\mathbf{x}}^{\text{reward}}$ [4], penalising inaccurate predictions with a similarly von-Mises

[2]While $\mathbf{V}$ is a learnable parameter that also allows our agent to modulate speed, step size α is chosen heuristically to ensure our movements are of a reasonable magnitude, forming trajectories that are visually 'smooth' yet enabling the agent to move at sufficient speed to complete the task.

[3]This can be considered to reflect real world 'motor noise' involved with imperfectly performing physical actions.

[4]Specifically, we read out 4 values which we assume to describe predicted reward location as $\mathbf{x}_t^{\text{readout}} = [\sin(\hat{x}_t), \cos(\hat{x}_t), \sin(\hat{y}_t), \cos(\hat{y}_t)]$, then compute $\hat{\mathbf{x}}_t^{\text{reward}} = \left(\arctan 2\left(\frac{\sin(\hat{x}_t)}{\cos(\hat{x}_t)}\right), \arctan 2\left(\frac{\sin(\hat{y}_t)}{\cos(\hat{y}_t)}\right)\right)$, enforcing a circular notion of relative location. The additional use of a circular loss of form $e^{\cos()}$ is therefore not strictly necessary, but is in keeping with circular functions described previously Section 2.4.

based loss. The third term is a cross entropy loss which enforces the agent to correctly identify the rewarded dot, using one-hot truth labels y_j and corresponding predictions $\hat{y}_j$ calculated following a softmax activation of a separate readout. By forcing the agent to represent these quantities the agent is encouraged to build some internal notion of rewarded location within egocentric space, which we will later examine. Both auxiliary losses are *supervised* - they are calculated with respect to unseen ground truth quantities. The λ_1 and λ_2 terms determine relative weighting of each loss.

At code level this objective is approximated as a sample estimate - often denoted $\hat{J}(\theta)$ - by averaging over N trajectories. Importantly, it can be shown this sample-based objective is entirely differentiable such that we can directly apply gradient based optimisation methods from Section 2.2[5]. In other words, all calculations involved with forming our objective from $t = 1 : T$ are differentiable with respect to θ - our system is *end-to-end differentiable*. While there exists stochastic motor noise at the level of action readout, as previously described, by considering this noise instead as an *input* to the system we are able to retain full differentiability throughout the RL loop. This is known as the *reparameterization trick*, detailed further in Appendix B.4. With appropriate parameter selection[6] and network initialisation (Appendix B.1) we can then optimise our objective with backpropagation (Appendix C.2.1). Figure 3.2A-C displays examples of trained agent behaviour for this task.

3.3.2 Renavigation Task

We then implement the *meta-testing* phase of our meta-learning paradigm, where we consider our agent's ability to generalise behaviour learnt to the renavigation task following a short adaptation period. For this period we use a heuristic training length of a factor of 10 less than that of the navigation task, and now set each auxiliary loss contribution to zero (such that our objective from eq. (3.4) reduces solely to the reward maximisation term). Both choices ensure any behaviours exhibited by our agent at test time can be attributed largely to its capacity for adaptation as built during the previous task, allowing us to consider the meta-learning capacity of our agent.

The main change in paradigm for this task comes in the form of modified state-transitions, where the agent is now teleported to a random location upon reaching the reward. The agent's objective is then to perform as many teleportations and renavigations back to the reward as possible during each trial. Such a modification does not affect the differentiability of our objective: by considering each teleportation as 'segmenting' our trials - severing backpropagating gradients at these points - each trial effectively represents a number of composite sub-trials. Specifically, we enforce teleportation *probabilistically* as

$$p(\text{teleport}|r_t) = \alpha_1 r_t - \alpha_2 r_t^2, \quad \text{where} \quad \alpha_1 > \alpha_2, \;\; r_t \approx e^{-\frac{\kappa_r}{2}\|\mathbf{x}_t^{\text{reward}}\|^2} \tag{3.5}$$

sampling from this function at each timestep. Note reward is modelled by a von Mises function but can be approximated as Gaussian within the small region where teleportation is likely (Section 2.4.2). The use of a probabilistic function is motivated by the observed behaviour that if teleportation is triggered at some static *threshold* of reward (or proximity thereof), the agent learns the unintended behaviour of 'stopping short' just before the threshold, remaining static thereafter and obtaining high total reward without needing to renavigate. However this behaviour also remains to an extent with probabilistic teleportation, given the agent can accrue similarly high total reward by remaining further from the target and hence triggering fewer teleports per trial. The function used solves these problems by decreasing

[5] At code level optimisers are designed to minimise some loss, and so formally we minimise $-\hat{J}(\theta)$.

[6] Simulation parameters consist of both environment parameters, such as scale parameters for the basis and reward functions and noise magnitude, and training hyperparameters, such as number of epochs and learning rate. These are detailed for each task in Appendix C.

teleportation probability slightly at the centre of the target, increasing expected reward associated with this position and therefore incentivising direct navigation to the centre of the target (Figure 3.2E)[7].

Finally, we can then optimise our objective as before with a short period of further training (Appendix C.2.2). Figure 3.2D displays examples of post-training task-behaviour.

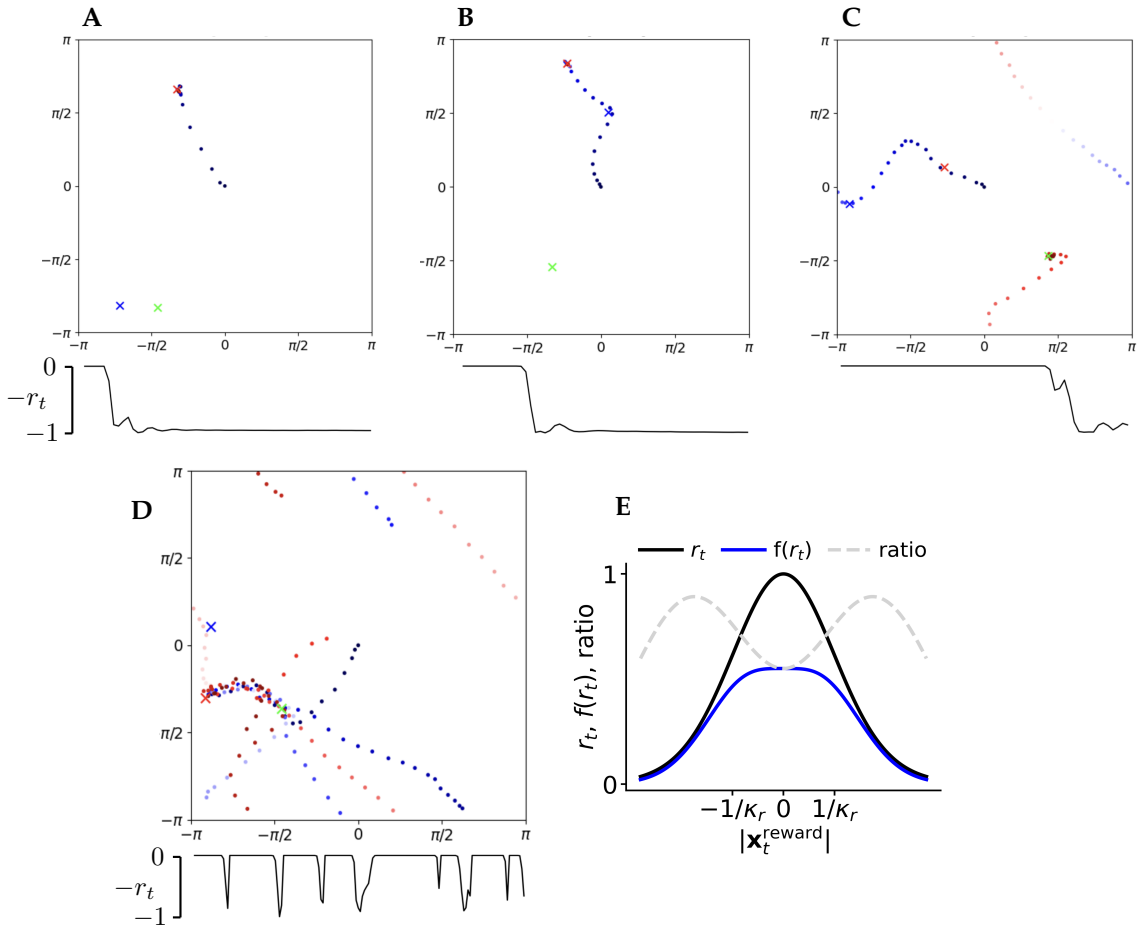


Figure 3.2: **Navigation and renavigation task behaviour. (A-C)** Post-training behaviour in the navigation Task. Each plot shows a single trial, demonstrating successful navigation to the rewarded object after sequentially checking each object (RGB crosses) to determine which is rewarded. The agent accomplishes this task on first attempt (A), second attempt (B) and third attempt (C) respectively, where blue-red colourmap denotes agent position within each $T = 80$ timestep trial. Appended timeseries below each plot describe reward r_t obtained at each timestep (negative reward reflects *minimisation* of the flipped objective $-\hat{J}(\theta)$ at code level). **(D)** Meta-testing behaviour in the renavigation task. The agent is now teleported upon reaching the rewarded object and must successively renavigate back to this location. The plot shown once again depicts a single trial, now of length $T = 200$ to allow for multiple renavigations per trial. We clearly observe multiple segmented trajectories due to the teleportation effect, each ending with successful renavigation to the rewarded object (red cross). Discrete jumps in the appended reward timeseries similarly denote instances of teleportation. **(E)** Probabilistic teleportation curve $p(\text{teleport}|r_t)$ (blue) as a function of reward r_t (black), as described in eq. (3.5), with the ratio between the two also shown (dotted grey). X axis describes relative position (Euclidean norm) of the rewarded object, where κ_r refers to the scaling of the reward function in eq. (2.55), in this case $\kappa_r = 2.5$. The local minima in this ratio demonstrates how direct navigation to the centre of the reward is promoted, granting maximum reward in expectation as discussed in the main text. Function constants α_1, α_2 are chosen empirically as to produce teleportation at appropriate intervals (Appendix C.2.2).

[7]While one intuitive choice to implement teleportation is to simply set reward and teleportation to happen simultaneously (i.e. using a discrete reward), our training procedure relies on a smooth, *differentiable* function of reward. The above method can therefore be considered a workaround to produce desired behaviour while retaining differentiability.

3.4 Results

3.4.1 Task Statistics

Following meta-training of our agent on the renavigation task, we consider task statistics describing the resulting behaviour with respect to that of the initial navigation task (Figure 3.3). Overall, it is

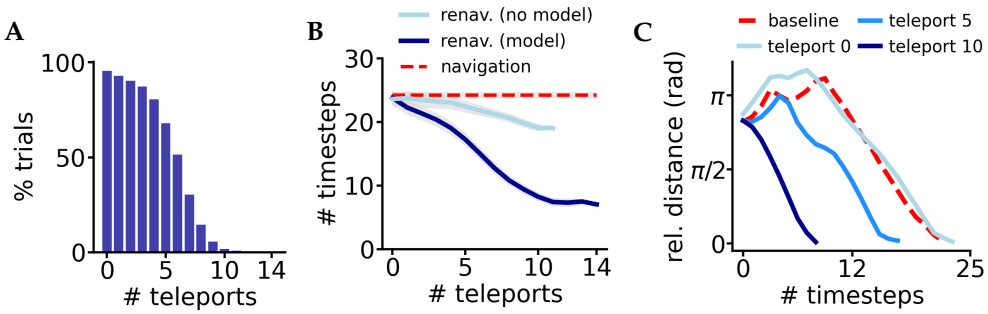


Figure 3.3: **Navigation and renavigation task statistics. (A)** Mean number of teleportations triggered per trial in the renavigation task. We observe a significant proportion of renavigation trials (length $T = 200$) result in successful discovery of the reward and hence at least one teleportation, with the percentage of trials for which a given number of teleportations were triggered steadily decreasing thereafter. **(B)** Mean number of steps taken to find reward across number of teleportations within-trial. Plots are shown for an agent pre-trained in the initial navigation task with the full objective Baseline (dashed red) describes the mean number of timesteps taken by the agent to find the reward in the navigation task. Standard error of the mean is described with grey shading. We observe a steady decline in number of timesteps required to renavigate to the reward following subsequent teleportations for both the model and no model agent. This decline is much more significant however for the model-based agent, which hence performs a greater total number of renavigations per trial. **(C)** Median renavigation trajectory. Trajectories are plotted as relative distance to target (Euclidian norm) vs timestep. These trajectories correspond to the median total timesteps taken for renavigation, for the 0^{th}, 5^{th} and 10^{th} renavigations (blue shades). Baseline (dashed red) describes median trajectory for the navigation task. We observe increasingly rapid renavigations within-trial, as shown by faster median renavigation trajectory for increasing number of teleportations.

clear our agent successfully completes the meta-learning task of adapting to the new context of the renavigation task under minimal training, successively renavigating back to the reward multiple times per trial (Figure 3.3A) at above-baseline performance. This is demonstrated by faster renavigations to the reward when compared with the initial navigation paradigm (Figure 3.3B,C), suggesting the agent leverages some internal representation of egocentric rewarded location to enable faster renavigations. These renavigations become quicker within-trial, where the agent appears to take an increasingly more optimal path to the reward for each successive renavigation. This further suggests the agent's internal representation of egocentric rewarded location which governs this behaviour becomes stronger within-trial as the agent performs successive renavigations.

Importantly, while the inclusion of auxiliary losses in the navigation task has negligible effect on *navigation* task performance (not shown), the agent pre-trained with auxiliary losses performs much better at the renavigation task post-adaptation, performing faster renavigations and a greater number per trial when compared to the agent pre-trained without these losses. This demonstrates the internal model that the agent is encouraged to build in the navigation task is successfully leveraged to improve performance in renavigation task. This supports the idea that explicitly encouraging the development of a predictive 'world model' can facilitate better meta-learning capability via stronger, more rapid adaptation to new contexts.

Chapter 4

Simulations in Discrete Action Space

The remaining simulations in this work focus on a more neurophysiologically inspired model, where our agent's network reflects a modular anatomical structure recently proposed to be employed within the SC [33, 62]. Such research suggests the SC's surface to be tiled by an array of discrete anatomical 'modules', each associated with a single egocentric region of space and governing both sensory cues received and saccades produced in that egocentric direction. Under this theory each module can be viewed as a self-contained 'functional orienting unit'. Reflecting this, our agent's actions are now considered to represent direct saccadic movements and as such are selected from a set of *discrete* vectors tiling egocentric space surrounding its instantaneous position. Such motor behaviour can be considered more reflective of the SC's role in mediating ballistic, endpoint-based saccadic movements, and allows us to consider the plausibility of such a modular physiology. Furthermore, we use this framework to also consider the ability of such a model to 'plan' a given saccade from this set - simulating its outcome via use of a separate pre-trained visual model - reflecting recent proposals the SC may trigger internal simulations of the environment [33, 63].

4.1 Task Overview

This chapter introduces an object tracking task, where our agent must make a series of discrete movements to track a rewarded object as it moves through angular space. Environment state is now characterised solely by the relative egocentric location of the moving target with respect to the agent, and there is no longer a notion of colour[1]. This rewarded target moves with constant velocity in angular space with randomly sampled direction and speed - which includes speeds greater than the maximum speed of the agent. As such, the agent is forced to develop a meta-strategy for maximising reward based on target speed: while the agent can comfortably track slower targets, for faster targets the optimal policy involves predicting target movement within the space and 'intercepting' the target when possible. Broadly, this task then investigates the egocentric tracking behaviour that emerges from such a model, drawing potential links to corresponding dynamics employed within the SC to govern similar tasks.

Given our desire to implement a more plausible model of a saccade-producing network under this paradigm, we further implement a number of ethologically-motivated changes compared to the previous task. First, we limit the visual scope of our agent from the full range of the environment in Chapter 3 down to a partially observed aperture, reflecting the limited scope of observations of our surroundings

[1]Hence our agent receives a single input channel - representing a greyscale depiction of the environment - simplifying computations by reducing the dimensionality involved.

experienced ethologically. We set the agent's maximum saccade range to be equal to that of the aperture, implementing the realistic heuristic limit that our agent can only move as far as it can observe.

Next, we introduce a refractory period of P timesteps alongside each movement where the agent cannot immediately take another action and must instead remain static, though it continues to sample the environment and receive reward[2]. The motivation for this multifold:

- Task constraints: To investigate the meta-strategy of our network to solve our task across target speeds we require some target velocities $\mathbf{v} = (v_x, v_y)$ to be sampled at speeds greater than the maximum agent speed V_{agent}, i.e.

$$v_x, v_y \sim \text{Uniform}(0, V_{target}), \quad V_{target} = R_v V_{agent} \tag{4.1}$$

  where V_{target} and V_{agent} denote maximum target and agent speeds in x or y directions in angular space, and $R_v > 1$ is the ratio set between these speeds. The problem is then that our agent may only receive a single visual sample of the target as it quickly passes through its aperture of maximum span $2V_{agent}$ (Figure 4.1B), so will be unable to internalise target velocity (as required to implement a model-based strategy). By implementing a refractory period P we lower maximum effective agent speed as $V_{agent} \leftarrow V_{agent}/P$, lowering the V_{target} required to satisfy eq. (4.1) and therefore reducing the speed of sampled targets, solving this problem[3].

- Ethological realism: Additionally, it is realistic to suggest the rate of environmental sampling far exceeds the rate at which animals can perform complex motor actions such as saccades. A refractory period P can be considered to implement this by increasing the relative sampling rate of the environment, where P then denotes the number of observations made for each movement.

- Analysis: Finally, such an assumption allows us to analyse the model's network during the static refractory period, allowing us to consider the contribution of such 'preparatory' dynamics to the agent's behaviour.

Finally, we instil in our agent the ability to 'plan' by simulating the movement described by a given saccade vector. Such a decision involves forgoing an overt movement to return a real state observation and reward, and instead obtains a covert, 'fictitious' estimate of this information without effecting the environment (i.e. the agent's true 'position' remains unchanged). To perform such a plan our agent uses a visual 'predictive model', which is separately pre-trained as detailed in Section 4.2.1. The inclusion of this capability is inspired by research proposing the possibility the SC may utilise a modular structure to trigger covert internal simulations (plans) in a similar manner to that of overt movements. Reflecting the intuition that internally simulating an action is typically much faster than performing said action we place no refractory period restriction on plans as with movements, heuristically setting plan length to be equal to a single timestep (i.e. the same rate as environmental sampling, and P times faster than a movement). This implicitly imposes a relative movement-plan 'time cost' ratio of P[4].

Broadly, we then seek to analyse the optimal policy developed for this tracking task to provide insight

[2]At code level the additional refractory time added to a movement is of length $P-1$, such that a total movement is considered to take P timesteps. However for simplicity, we refer to the refractory period and 'total movement length' as both being of length P (i.e. the refractory period 'starts' on the same timestep as the movement).

[3]To further explain this, consider that we need at least 2 samples of the target within the aperture to internalise its velocity, such that in the worst case scenario (when the target starts just outside the aperture), we need $3V_{target} \leq 2V_{agent}$ to ensure 2 samples of the target within the agent's aperture spanned by $2V_{agent}$. Together with eq. (4.1) we then obtain $R \leq \frac{2}{3}$, but our task requires $R > 1$. By lowering effective agent speed with a refractory period as $V_{agent} \leftarrow V_{agent}/P$, we obtain $R \leq \frac{2P}{3}$, such that we can freely set R given we implement a corresponding refractory period $P \geq \frac{3R}{2}$.

[4]This is important as planning is detrimental in that sense that it provides no 'true' reward but costs time, so must be relatively incentivised compared to moving. This ratio neatly implements this, in lieu of more complex schemes such as those considering metabolic cost of movement [111].

into the following, once again drawing appropriate links to superior colliculus literature where relevant:

- What is the optimal policy developed across the range of target speeds, and to what extent is planning utilised within this policy?
- How is this optimal policy represented by the agent - to what extent does it implement a deterministic 'winner takes all' strategy?
- What is the form of sensorimotor transform implemented - does the agent implement a 1-1 mapping from visual input to action policy, as implied by evidence of a modular structure in the SC?

4.2 Model Overview

This model in this task uses a GRU as previously, where now $H = 100$ units, with a few key differences. First, we limit the agent's aperture to a $\pm\pi/2$ window either side of its angular location to enforce our limited scope condition[5]. This aperture is now tiled by 36 equally spaced visual basis functions. In addition, the model's output action vector is now sampled from a discrete set of possible vectors $S_a = \{\mathbf{a}^{(1)}, \mathbf{a}^{(2)}, ..., \mathbf{a}^{(n)}\}$, where $n = 81$, which tile egocentric space within the agent's aperture (Figure 4.2B)[6]. Such action vectors are sampled from the agent's (now stochastic) categorical policy $\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$ produced from a softmax activation of a linear readout, i.e.

$$\mathbf{a}_t \sim \pi_\theta(\mathbf{a}_t|\mathbf{s}_t), \quad \text{where } \pi_\theta(\mathbf{a}_t|\mathbf{s}_t) = \text{Softmax}(\mathbf{V}\mathbf{h}_t) \tag{4.2}$$

Importantly, the agent's action space is now additionally expanded: for each action in S_a it now has the additional, alternative option of *planning* this action, forgoing an overt movement to covertly *simulate* the effect of the action via use of a separate, pre-trained predictive model. This introduces an additional choice to move or plan which we denote with the binary 'decision' variable $d \in \{0, 1\}$ for moving or planning respectively. This effectively doubles the agent's total action space, which can be formalised as a cartesian product between S_a and d as $S_a \times d = \{(\mathbf{a}^{(1)}, 0), (\mathbf{a}^{(1)}, 1), (\mathbf{a}^{(2)}, 0), ...\}$. This decision variable is sampled as in eq. (4.2) from a separate softmax read-out, and we explicitly denote the total sampled action (both vector and decision) as $\mathbf{a}_t^{(k,d)}$, where $\mathbf{a}^{(k)} \in S_a, d \in \{0, 1\}$.

This predictive model uses a Minimal Gated Unit (MGU) [115] which receives identical inputs to the main GRU network - denoted the 'policy network' - however instead of producing an output policy over actions and decision it is trained to *predict the next state*. In other words given information from the current timestep, the model's goal is to predict the vector of activations obtained for the next state $\mathbf{o}_{t+1}^{tot}$. The *tot* superscript denotes the fact the model is trained to predict activations over the *total space* - represented by 144 basis functions which tile the entire torus - as opposed to just the 36 contained within the agent's aperture of range $\pm\pi/2$[7]. This is motivated by the fact we wish to encourage our agent to predict target location even when not within scope of its aperture, internalising this information within its hidden state to enable strong task performance. Despite this, if the agent decides to plan only the 36 simulated activations from *within* the aperture at the planned location are fed back to the main network (i.e. *predicted* observations retain the same dimensionality as true observations).

[5]Our agent hence observes 25% of the toroidal space at all times. This is a reasonable heuristic, given the proportion of egocentric environment observed can be appoximated as around 21% in humans [112], is similar in other primates [113], and is significantly larger in animals with side positioned eyes such as birds and rodents (at the sacrifice of a smaller binocular region, reducing depth sensing ability) [114].

[6]Choice of a square of an odd number for $n = |S_a|$ (the *cardinality* of S_a) ensures there exists a 'zero vector', giving our agent the choice to remain stationary if needed.

[7]We arrange this 'total' array of basis functions to uniformly tile the entire space, whilst ensuring the corresponding basis functions contained within the agent's aperture are co-located with those received as input by our agent in the main task. In this way, predictions made at the within-aperture basis functions are directly comparable to the inputs received during movement.

Note that while movements ($d = 0$) incur a refractory period as discussed in Section 4.1, planned movements do not and so can be sequentially enacted at subsequent timesteps, reflecting a *rollout* of a planned trajectory through space. This has the effect that planning is P *times faster* than moving, where P is the refractory period, allowing our agent to internally simulate movement at a faster rate than overt movement. Figure 4.1A graphically depicts the above described model within the RL loop, describing interactions between the policy and predictive networks.

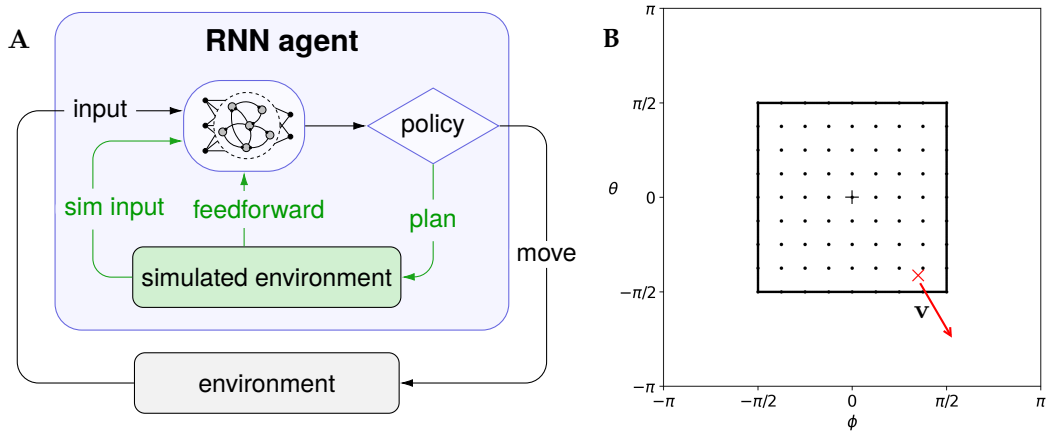


Figure 4.1: **Discrete action space task paradigm. (A)** Model structure for the tracking task. As before, at each timestep our agent receives state observations $\mathbf{o}_t$ and reward r_t, and produces output action $\mathbf{a}_t$. These actions are now *sampled* from the agent's stochastic, categorical policy which consists of both a movement vector sampled as $\mathbf{a}^{(k)} \in S_a$ as well as a separate decision variable $d \in \{0, 1\}$ denoting the decision to move or plan, with total action then denoted $\mathbf{a}_t^{(k,d)}$. This decision broadly splits the policy into two options: a *movement* $\mathbf{a}_t^{(k,d=0)}$ where our agent overtly moves to the location dictated by vector $\mathbf{a}^{(k)}$, or a *plan* $\mathbf{a}_t^{(k,d=1)}$ where our agent instead covertly *simulates* the environmental state encountered following the same movement. While the former involves interacting with the 'true' environment (grey), the latter evokes use of a separately pre-trained visual predictive model to return a *simulated* version of the information that would be received (i.e. $\hat{\mathbf{o}}_t$ and $\hat{r}_t$), and is therefore considered as interacting with a *simulated environment* (green) self-contained within the agent (note simulated reward does not contribute to the total reward describing agent performance). Note overt movements invoke a P timestep refractory period - during which a zero vector $\mathbf{a}_{\text{refac}} = (0, 0)$ is fed into the network - which planning does not, such that the agent can effectively plan P times faster than it can move. Additionally, the hidden state of this predictive model is continuously fed into the policy network (GRU) in a feedforward manner, allowing predictive information contained within this network to be integrated into decisions made by the policy network. Finally, sampled actions and decisions are now also fed back as inputs to the model at each timestep in the form of *one-hot* binary encodings. Such information enables the agent to learn the motor map relating each action to its corresponding movement vector (and hence correctly assign probability mass to the intended movement), as well as discern if it currently moving or planning. **(B)** Graphical depiction of the paradigm. Rewarded target location $\mathbf{x}_t$ (red cross) is randomly initialised within the agent's aperture (black box), with randomly sampled constant velocity $\mathbf{v}$ (red arrow) as in eq. (4.1). Target position evolves as $\mathbf{x}_{t+1} = \mathbf{x}_t + \mathbf{v}$, and the agent (black plus) must then track the target via a sequence of movements and plans $\mathbf{a}_t^{(k,d)}$ - each associated with one of the 81 locations (black dots) within its aperture (black box) - noting each movement incurs a P timestep refractory period. The agent's observation $\mathbf{o}_t$ consists of 36 scalar basis functions activations tiled in a 6×6 grid within the aperture (not shown), which extends to $\pm\pi/2$ in both directions. Note alongside its *true* position $\mathbf{p}_t$ the agent now also has a *planned* position $\tilde{\mathbf{p}}_t$ (grey plus, not shown), describing the agent's theorised location as it internally simulates (plans) a movement. During overt movements (including refractory periods) these can be considered colinear (i.e. $\tilde{\mathbf{p}}_t = \mathbf{p}_t$), but during planning the planned location deviates as the agent *internally simulates* the environment from an alternative location, with the two then returning to be colinear when the plan ends. True and planned positions are both updated according to the sampled action $\mathbf{a}_t^{(k,d)}$, where movements update both $\tilde{\mathbf{p}}_t$ and $\mathbf{p}_t$ (which remain colinear), and 'plans' update only $\tilde{\mathbf{p}}_t$.

4.3 Training

4.3.1 Visual Model Pre-training

Before discussing training our agent under the above paradigm, we must describe the pre-training of the visual predictive model used by the agent to implement its ability to plan. Formally the objective of this model is to minimise the error made in its predictions of the environment

$$\mathcal{L}(\theta) = \sum_{t=1}^{T} \|\hat{\mathbf{o}}_t^{tot} - \mathbf{o}_t^{tot}\|_2^2 \tag{4.3}$$

where $\hat{\mathbf{o}}_t^{tot}$ is the predicted vector of observations extending over the 'total' space, and $\mathbf{o}_t^{tot}$ is the ground truth of this quantity. Note that when computing the ground truth observation vector we continue to use the form of activations in Section 2.4.2 which includes a small amount of sampled Gaussian noise[8]. One benefit of enforcing our model to predict total space is that we can quantify the model's effectiveness at tracking the target as it moves through space, both within and out of scope of the aperture, by representing the circular mean and variance of our predicted activations with a corresponding ellipse and comparing this to the true target location (Figure 4.2).

Such a training paradigm reflects a form of *self-supervised learning* - where predictions made preceding an action are subsequently updated based on environmental feedback - reflecting the method often considered to be employed ethologically [119, 120]. In this paradigm, however, we modify this format to consider predictions over the entire space that cannot be realistically self-supervised by our limited-scope agent, instead simply assuming the entire ground truth $\mathbf{o}_t$ is available at all times as our teaching signal[9]. While the inputs to this model are identical to the policy network, action vectors and decisions at each timestep are randomly sampled during training with intent to approximately match the marginal distribution of actions and decisions selected by the agent when performing the task. Figure 4.2 depicts this training paradigm, with further model training details given in Appendix C.2.3.

4.3.2 Tracking Task

After pre-training this predictive model and integrating it into the agent's structure (Figure 4.1A), we can begin training on the tracking task. Due to the stochasticity of our agent's policy our training loop is no longer fully differentiable, and so we turn instead to the methods in Section 2.3 which enable us to optimise our objective based on a large number of sampled trajectories $\tau = (\mathbf{s}_1, \mathbf{o}_1, \mathbf{a}_1, r_1, \mathbf{s}_2, \mathbf{o}_2, \mathbf{a}_2, r_2, ..., \mathbf{s}_T, \mathbf{o}_T, \mathbf{a}_T, r_T)$. Specifically we use the PPO algorithm (Section 2.3.4), making a few key modifications to the task paradigm and optimisation algorithm to encourage desired behaviour and stabilise training:

- Kullback-Leibler Regularisation: During RL training it is common to include some form of regularisation to encourage exploration. In our work we introduce a Kullback-Leibler (KL) regularisation term over the output distributions of both vectors and decisions. Our full actor objective is then[10]

$$J_{\text{actor}}(\theta) = J_{\text{PPO}}(\theta) + \lambda_1 \text{KL}(\pi_\theta^{\text{vec}} || \mathbf{p}^{\text{vec}}) + \lambda_2 \text{KL}(\pi_\theta^{\text{dec}} || \mathbf{p}^{\text{dec}}), \quad \text{KL}(\mathbf{q}||\mathbf{p}) = \sum_i q_i \log \frac{q_i}{p_i} \tag{4.4}$$

[8]Such additive noise can be shown to be beneficial when training neural networks, acting as a form of regularisation and helping to better generalise predictions made, especially early in training [116, 117].

[9]This can be considered 'self-supervised in the limit' - as the number of training samples tends to infinity, such a model could reasonably self-supervise the process of learning to predict activations across the total space.

[10]Note the critic loss is considered separately - *minimised* by separate updates as described below under 'Value Estimation'.

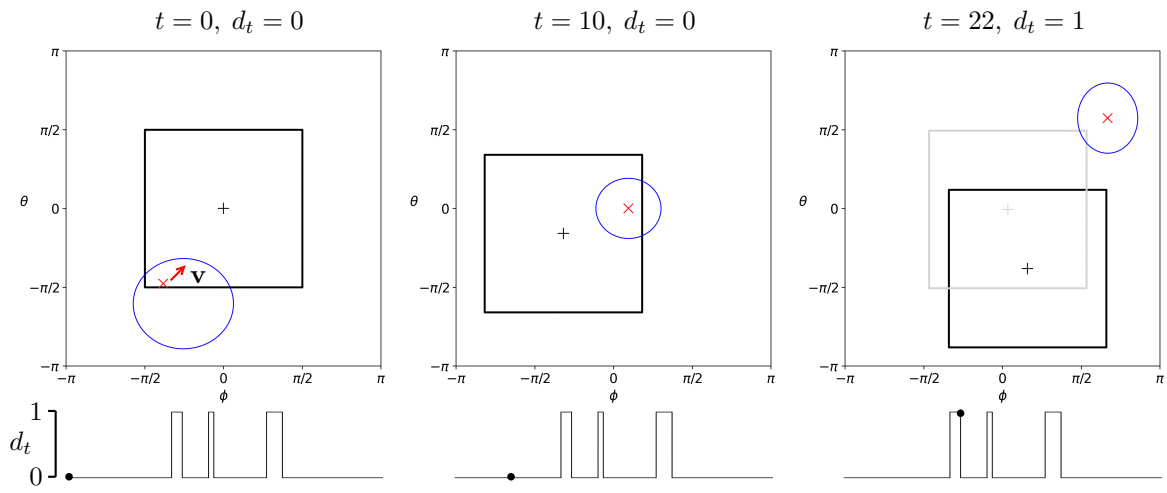


Figure 4.2: **Visual model pre-training.** Pre-training paradigm. Our agent (black plus) attempts to predict total visual activations $\hat{\mathbf{o}}_{t+1}^{tot}$ (not shown) at the following timestep, given current partial observation $\mathbf{o}_t$ from within its aperture (black box) and the *one-hot* encoding of the action just taken $\mathbf{a}_t^{(k,d)}$. The blue ellipses describes the circular mean and variance of the agent's predicted activations - a small ellipse encircling the target (red cross, constant velocity vector $\mathbf{v}$ indicated with arrow) indicates an accurate prediction of target location. The 3 panels depict the models performance at 3 different points within a single trial post-training, with appended timeseries below denoting the sampled decision to move $d_t = 0$ or plan $d_t = 1$. These figures refer to a task paradigm with refractory period $P = 5$ (i.e. where a movement takes 5 timesteps). The left panel displays the model's initial prediction at $t = 0$, correctly capturing target location within its predictive ellipse albeit with high initial variance. The centre panel shows the prediction after $t = 10$ timesteps of sampled observations (i.e. 2 movements), demonstrating higher confidence in the target location via a significantly smaller ellipse. The right most panel shows the model's prediction during a sampled plan at $t = 22$, where planned position $\tilde{\mathbf{p}}_t$ (grey plus; planned aperture described with grey box) now deviates from true position $\mathbf{p}_t$ and the agent now receives fictitious, simulated input from within the grey aperture. Nevertheless the model performs well, having internalised target velocity across prior observations. In attempt to roughly mirror the marginal distribution of total actions $\mathbf{a}_t^{(k,d)}$ (vectors and decisions) made by the agent post-training, vectors $\mathbf{a}_t^{(k)}$ are uniformly sampled from S_a at each (non-refractory period) timestep and decision d_t is sampled at each non-refractory period timestep using a random variable with constant probability α of switching between moving ($d_t = 0$) and planning ($d_t = 1$), where $d_t = \begin{cases} 1 - d_{t-1}, & \text{probability } \alpha \\ d_{t-1}, & \text{probability } (1-\alpha) \end{cases}$. This reflects our expectation that our agent will perform multiple sequential plans and movements, switching between the two across a given trial. As previously explained each movement incurs a P step refractory period, demonstrated in the binary timeseries above as segments of P consecutive zeros. An example video for this task can be found at [118].

where π_θ^{vec} and π_θ^{dec} denote the movement vector and decision distributions which together comprise the agent's policy, $\mathbf{p}^{\text{vec}}$ and $\mathbf{p}^{\text{dec}}$ are the priors placed on these distributions[11], and λ_1 and λ_2 govern the relative weighting of each loss term. By penalising deviations of the agent's policy from the priors, we encourage our agent to remain close to these priors early in training, promoting our agent to explore its full action space. This effect is discussed further in Appendix C.2.4.

- Reward Shaping: One problem found empirically is that given the majority of available reward is associated with lower speed targets[12], the agent overfits its policy to focus on these targets and develops a suboptimal approach for faster targets. To combat this we introduce a form of reward

[11]Such a form of KL regularisation represents the *reverse* KL, where we penalise deviations in the output distribution from our prior (or *target*) distribution. Unlike the *forward* KL, such a term does not diverge under typical conditions [121].

[12]Targets sampled in a range below the maximum 'speed' of the agent can be easily tracked, obtaining high cumulative reward, whereas faster targets may only be 'caught' a few times per episode.

shaping, modifying maximum reward from unit magnitude to now allocate increasing reward to faster targets as

$$r_{\max} = 1 + \alpha_1 \|\mathbf{v}\|^{\alpha_2} \tag{4.5}$$

where $\|\mathbf{v}\|$ denotes target speed (Euclidean norm of target velocity) and α_1 and α_2 are chosen empirically to obtain similar average post-training reward across target speeds (Appendix C.2.4).

- Value Estimation: An important element of A2C-based policy gradient methods (Section 2.3) is the *critic* estimation of the value of a given state, $\hat{V}(\mathbf{s}_t)$[13]. An accurate critic is crucial to stabilise training and as such is often parameterised by a separate network. In our case we simply obtain our value estimate from a readout of the evolving hidden state of the policy network. However this represents a particularly challenging task in our partially observed paradigm, which does not feature an explicit state-value mapping let alone an observation-value mapping. To improve the quality of our critic's estimates we therefore introduce the following modifications:

  1. Multi-layer Readout: Instead of using a single linear hidden state readout for the value estimate, we increase the complexity of our value function by including an intermediate hidden layer of $N = 32$ neurons, with our estimate then

  $$\hat{V}(\mathbf{h}_t) = \mathbf{V}_2(\mathrm{ReLU}(\mathbf{V}_1\mathbf{h}_t + \mathbf{b})) \tag{4.6}$$

  where $\mathbf{V}$ and $\mathbf{b}$ denote readout weights and biases, and ReLU is the Rectified Linear Unit activation function.

  2. Reverse Counter: Another difficulty is that value is highly correlated with time remaining in the trial, but state observations don't necessarily contain sufficient information to estimate this. To this extent we introduce a 'reverse counter' into the value estimation, appending the hidden state referenced in eq. (4.6) with the normalised trial time remaining

  $$t' = 1 - \frac{t}{T} \tag{4.7}$$

  where T is trial length, to enable the agent to decorrelate time from the value estimate.

  3. Multiple Critic Updates: We also perform multiple critic updates for each mini-batch of sampled trajectories used to update the actor. This approach - performing multiple incremental optimisation updates on the critic using a single batch of trajectories - stabilises training by helping to ensure the critic continues to perform well as the actor's policy changes [122].

Figure 4.3 shows this paradigm post-training across 2 different task scenarios. A more detailed discussion of this training process is given in Appendix C.2.4.

[13] Or in our partially observed case, $\hat{V}(\mathbf{h}_t)$.

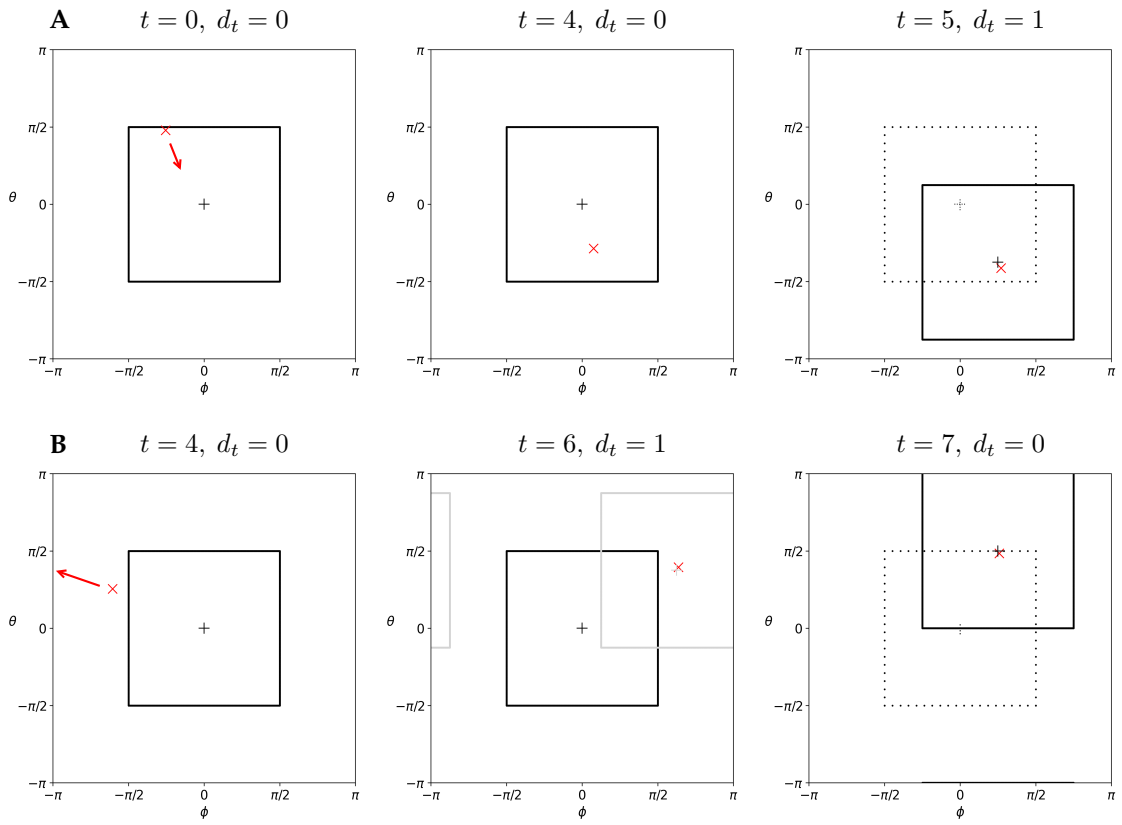


Figure 4.3: **Tracking task behaviour. (A)** Example task behaviour for a slow moving target. In this task the target (red cross) is sampled with low velocity (red arrow, unseen by the agent), producing 'following' behaviour from the agent (black plus), which maximises reward by choosing saccades which successively follow the target through space. After first observing the target at $t = 0$ (left tile; in this case the agent begins the trial with a 'zero vector', remaining static), the agent receives multiple sampled observations during the ensuing refractory period up until $t = 4$ (centre tile). At this point the agent has successfully internalised target velocity such that it can perform the optimal saccade at $t = 5$ (right tile), moving directly to the target and obtaining maximum reward. Black dotted lines show agent position in the previous timestep. Given our agent may only make discrete saccades as contained in S_a (Figure 4.1B), it cannot repeat a single optimal movement (i.e. the target vector) for the length of the trial and must instead dynamically determine the optimal movement vector throughout the trial of length $T = 60$. **(B)** Example task behaviour for a faster sampled target, demonstrating planning within an 'interception' paradigm. The target once again initially appears within the agent's aperture, where its velocity is internalised within the agent's network within the first few steps of the trial (left tile). However in this case the target is too fast to track and quickly moves out of scope. The agent therefore decides to make a covert plan, internally simulating target location during timesteps $t = 5, 6$ (centre tile). The agent then uses the information obtained in this plan (which includes both simulated state observations and reward from the planned location) to subsequently select the appropriate saccade to 'intercept' the target at $t = 7$ (right tile) when it reappears at an accessible location within the periodic space (i.e. with respect to the agent's 'true' location). Such an strategy can be considered as offloading the difficult task of predicting out-of-scope target location to a separate region of the network which is better suited to perform this. Both strategies are implemented by the agent's trained policy after a single meta-training period across a range of target speeds, as discussed in Section 4.4. Example videos for this task can be found at [118].

4.4 Results

4.4.1 Task Statistics

To address our objectives in Section 4.1 we can first consider the macroscopic statistics describing our agent's behaviour post-training (Figure 4.4). From Figure 4.3, we observe qualitatively the optimal policy developed varies based on target speed, where the agent employs 'following' or 'interception' behaviours accordingly. This is also represented within task behavioural statistics, where across increasing target speed we observe both a decrease in relative reward and an increasing use of planning (Figure 4.4B). Such plans are triggered largely in cases where the target is just out of scope of the agent's aperture (Figure 4.4C), and are typically followed by movements chosen in directions that follow the plan just made (Figure 4.4D). This suggests planning is used to quickly simulate the effectiveness of a movement before it is made, effectively offloading the complex task of target location prediction (Figure 4.3B). Importantly planning is shown to be overall beneficial to the agent: shuffling planning periods or removing the ability to plan from our trained agent both negatively impact performance (Figure 4.4A).

Differences in behaviour across target speeds can also be examined at policy level. For slower targets we observe a more deterministic policy, where the agent's sampled actions typically directly align with the target's velocity vector (Figure 4.4E) and the agent's policy overall demonstrates a lower average entropy (Figure 4.4F). At faster target speeds the converse is true. The agent can no longer directly follow the target so instead recruits more complex behaviours associated with a greater degree of uncertainty (higher entropy), assigning mass to multiple possible movements including both moving and planning. This can be explained by the fact that for faster targets there may be multiple similarly optimal future action sequences at any given timestep.

We can hence comment on the degree to which input activations appear to be transformed to motor actions in a direct '1-1' mapping - for slower targets this appears to be the case, where the optimal action across the trial represents a direct movement to the target (Figure 4.4E). For faster target velocities this breaks down, as there is no longer a direct relation between target location and optimal action. We also observe a large difference in policy entropy for actions made when the target is inside or outside its aperture (Figure 4.4G), further suggesting a more deterministic state-action mapping exists only in situations where the target is directly accessible (i.e. within-aperture), as is more often the case for slower targets.

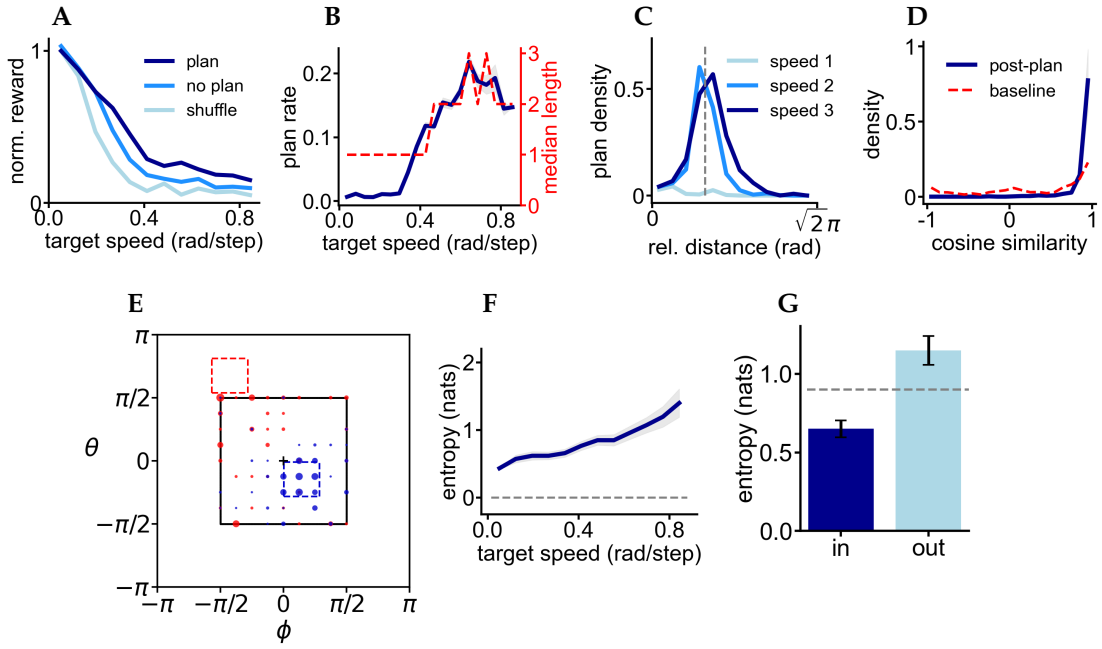


Figure 4.4: **Tracking task statistics.** Macroscopic statistics. **(A)** Mean total reward obtained per trial, binned by target speed in that trial (radians per timestep). Plots are shown for trials recorded with an agent capable of planning (dark blue) as well as data from agents following identical trajectories albeit with planning stripped out post-training (bright blue), and with planning times randomly shuffled throughout the trial (light blue). While in practise the reward signal received is modified ('shaped') across target speed (Section 4.3.2, Appendix C.2.4), reward plots represents unmodified total reward, normalised with respect to the maximum total reward bin (i.e. slowest target speed bin) for the planning-capable agent. **(B)** Average plan rate (dark blue) and median plan length (dashed red) per trial, binned across target speeds. Plan rate is computed as a proportion of decisions made (i.e. not including refractory periods), and median plan length refers to the median length of each planning period. Grey shading indicates standard error of the mean. The increase in planning across target speeds reflects the agent's change in strategy, from directly following the target to leveraging planning to intercept the target at multiple points across each trial. Further planning statistics. **(C)** Planning density as a function of relative distance between the agent and target at the initiation of planning. Data is shown for 3 target speed bins, each normalised over all plans. The peak at higher speeds aligns closely with the agent's aperture range at $\pi/2$ (dashed grey). **(D)** Density of post-plan movements as a function of cosine similarity with the previous plan (i.e. normalised vector describing the previous 'planned trajectory'). We observe post-plan movements (blue) are aligned with previous plan direction to a much greater extent than the baseline (dashed red) of marginal average cosine similarity between all movements and plans. Standard error shown with grey shading. **(E)** Heatmap plot of agent policy for fast and slow targets. Agent's average policy across each of its 81 discrete movement vectors (i.e. not including plans) is shown for both a selection of fast targets (sampled with velocity within the red square) and slow targets (within blue square). Red and blue dots of varying size within the agent's aperture (black square) then correspond to the relative mass of that movement vector under the agent's policy for trials corresponding to red or blue sampled targets respectively. For the slow targets we observe a clustering of movement vectors aligning closely with the target velocity itself, whereas for the fast targets the agent's policy is more spread out, incorporating a wider range of movements. Note sampled target directions are adjusted by a factor of $P = 5$ to account for the refractory period, which reduces effective movement vector magnitude by the same factor. Policy entropy statistics. **(F)** Total post-training policy entropy (both vector chosen and decision to move or plan) as a function of target speed. Grey dashed line indicates minimum theoretical policy entropy of 0 (maximum entropy of $\log(2 \times 81) = 5.09$ is not shown). We observe policy entropy steadily increasing across speed along with standard error (shading), corresponding to a less deterministic policy at higher target speeds. **(G)** Policy entropy across actions selected when the target is within the agent's aperture (in) and outside the aperture (out). These conditions correlate strongly with target speed (low and fast target speeds, respectively). Error bars denote 95% confidence intervals, indicating statistical significance between these two scenarios.

4.4.2 Network Analysis

To understand how the agent's internal dynamics relate to its behavior, we visualise network activity in a lower dimensional space using PCA (Section 2.5). Observing trial trajectories in the subspace spanned by the top 3 PCs (Figure 4.5A) we notice repeating periodic motifs shortly after trial onset, appearing to represent limit cycle dynamics. Binning and averaging trajectories across similar trials we observe these motifs more clearly (Figure 4.5B,C), as well as a clear separation of dynamics across target angle and speed. This suggests the agent not only represents these quantities within its network but also *partitions* its activity accordingly[14]. The periodicity observed can be explained in relation to the P timestep refractory period imposed on movement. Given the agent chooses similar movements every P timesteps when following slow targets, average activity is dominated by periodic motifs reflecting these repeated movements, and other trial paradigms with less consistent behaviours are not observed in this average. This is further demonstrated in the speed-binned plot, where trajectories at lower target speeds show more pronounced periodicity, with such a clear separation across speed (strongly represented in PC1) reinforcing this as an important quantity in dictating agent behaviour.

The correlation between agent activity and target velocity is also unsurprising given this represents the sole task parameter of interest in each trial, hence fully dictating agent behaviour. To further consider the extent to which network activity internalises and *encodes* target velocity, we train linear decoders to predict target velocity at each trial timestep using the top K PCs (Figure 4.5D). We observe decoder performance increasingly outperform the baseline as more PCs are used. Interestingly, while variance explained by our PCs follows a typical power law curve [123], decoder performance plateaus for $K > 10$ (not shown) suggesting target velocity is selectively encoded by the top PCs, further emphasising the importance of this quantity[15]. We also observe strong variation in decoder performance within-trial. For example, performance improves across initial timesteps, suggesting target velocity is rapidly internalised based on initial observations. Additionally, decoder performance fluctuates periodically across the trial. This can once again be explained by the movement refractory period[16] - where performance increases during each $P = 5$ period up to peak performance (minimum error) during each movement timestep. This further suggests an internal representation is refined over the refractory period, suggesting such a period may be utilised by the agent to optimise its policy for the upcoming movement[17].

To consider how the network represents planning, we first observe from a scatter plot of network states (Figure 4.5E) that planning activity occupies a largely distinct subspace of the network compared to movements, indicating the agent effectively partitions its state space of network activity into movement and planning subspaces, reflecting the differential encoding of these behaviours[18]. To further investigate the effect of planning dynamics on resultant behaviour, we perform demixed principal component analysis (dPCA) on the network activity to identify principal components that explain variance solely attributed to the decision to plan or move. By perturbing our network's activity along the direction of the top planning dPC, we observe perturbations beyond a certain point consistently trigger plans (Figure 4.5H)[19]. Furthermore, visualising trajectories in the phase plane spanned by the top dPC for planning and relative target distance respectively (Figure 4.5G), we observe a clear relationship between

[14]Further analysis would be needed, however, to determine if this is a result of learned network activity or simply correlated to the network's input (ie. observations of the moving target).

[15]Specifically the top $K = 10$ PCs explain 70.7% of variance, but decoder improvement using additional PCs is negligible.

[16]The full explanation is slightly more nuanced. Movements and plans can occur at any points in each trial, however when averaging across a large number of trials the most dominant mode consists of a sequence of movements every $P = 5$ timesteps, i.e. for $t = \{0, 5, 10, ...\}$. This mode is present across all bulk average trial analysis, such as in Figure 4.5B-D.

[17]Further work to investigate this could examine the effect of inhibiting network activity during this period.

[18]Close inspection of single trial trajectories additionally show a clear deviation from movement to planning subspaces at the point of planning.

[19]This is unsurprising given the agent's mechanism for triggering plans is presumably related to the row of the readout matrix which maps hidden state to (pre-softmax) planning mass - which the top planning dPC is likely highly correlated to.

activity along these two directions - where relative distance-related activity (i.e. as the target moves *away* from the agent) appears to *lead* planning activity. It can hence be concluded that the trained agent has developed a neural mechanism for plan initiation based significantly on target distance, consistent with task statistics in Figure 4.4 which similarly describe such a correlation.

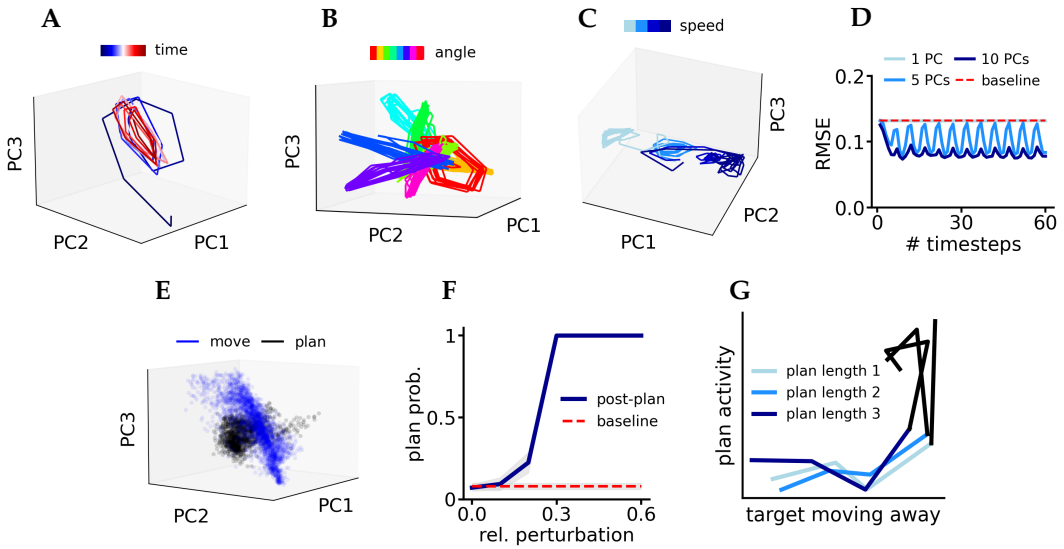


Figure 4.5: **Principal component analysis of the tracking task. (A)** Example single trial trajectory in the subspace spanned by the top 3 PCs. Blue-red colourmap describes trajectory evolution with time, demonstrating periodic, limit cycle behaviour. The period of this behaviour is equal to refractory period length P, where each period reflects similar movements repeated across the trial. **(B)** Average trajectory for each target angle bin, in the same subspace. Each colour denotes average trajectory over all trials with targets in a given angle range - in this case over 8 bins, each spanning $45°$ - where we observe strong separation across target velocity angle. **(C)** Average trajectories binned by target speed for the blue target angle bin from the previous plot. We observe regular, limit cycle dynamics at lower speeds (light blue), with this regularity decreasing as speed increases (darker blue). **(D)** Linear decoder performance for prediction of target velocity from the top K PCs. Each datapoint describes the performance (RMSE) of a separate decoder trained across the top $K = \{1, 5, 10\}$ PCs to predict target velocity from neural activity during the t^{th} timestep. Each decoder was trained via 5-fold cross validation. For $K = 1$ (light blue) we observe decoder performance close to baseline (dashed red). Performance improves past baseline, however, for increasing K, at which point a periodic pattern emerges. This reflects the agent's refractory period, where peak decoder performance (minimum error) corresponds to timesteps where movements are made. **(E)** Scatter plot of movement and planning states in the top 3 PC subspace. The plot is rotated to best demonstrate the separation between movement (blue) and planning (black) states, where we observe activity associated with each occupy largely distinct regions of state space. **(F)** Planning probability as a function of network perturbation magnitude along the top planning demixed principal component (dPC). By extracting the top dPC associated with planning (top PC in the projection explaining variance associated solely with the decision to plan or move) we perturb the network's hidden state by random increments in this direction, where relative perturbation is computed as a proportion of mean network activity magnitude. The resulting curve shows a 'knee' in resultant planning probability, where relative perturbations greater than 0.2 result in planning with probability close to 1. Baseline (dashed red) describes marginal (unperturbed) planning probability, and grey shading indicates standard error. **(G)** Pre-planning network activity in the phase plane spanned by the top demixed planning and distance dPCs. The distance in question is relative distance between agent and target, with distance dPCs then referring to network activity explaining variance associated solely with variation in this distance (i.e. due to the target moving toward or away from the agent). In this projected space (which is non-orthogonal - dPCA imposes no orthogonality constraint between task parameters dPCs), we plot 'snippets' of average network activity for the 4 timesteps preceding plans of length $1:3$. We note network activity along the top distance dPC *leads* activity along the top planning dPC - in other words activity associated with variation in target distance appears to precede that associated with planning.

Chapter 5

Conclusion

5.1 Summary of Results

Chapter 3 considered navigation tasks in periodic angular space in a closed loop setting, where our agent received inputs from visual basis functions tiling the entire space and produced output movements selected from a continuous action space. Enforcing that our agent internally represented egocentric reward location in its network, we pre-trained the agent on a simple navigation task before then considering the network's ability to adapt to a *renavigation* task, which necessitated multiple renavigations to the same reward across a single task. By observing successful macroscopic behaviour and analysing relevant task statistics in the resulting adapted agent, we demonstrated our agent was able to form and incorporate egocentric notions of reward in space to solve the novel renavigation task, generalising the angular navigation behaviour previously learnt to a more complex paradigm - a form of *meta-learning*.

Chapter 4 then considered a tracking task using a modified network model, where our agent obtained visual input from within a limited aperture and selected actions from a discrete action space to track a moving target. This agent was additionally endowed with the option to covertly simulate (plan) a movement, doing so at a faster rate than overt movements. The pre-trained predictive network governing this plan can be considered a separate module, describing a *simulated* environment with which the agent interacts during planning. Our agent once again successfully solves the task, learning a meta-strategy which maximises reward across target speeds: following slow targets, and incorporating planning to intercept faster targets as many times as possible within a trial. Examining the behaviour developed and the trained network dynamics, we comment on the representation of the optimal policy within the agent and the internal neural dynamics associated with its behaviour across different scenarios, including with regards to planning.

5.2 Discussion

The broad aims of this project considered to what extent we could provide insights into the complex sensorimotor transform occurring within the superior colliculus from our meta-learning simulations in periodic egocentric space. This was of course a highly ambitious objective for a work of this nature given the challenges of making links between artificial RL and the brain. Even for sophisticated simulations more advanced than those in this work, such research typically draws parallels based more on the broad principles of neural dynamics recruited across artificial and biological agents (Section 1.3), as opposed to making more concrete predictions. Nevertheless our work produced a number of interesting results

across the simulations considered, and represents a launching point for further work which could provide more direct insights into the myriad open questions involving the SC.

Considering first the renavigation task, we consider such a task to represent largely unconstrained angular movement behaviour learnt by an agent tasked with navigating to a reward in periodic egocentric space, loosely inspired by continuous motor control systems as theorised to govern saccades in the SC. Examining the behavioural statistics obtained from successful task performance, we posit that egocentric notion of reward location is a useful concept to develop by such an agent, and that such a representation can be built by an agent with no explicit periodicity in its network structure (past the input layer[1]). This can be compared with our understanding of the SC's role which is often considered as an angular 'navigation module', representing potential target locations directly on an internal, egocentric spatial map. It can be argued our simulations reinforce the utility of such a neural system created largely for the role of internally representing and mediating movements to egocentric targets.

We next consider in more detail the tracking task with an expanded model featuring discrete saccades, a limited aperture, and the ability to plan. While we once again make any potential links to the SC with caution, the further constrained nature of the task does allow us to make somewhat more specific comments regarding possible neurophysiological links between our results and research objectives. First, we must comment on to what extent our task represents a more plausible paradigm under which to explore SC activity. While ethological tracking behaviour typically consists of smooth pursuit movement in which the target is constantly foveated, studies have shown tracking behaviour of human infants to in fact consist of a series of exogenous[2] re-foveations (saccades) [126]. Hypotheses for this are centred around the idea that smooth pursuit is a cortically-mediated ability, and that until the cortex fully develops such a task is governed instead by subcortical mechanisms (typically associated with more innate behaviours) - in this case the discrete saccadic movements of the SC [127, 128]. With this in mind our task can be considered to model the SC's 'self-contained' approach to such a tracking task, in isolation of any cortical contribution.

Beginning then by examining our agent's policy post-training, we can make a number of deductions. In the simple case of slow-target following our agent implements a largely deterministic (low entropy) policy, saccading directly to the target at each opportunity and drawing comparison to the exogenous, largely innate behaviour described above[3]. In the case of faster targets however, the model-based behaviour observed represents more complex dynamics. As discussed in our analysis, the agent instead chooses between movement and planning based on the situation with a degree of stochasticity[4] model-free approach to solving the task as for slow targets. It could therefore be suggested an isolated SC-like network may perform poorly in such a task, without either the ability to plan movements or the additional decision-making ability provided by the cortex.

While its unreasonable to draw direct connections between our agent's policy and the motor code implemented in the deep SC layers, our results in the slow target paradigm at the very least provide a degree of justification for the the existence of direct retinal pathways to the SC - something plausibly important to enable the rapid, direct movements to salient visual stimuli observed both experimentally and in our simulations. A modular design can similarly be considered an efficient neural implementation to mediate this behaviour, facilitating direct movement to motor regions associated with equivalent incident

[1] While we enforce periodicity via the agent's input basis functions, there is no explicit periodicity in the network's connectivity - as is, for example, typically the case for ring-based models of the visual system [103, 124, 125].

[2] I.e triggered by external stimuli as opposed to internal decision (endogenous).

[3] To further investigate this comparison, we could decompose our network [129] to explore if there exists embedded feedforward pathways between visual input and motor output (as would conceivably mediate such behaviours ethologically).

[4] It can be argued some degree of stochasticity post-training is unsurprising given the randomness inherent in the sampling-based training of the agent, and suggests multiple actions may be similarly optimal at each point.

sensory inputs via the integration of these signals within a single anatomical 'module'. Additionally, the strong modulation of activity across the top principal component - which appears to approximately encode target speed - suggests we could similarly expect to observe stimuli movement speed represented by one of the top principal components of SC population activity.

Finally, the utilisation of planning under the trained agent's policy reinforces the idea of internal simulation of actions as useful for an agent tasked with making complex context-dependent decisions which could benefit from model-based simulation - specifically when such simulation has a lower cost than overt actions. Such a paradigm similarly emphasises the importance of recurrent connectivity in our agent's network: while slow-target 'following' consists broadly of a mapping from visual input to motor output that could be fully characterised by a feedforward network, the learning of temporal dependencies during planning in the 'interception' paradigm necessitates some form of recurrence.

Our experimental data of course represents vastly insufficient evidence to comment on any theories of self-contained planning within the SC itself. Existing evidence of this is minimal as-is, and there are suggestions the observed corollary discharge from the SC during movements - which in part motivate this theory - are more plausibly for the purpose of informing other brain areas of impending movements [130] (for example signalling to cortex that upcoming movements originate from the agent itself, as opposed to external motion). In addition, motor planning is typically considered a complex process recruiting multiple neural regions[5], further casting doubt on the theory such a process could be self-contained in the SC. Any evidence for such a mechanism would therefore need to go beyond merely demonstrating the usefulness of an analogous system in a toy simulation such as ours.

5.3 Future Work

While the models considered in this work draw inspiration from the SC, they are highly simplified in comparison. Compared to the complex, layered anatomy of the SC which features topographic sensory and motor representations, our models consider only a single combined network with no explicit topographic organisation. The sensorimotor transform implemented by the SC is hence vastly simplified within our models, which also consider a limited number of visual basis functions[6] with no retinotopic mapping, and use simple models of output movement which in both cases lack the online motor control typically believed to govern saccadic movements.

To enable more direct comparison with experimental data and hence further justify any inferences made, we propose a model for further work that explicitly incorporates retinotopic visual mapping, now capable of extracting both spatial and temporal information from the environment to make decisions. This model is built on a convolutional GRU (convGRU) [134], which modifies the previous GRU model to now consider our network's hidden state as a 2 dimensional *array* which evolves according to a series of convolution operations, where such a state can be considered to reflect topographic activations on the *motor map* in the SC's deep layers. The matrix-vector multiplications from our previous GRU model are

[5]Including but not limited to Primary Motor Cortex (M1), Premotor Cortex (PMC) and Supplementary Motor Area (SMA).

[6]We use of order $10^2 - 10^3$ such functions in our models, compared to the order $10^4 - 10^6$ number of retinal ganglion cells recruited during vision across humans, other primates, and rodents [131–133].

hence now replaced with (kernel) matrix-matrix convolutions as

$$\mathbf{Z}_t = \sigma(\mathbf{W}_z * \mathbf{X}_t + \mathbf{U}_z * \mathbf{H}_{t-1} + \mathbf{B}_z) \tag{5.1}$$

$$\mathbf{R}_t = \sigma(\cdot) \tag{5.2}$$

$$\hat{\mathbf{H}}_t = \tanh(\cdot) \tag{5.3}$$

$$\mathbf{H}_t = \mathbf{Z}_t \odot \mathbf{H}_{t-1} + (1 - \mathbf{Z}_t) \odot \hat{\mathbf{H}}_t \tag{5.4}$$

where $\mathbf{H}_t$ now denotes a hidden state array representing the SC motor layers and $\mathbf{X}_t$ are again basis function activations in egocentric space, now retinotopically mapped into the network. $\mathbf{W}$ and $\mathbf{U}$ are input and recurrent convolution kernels, and $\mathbf{B}$ is a bias term. The input term $\mathbf{W}_z * \mathbf{X}_t$ can hence be considered to reflect the SC's *visual map* in the superficial layers, subsequently transformed to the motor layer representation in $\mathbf{H}_t$.

Such a model therefore explicitly captures both spatial information - from the convolution operations on its retinotopically arranged input and hidden state - as well as temporal dependencies via its recurrent structure[7]. We therefore effectively model the SC as a neural module which extracts features from the timeseries of its visual input to build an internal topographic representation which enables it to make optimal saccadic movements to solve a given task. This model can also be extended in a hierarchical fashion to an arbitrary number of layers L, where each receives as input the hidden state of the preceding layer (termed the *feature map* in CNN literature), i.e. $\mathbf{X}_t^{(l)} = \mathbf{H}_t^{(l-1)}$. This structure enables hierarchical representations of increasingly abstract spatio-temporal features - reflecting the hierarchical nature of feature detection in the visual system [137–139] - where each evolving hidden state $\mathbf{H}_t^{(l)}$ (similarly termed a *percept*) encapsulates the model's learned patterns at varying levels of abstraction. Alternatively we could expand model complexity laterally - feeding our input through multiple 'channels' each associated with a separate convolution operations, the outputs of which are then combined downstream[8].

During model training the input and recurrent convolution operators can be initialised randomly as before, or alternatively assigned prior weights which attempt to model their respective ethological roles. For example $\mathbf{W}$ could be a feature detecting kernel - where if our model is extended to multiple channels or layers, each $\mathbf{W}$ could correspond to a different spatial frequency. Similarly, $\mathbf{U}$ could be a Difference-of-Gaussians kernel to simulate the short range excitation and surround inhibition contributions of surrounding neural activity [142, 143], reflecting the *winner-takes-all* mechanism believed to be recruited by the SC in forming its motor code.

For the motor readout, we can leverage existing models from the literature to produce a saccade vector $\vec{S}(t)$ based on the spatial activations in our 'motor map' $\mathbf{H}_t$. By obtaining our saccade vector in this way we force our model to represent motor activity in a form that reflects our understanding of motor-level neural activity in the SC. To decode $\vec{S}(t)$ we can consider use of both a 'static' or 'dynamic' saccade model (Section 1.2, Appendix A). For static models, we instantaneously decode $\vec{S}(t)$ via a vector sum or average (or with probabilistic techniques, if there exist multiple competing activity loci [32, 33]), then implement this either with a downstream motor loop, or as a ballistic approximation[9]. Dynamic saccade models could similarly be incorporated, where the timeseries of activity then governs a time-varying saccadic movement (Appendix A.2-3).

[7]Research suggests both spatial and temporal aspects of visual processing to be similarly important for extracting visual information [135, 136] - it can be argued explicitly modelling of both aspects together constitutes a more realistic model of the visual system.

[8]This reflects the hierarchical visual system believed to be implemented in a range of animals, where inputs undergo a number of simultaneous filtering operations ('channels'), followed by downstream 'pooling' to achieve spatial invariance and dimensionality reduction [139–141]. This process continues in a hierarchical way, enabling the representation of increasingly complex objects.

[9]I.e. $x_{t+1}^{i,j} = x_t^{i-\vec{S}_{t,y},j+\vec{S}_{t,x}}$.

An important benefit of representing activity in this topographic form is that activity in the 'motor map' of our model can be directly visualised on the SC's anatomical structure, by considering an empirical parametric mapping from each egocentric (visual) location (x, y) within $\mathbf{H}_t$[10] to an anatomical SC location (x', y') [25]

$$x' \leftarrow B_x \ln\left(\frac{\sqrt{x^2 + y^2 + 2Ax + A^2}}{A}\right) \tag{5.5}$$

$$y' \leftarrow B_y \arctan\left(\frac{y}{x + A}\right) \tag{5.6}$$

as in Figure 5.1. By then building on our previous RL framework to train our agent to perform simple

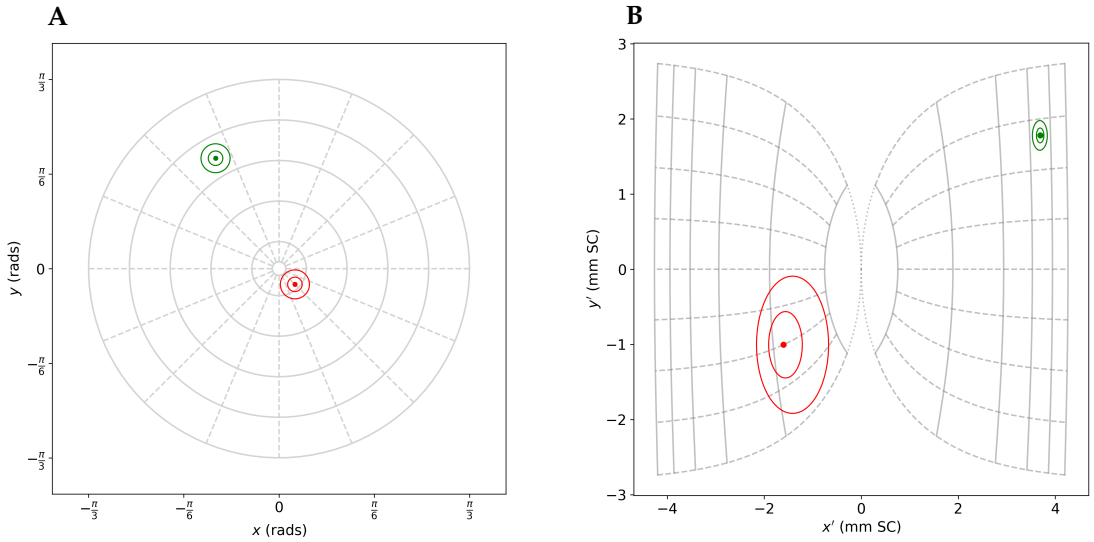


Figure 5.1: **Parametric mapping from visual to retinotopic coordinates. (A)** Example scene in egocentric visual coordinates. Two targets are shown both near-foveal (red) and afoveal (green). Axes $x - y$ reflect notation used previously to describe longitudinal and latitudinal angular rotations ϕ and θ. In this case, the agent has an observable range of $\pm\pi/3$. Grey lines describe locations of constant eccentricity (solid) and angle (dotted). The inner-most circle, reflecting near-foveal information, is left blank for visual clarity. **(B)** Identical scene in retinotopic coordinates. Based on the parametric mapping in eqs. (5.5-5.6), we consider the same scene now retinotopically mapped onto the SC's surface. While the afoveal target appears largely unchanged we observe strong distortion for the foveal target, reflecting the logarithmic nature of our warping based on target eccentricity. This figure uses example parameters obtained from *in-vivo* experimentation on monkeys, where $A = 3$, $B_x = 1.4\,\text{mm}$, $B_y = 1.8\,\text{mm/deg}$ [144].

saccade-based tasks, we can observe the dynamics and activity patterns recruited with respect to their equivalent representations on the surface of the SC. Where possible, we can then make predictions or comparisons with *in-vivo* recordings from animals performing analogous tasks. With this in mind, we could train our agent on tasks representing common experimental paradigms to record from the SC:

- Reflexive Saccade: Our agent must saccade immediately to a target as it enters its visual periphery [145–147].
- Reflexive Anti-saccade: Similar to the above but the agent must instead saccade in the opposite direction, reflecting 'avoidance' behaviour [148–150].
- Visually Guided Saccade: After an initial period of fixation a visual stimulus is presented in the periphery, which the agent must then saccade to [151–153].

[10] As before, we use (x, y) coordinates to represent egocentric angles (ϕ, θ). By considering our points in this form we greatly simplify the convolution operations involved, with the caveat we must then consider a 'square' window of egocentric space.

Specifically, such tasks recruit reflexive behaviours considered largely innate and therefore largely 'self-contained' by the SC, accomplished with minimal cortical input. This is important as the model described is considered in isolation of any cortical (or other brain region) input, and so such tasks allow us to draw more direct comparisons to experimental data and hence more confident inferences post-training[11].

Furthermore, we could meta-train across the first two tasks using different target representations for the saccade and anti-saccade respectively, exploring how each target is transformed into separate representation for pursuing or avoidance respectively (drawing parallels to ethological hunting or escape behaviours). Additionally we could examine dynamics during the initial fixation period for the visually-guided saccade task, to consider how the upcoming movement is represented in the pre-saccade period. For all tasks, we could consider under what conditions our agent's 'motor code' mirrors deep layer activity in comparable experimental data, as well as to what extent this aligns with theories of dynamic 'motor error' representation in the SC which situate it directly in the motor control feedback loop.

In summary, while the RL paradigms in this work demonstrate interesting results regarding the behaviour and dynamics recruited by an artificial agent to solve simple tasks in egocentric angular space, these results are notable predominantly for their role in demonstrating the potential of this paradigm as a foundation for further experimentation, as opposed to any direct predictions made regarding the neural basis of sensorimotor transform in the SC. One such example of potential further experimentation is given above, where we consider an expanded, retinotopic model trained under saccadic tasks designed to enable more direct comparative analysis with experimental data. Such work represents just one of many techniques recruited in the path to understanding the SC - itself just a small step in the broader journey toward unravelling the full complexities of the brain.

[11]For more complex, cortically-mediated tasks, our 'isolated' model of the SC would produce less meaningful comparisons with experimental data.

References

[1] S. Crochet, S. Lee, and C. C. H. Petersen, "Neural Circuits for Goal-Directed Sensorimotor Transformations," *Trends Neurosci.*, vol. 42, no. 1, pp. 66–77, 2019. DOI: 10.1016/j.tins.2018.08.011.

[2] W. R. Hess, S. Buergi, and V. Bucher, "[Motor function of the tectal and tegmental area]," *Monatsschr. Psychiatr. Neurol.*, vol. 112, no. 1-2, pp. 1–52, 1946.

[3] K. P. Schaefer, "Unit analysis and electrical stimulation in the optic tectum of rabbits and cats," *Brain Behav. Evol.*, vol. 3, no. 1, pp. 222–240, 1970. DOI: 10.1159/000125475.

[4] T. Wheatcroft, A. B. Saleem, and S. G. Solomon, "Functional Organisation of the Mouse Superior Colliculus," *Front. Neural Circuits*, vol. 16, p. 792 959, 2022. DOI: 10.3389/fncir.2022.792959.

[5] S. Ito and D. A. Feldheim, "The Mouse Superior Colliculus: An Emerging Model for Studying Circuit Formation and Function," *Front. Neural Circuits*, vol. 12, p. 327 065, 2018. DOI: 10.3389/fncir.2018.00010.

[6] X. Liu, H. Huang, T. P. Snutch, P. Cao, L. Wang, and F. Wang, "The Superior Colliculus: Cell Types, Connectivity, and Behavior.," *Neuroscience Bulletin*, vol. 38, no. 12, pp. 1519–1541, 2022. DOI: 10.1007/s12264-022-00858-1.

[7] E. Adamiik, "Uber angeborene und erworbene," *Albrecht V. Graefes Archiv für Ophthalmologie*, vol. 18, pp. 153–156, 1872.

[8] D. A. Robinson, "Eye movements evoked by collicular stimulation in the alert monkey," *Vision Res.*, vol. 12, no. 11, pp. 1795–1808, 1972. DOI: 10.1016/0042-6989(72)90070-3.

[9] D. L. Sparks, R. Holland, and B. L. Guthrie, "Size and distribution of movement fields in the monkey superior colliculus," *Brain Res.*, vol. 113, no. 1, pp. 21–34, 1976. DOI: 10.1016/0006-8993(76)90003-2.

[10] X. Wang *et al.*, "Computational neuroscience: a frontier of the 21st century," *Natl. Sci. Rev.*, vol. 7, no. 9, pp. 1418–1422, 2020. DOI: 10.1093/nsr/nwaa129.

[11] A. P. Davison and S. Appukuttan, "Computational neuroscience: A faster way to model neuronal circuitry," *eLife*, vol. 11, e84463, 2022. DOI: 10.7554/eLife.84463.

[12] R. D. Zubricky and J. M. Das, "Neuroanatomy, Superior Colliculus," in *StatPearls*, StatPearls Publishing, 2022.

[13] T. Isa, E. Marquez-Legorreta, S. Grillner, and E. K. Scott, "The tectum/superior colliculus as the vertebrate solution for spatial sensory integration and action," *Curr. Biol.*, vol. 31, no. 11, pp. 741–762, 2021. DOI: 10.1016/j.cub.2021.04.001.

[14] M. Driscoll and P. Tadi, "Neuroanatomy, inferior colliculus," *StatPearls*, 2023, Treasure Island (FL): StatPearls Publishing.

[15] J. E. Hyde and R. G. Eason, "Characteristics of ocular movements evoked by stimulation of brainstem of cat," *J. Neurophysiol.*, vol. 22:666-78. 1959. DOI: 10.1152/jn.1959.22.6.666.

[16] S. Bernheimer, "Experimentelle studien zur kenntniss der bahnen der synergischen augenbewegungen beim affen und der beziehungen der vierhügelzu denselben," *Sitzber Deut Akad Wiss*, vol. 180, no. 3, pp. 299–317, 1899.

[17] M. Ronald E. Myers, "Visual Deficits After Lesions of Brain Stem Tegmentum in Cats," *Arch. Neurol.*, vol. 11, no. 1, pp. 73–90, 1964. DOI: 10.1001/archneur.1964.00460190077006.

[18] T. Pasik, P. Pasik, and M. B. Bender, "The Superior Colliculi and Eye Movements: An Experimental Study in the Monkey," *Archives of Neurology*, vol. 15, no. 4, pp. 420–436, 1966. DOI: 10.1001/archneur.1966.00470160086012.

[19] P. J. May, "The mammalian superior colliculus: laminar structure and connections," *Prog. Brain*

Res., vol. 151:321-78. No. ; Prog, 2006. DOI: 10.1016/S0079-6123(05)51011-2.

[20] A. Rees, "Sensory maps: Aligning maps of visual and auditory space," *Curr. Biol.*, vol. 6, no. 8, pp. 955–958, 1996. DOI: 10.1016/S0960-9822(02)00637-1.

[21] J. W. Triplett, A. Phan, J. Yamada, and D. A. Feldheim, "Alignment of Multimodal Sensory Input in the Superior Colliculus through a Gradient-Matching Mechanism," *J. Neurosci.*, vol. 32, no. 15, p. 5264, 2012. DOI: 10.1523/JNEUROSCI.0240-12.2012.

[22] Q. Wang and A. Burkhalter, "Stream-related preferences of inputs to the superior colliculus from areas of dorsal and ventral streams of mouse visual cortex," *J. Neurosci.*, vol. 33, no. 4, pp. 1696–1705, 2013. DOI: 10.1523/JNEUROSCI.3067-12.2013.

[23] P. H. Schiller and M. Stryker, "Single-unit recording and stimulation in superior colliculus of the alert rhesus monkey," *J. Neurophysiol.*, vol. 35, no. 6, pp. 915–924, 1972. DOI: 10.1152/jn.1972.35.6.915.

[24] T. J. Anastasio and P. E. Patton, "Analysis and Modeling of Multisensory Enhancement in the Deep Superior Colliculus," in *The Handbook of Multisensory Processes*, The MIT Press, 2004, ISBN: 9780262269704. DOI: 10.7551/mitpress/3422.003.0021.

[25] F. P. Ottes, J. A. Van Gisbergen, and J. J. Eggermont, "Visuomotor fields of the superior colliculus: a quantitative model," *Vision Res.*, vol. 26, no. 6, pp. 857–873, 1986. DOI: 10.1016/0042-6989(86)90144-6.

[26] Z. M. Hafed, C. Chen, and X. Tian, "Vision, Perception, and Attention through the Lens of Microsaccades: Mechanisms and Implications," *Front. Syst. Neurosci.*, vol. 9, 2015. DOI: 10.3389/fnsys.2015.00167.

[27] D. P. Munoz and R. H. Wurtz, "Role of the rostral superior colliculus in active visual fixation and execution of express saccades," *J. Neurophysiol.*, vol. 67, no. 4, pp. 1000–1002, 1992. DOI: 10.1152/jn.1992.67.4.1000.

[28] Z. M. Hafed, C.-Y. Chen, X. Tian, M. P. Baumann, and T. Zhang, "Active vision at the foveal scale in the primate superior colliculus," *J. Neurophysiol.*, vol. 125, no. 4, pp. 1121–1138, 2021. DOI: 10.1152/jn.00724.2020.

[29] E. M. Klier, H. Wang, and J. D. Crawford, "The superior colliculus encodes gaze commands in retinal coordinates," *Nat. Neurosci.*, vol. 4, no. 6, pp. 627–632, 2001. DOI: 10.1038/88450.

[30] E. Klier, H. Wang, and J. Crawford, "Three-dimensional eye-head coordination is implemented downstream from the superior colliculus," *J Neurophysiol*, vol. 89, no. 5, pp. 2839–2853, 2003. DOI: 10.1152/jn.00763.2002.

[31] R. A. Tikidji-Hamburyan, T. A. El-Ghazawi, and J. W. Triplett, "Novel Models of Visual Topographic Map Alignment in the Superior Colliculus," *PLoS Comput. Biol.*, vol. 12, no. 12, e1005315, 2016. DOI: 10.1371/journal.pcbi.1005315.

[32] B. Kim and M. A. Basso, "A Probabilistic Strategy for Understanding Action Selection," *J. Neurosci.*, vol. 30, no. 6, pp. 2340–2355, 2010. DOI: 10.1523/JNEUROSCI.1730-09.2010.

[33] L. Masullo, L. Mariotti, N. Alexandre, P. Freire-Pritchett, J. Boulanger, and M. Tripodi, "Genetically Defined Functional Modules for Spatial Orienting in the Mouse Superior Colliculus," *Curr. Biol.*, vol. 29, no. 17, 2892–2904.e8, 2019. DOI: 10.1016/j.cub.2019.07.083.

[34] D. P. Munoz and P. J. Istvan, "Lateral inhibitory interactions in the intermediate layers of the monkey superior colliculus," *J. Neurophysiol.*, vol. 79, no. 3, pp. 1193–1209, 1998. DOI: 10.1152/jn.1998.79.3.1193.

[35] T. P. Trappenberg, M. C. Dorris, D. P. Munoz, and R. M. Klein, "A model of saccade initiation based on the competitive integration of exogenous and endogenous signals in the superior colliculus," *J. Cognit. Neurosci.*, vol. 13, no. 2, pp. 256–271, 2001. DOI: 10.1162/089892901564306.

[36] J. Essig, J. B. Hunt, and G. Felsen, "Inhibitory neurons in the superior colliculus mediate selection of spatially-directed movements," *Commun. Biol.*, vol. 4, no. 719, pp. 1–14, 2021. DOI: 10.1038/s42003-021-02248-1.

[37] C. Gehr, J. Sibille, and J. Kremkow, "Retinal input integration in excitatory and inhibitory neurons in the mouse superior colliculus in vivo," *eLife*, 2023. DOI: 10.7554/eLife.88289.

[38] J. M. Findlay, "Global visual processing for saccadic eye movements," *Vision Res.*, vol. 22, no. 8, pp. 1033–1045, 1982. DOI: 10.1016/0042-6989(82)90040-2.

[39] F. P. Ottes, J. A. Van Gisbergen, and J. J. Eggermont, "Metrics of saccade responses to visual double stimuli: two different modes," *Vision Res.*, vol. 24, no. 10, pp. 1169–1179, 1984. DOI: 10.1016/0042-6989(84)90172-x.

[40] M. C. Helms, G. Ozen, and W. C. Hall, "Organization of the intermediate gray layer of the superior colliculus. I. Intrinsic vertical connections," *J. Neurophysiol.*, vol. 91, no. 4, pp. 1706–1715, 2004. DOI: 10.1152/jn.00705.2003.

[41] J. A. Edelman and E. L. Keller, "Dependence on target configuration of express saccade-related activity in the primate superior colliculus," *J. Neurophysiol.*, vol. 80, no. 3, pp. 1407–1426, 1998. DOI: 10.1152/jn.1998.80.3.1407.

[42] C. Lee, W. H. Rohrer, and D. L. Sparks, "Population coding of saccadic eye movements by neurons in the superior colliculus," *Nature*, vol. 332, no. 6162, pp. 357–360, 1988. DOI: 10.1038/332357a0.

[43] M. M. G. Walton, D. L. Sparks, and N. J. Gandhi, "Simulations of Saccade Curvature by Models That Place Superior Colliculus Upstream From the Local Feedback Loop," *J. Neurophysiol.*, vol. 93, no. 4, p. 2354, 2005. DOI: 10.1152/jn.01199.2004.

[44] D. L. Sparks and L. E. Mays, "Signal transformations required for the generation of saccadic eye movements," *Annu. Rev. Neurosci.*, vol. 13:309-36. No. ; Annu, 1990. DOI: 10.1146/annurev.ne.13.030190.001521.

[45] M. J. Nichols and D. L. Sparks, "Component stretching during oblique stimulation-evoked saccades: the role of the superior colliculus," *J. Neurophysiol.*, vol. 76, no. 1, pp. 582–600, 1996. DOI: 10.1152/jn.1996.76.1.582.

[46] A. P. Georgopoulos, A. B. Schwartz, and R. E. Kettner, "Neuronal population coding of movement direction," *Science*, vol. 233, no. 4771, pp. 1416–1419, 1986. DOI: 10.1126/science.3749885.

[47] J. A. Van Gisbergen, A. J. Van Opstal, and A. A. Tax, "Collicular ensemble coding of saccades based on vector summation," *Neuroscience*, vol. 21, no. 2, pp. 541–555, 1987. DOI: 10.1016/0306-4522(87)90140-0.

[48] H. H. L. M. Goossens and A. J. Van Opstal, "Dynamic ensemble coding of saccades in the monkey superior colliculus," *J. Neurophysiol.*, vol. 95, no. 4, pp. 2326–2341, 2006. DOI: 10.1152/jn.00889.2005.

[49] A. J. van Opstal and H. H. L. M. Goossens, "Linear ensemble-coding in midbrain superior colliculus specifies the saccade kinematics," *Biol. Cybern.*, vol. 98, no. 6, pp. 561–577, 2008. DOI: 10.1007/s00422-008-0219-z.

[50] R. Jürgens, W. Becker, and H. H. Kornhuber, "Natural and drug-induced variations of velocity and duration of human saccadic eye movements: evidence for a control of the neural pulse generator by local feedback," *Biol. Cybern.*, vol. 39, no. 2, pp. 87–96, 1981. DOI: 10.1007/BF00336734.

[51] C. Quaia, P. Lefèvre, and L. M. Optican, "Model of the control of saccades by superior colliculus and cerebellum," *J. Neurophysiol.*, vol. 82, no. 2, pp. 999–1018, 1999. DOI: 10.1152/jn.1999.82.2.999.

[52] L. E. Mays and D. L. Sparks, "Dissociation of visual and saccade-related responses in superior colliculus neurons," *J. Neurophysiol.*, vol. 43, no. 1, pp. 207–232, 1980. DOI: 10.1152/jn.1980.43.1.207.

[53] O. Hirosaka and R. Wurtz, "Modification of saccadic eye movements by gaba-related substances. i. effect of muscimol and bicuculline in monkey superior colliculus," *J. Neurophysiol.*, vol. 53, pp. 266–289, 1985, 1985a.

[54] A. J. van Opstal and J. A. van Gisbergen, "A model for collicular efferent mechanisms underlying the generation of saccades," *Brain Behav. Evol.*, vol. 33, no. 2-3, pp. 90–94, 1989. DOI: 10.1159/000115906.

[55] T. R. Stanford, E. G. Freedman, and D. L. Sparks, "Site and parameters of microstimulation: Evidence for independent effects on the properties of saccades evoked from the primate superior colliculus," *J. Neurophysiol.*, vol. 76, no. 5, pp. 3360–81, 1996. DOI: 10.1152/jn.1996.76.5.3360.

[56] W. Hall and A. Moschovakis, "Commands for coordinated eye and head movements," in *The Superior Colliculus: New Approaches for Studying Sensorimotor Integration*, CRC Press LLC, 2003, pp. 303–318.

[57] D. M. Waitzman, T. P. Ma, L. M. Optican, and R. H. Wurtz, "Superior colliculus neurons mediate the dynamic characteristics of saccades," *J. Neurophysiol.*, vol. 66, no. 5, pp. 1716–1737, 1991. DOI: 10.1152/jn.1991.66.5.1716.

[58] D. P. Munoz, D. Pelisson, and D. Guitton, "Movement of Neural Activity on the Superior Colliculus Motor Map During Gaze Shifts," *Science*, vol. 251, no. 4999, pp. 1358–1360, 1991. DOI: 10.1126/science.2003221.

[59] J. H. Kaas, "Evolution of columns, modules, and domains in the neocortex of primates," *Proc.*

Natl. Acad. Sci. U.S.A., vol. 109, no. Suppl 1, p. 10 655, 2012. DOI: 10.1073/pnas.1201892109.

[60] R.-B. Illing, "Chapter 2 the mosaic architecture of the superior colliculus," in *Extrageniculostriate Mechanisms Underlying Visually-Guided Orientation Behavior*, ser. Progress in Brain Research, M. Norita, T. Bando, and B. E. Stein, Eds., vol. 112, Elsevier, 1996, pp. 17–34. DOI: https://doi.org/10.1016/S0079-6123(08)63318-X.

[61] G. Chevalier and S. Mana, "Honeycomb-like structure of the intermediate layers of the rat superior colliculus, with additional observations in several other mammals: AChE patterning," *J. Comp. Neurol.*, vol. 419, no. 2, pp. 137–153, 2000. DOI: 10.1002/(sici)1096-9861(20000403)419:2<137::aid-cne1>3.0.co;2-6.

[62] S. Mana and G. Chevalier, "The fine organization of nigro-collicular channels with additional observations of their relationships with acetylcholinesterase in the rat," *Neuroscience*, vol. 106, no. 2, pp. 357–374, 2001. DOI: 10.1016/s0306-4522(01)00283-4.

[63] C. A. Duan, Y. Pan, G. Ma, T. Zhou, S. Zhang, and N. Xu, "A cortico-collicular pathway for motor planning in a memory-dependent perceptual decision task," *Nat. Commun.*, vol. 12, no. 2727, pp. 1–16, 2021. DOI: 10.1038/s41467-021-22547-9.

[64] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Mach. Learn.*, vol. 3, no. 1, pp. 9–44, 1988. DOI: 10.1007/BF00115009.

[65] W. Schultz, P. Dayan, and P. R. Montague, "A neural substrate of prediction and reward," *Science*, vol. 275, no. 5306, pp. 1593–1599, 1997. DOI: 10.1126/science.275.5306.1593.

[66] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.

[67] K. M. Crisp and K. J. Muller, "A 3-Synapse Positive Feedback Loop Regulates the Excitability of an Interneuron Critical for Sensitization in the Leech," *J. Neurosci.*, vol. 26, no. 13, p. 3524, 2006. DOI: 10.1523/JNEUROSCI.3056-05.2006.

[68] B. Or, "The Exploding and Vanishing Gradients Problem in Time Series," *Medium*, 2023. [Online]. Available: https://towardsdatascience.com/the-exploding-and-vanishing-gradients-problem-in-time-series-6b87d558d22.

[69] S. Hochreiter and J. Schmidhuber, "Long Short-term Memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–80, 1997. DOI: 10.1162/neco.1997.9.8.1735.

[70] K. Cho *et al.*, "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation," *arXiv*, 2014. DOI: 10.48550/arXiv.1406.1078.

[71] A. Zhang, Z. C. Lipton, M. Li, and A. J. Smola, "Dive into Deep Learning," *arXiv*, 2021. DOI: 10.48550/arXiv.2106.11342.

[72] J. X. Wang *et al.*, "Learning to reinforcement learn," *arXiv*, 2016. DOI: 10.48550/arXiv.1611.05763.

[73] Y. Duan, J. Schulman, X. Chen, P. L. Bartlett, I. Sutskever, and P. Abbeel, "RL^2: Fast Reinforcement Learning via Slow Reinforcement Learning," *arXiv*, 2016. DOI: 10.48550/arXiv.1611.02779.

[74] J. X. Wang *et al.*, "Prefrontal cortex as a meta-reinforcement learning system," *Nat. Neurosci.*, vol. 21, pp. 860–868, 2018. DOI: 10.1038/s41593-018-0147-8.

[75] A. Gupta, R. Mendonca, Y. Liu, P. Abbeel, and S. Levine, "Meta-Reinforcement Learning of Structured Exploration Strategies," *arXiv*, 2018. DOI: 10.48550/arXiv.1802.07245.

[76] M. Botvinick, S. Ritter, J. X. Wang, Z. Kurth-Nelson, C. Blundell, and D. Hassabis, "Reinforcement Learning, F. and Slow," *Trends in Cognitive Sciences*, vol. 23, no. 5, pp. 408–422, 2019. DOI: 10.1016/j.tics.2019.02.006.

[77] R. Brette, "Philosophy of the spike: Rate-based vs. spike-based theories of the brain," *Frontiers in Systems Neuroscience*, vol. 9, 2015. DOI: 10.3389/fnsys.2015.00151.

[78] B. Yin, F. Corradi, and S. M. Bohté, "Effective and efficient computation with multiple-timescale spiking recurrent neural networks," in *International Conference on Neuromorphic Systems 2020*, ser. ICONS 2020, Association for Computing Machinery, 2020, ISBN: 9781450388511. DOI: 10.1145/3407197.3407225.

[79] Y. Li, R. Kim, and T. J. Sejnowski, "Learning the Synaptic and Intrinsic Membrane Dynamics Underlying Working Memory in Spiking Neural Network Models," *Neural Computation*, vol. 33, no. 12, pp. 3264–3287, 2021. DOI: 10.1162/neco_a_01409.

[80] D. Bahdanau, K. Cho, and Y. Bengio, *Neural machine translation by jointly learning to align and translate*, 2016.

[81] D. P. Kingma and J. Ba, *Adam: A method for stochastic optimization*, 2017.

[82] T. Schaul and Y. LeCun, *Adaptive learning rates and parallelization for stochastic, sparse, non-smooth gradients*, 2013.

[83] S. Ruder, "An overview of gradient descent optimization algorithms," *arXiv*, 2016. DOI: 10.48550/arXiv.1609.04747.

[84] C. C. Margossian, "A Review of automatic differentiation and its efficient implementation," *arXiv*, 2018. DOI: 10.1002/WIDM.1305.

[85] A. Griewank and A. Walther, *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation*, Second. Society for Industrial and Applied Mathematics, 2008, ch. 3.3, pp. 44–50, ISBN: 0898716594.

[86] S. A. Frank, "Automatic differentiation and the optimization of differential equation models in biology," *Frontiers in Ecology and Evolution*, vol. 10, 2022. DOI: 10.3389/fevo.2022.1010278.

[87] A. G. Baydin, B. A. Pearlmutter, A. A. Radul, and J. M. Siskind, *Automatic differentiation in machine learning: A survey*, 2018.

[88] A. J. Robinson and F. Fallside, "The utility driven dynamic error propagation network," Engineering Department, Cambridge University, Cambridge, UK, Tech. Rep. CUED/F-INFENG/TR.1, 1987.

[89] M. Mozer, "A focused backpropagation algorithm for temporal pattern recognition," *Complex Systems*, vol. 3, pp. 349–381, 1989.

[90] user: guest_blog, "Baseline for Policy Gradients that All Deep Learning Enthusists Must Know," *Analytics Vidhya*, 2020. [Online]. Available: https://www.analyticsvidhya.com/blog/2020/11/baseline-for-policy-gradients.

[91] R. J. Williams, "Toward a theory of reinforcement-learning connectionist systems," Northeastern University, College of Computer Science, Tech. Rep. NU-CCS-88-3, 1988.

[92] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Mach. Learn.*, vol. 8, no. 3, pp. 229–256, 1992. DOI: 10.1007/BF00992696.

[93] J. Schmidhuber, "Learning to control fast-weight memories: An alternative to dynamic recurrent networks," *Neural Computation*, vol. 4, no. 1, pp. 131–139, 1992. DOI: 10.1162/neco.1992.4.1.131.

[94] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, *Proximal policy optimization algorithms*, 2017.

[95] J. Schulman, S. Levine, P. Moritz, M. I. Jordan, and P. Abbeel, "Trust Region Policy Optimization," *arXiv*, 2015. DOI: 10.48550/arXiv.1502.05477.

[96] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-Dimensional Continuous Control Using Generalized Advantage Estimation," *arXiv*, 2015. DOI: 10.48550/arXiv.1506.02438.

[97] S. Huang, R. F. J. Dossa, A. Raffin, A. Kanervisto, and W. Wang, "The 37 implementation details of proximal policy optimization," in *ICLR Blog Track*, 2022. [Online]. Available: https://iclr-blog-track.github.io/2022/03/25/ppo-implementation-details/.

[98] C. Yu *et al.*, *The surprising effectiveness of ppo in cooperative, multi-agent games*, 2022.

[99] W. Brendel, R. Romo, and C. K. Machens, "Demixed principal component analysis," in *Advances in Neural Information Processing Systems*, J. Shawe-Taylor, R. Zemel, P. Bartlett, F. Pereira, and K. Weinberger, Eds., vol. 24, Curran Associates, Inc., 2011.

[100] S. Saxena, A. A. Russo, J. Cunningham, and M. M. Churchland, "Motor cortex activity across movement speeds is predicted by network-level strategies for generating muscle activity," *Elife*, vol. 11:e67620. 2022. DOI: 10.7554/eLife.67620.

[101] M. Khona and I. R. Fiete, "Attractor and integrator networks in the brain," *arXiv*, 2021. DOI: 10.48550/arXiv.2112.03978.

[102] A. Nair *et al.*, "An approximate line attractor in the hypothalamus encodes an aggressive state," *Cell*, vol. 186, no. 1, 178–193.e15, 2023. DOI: https://doi.org/10.1016/j.cell.2022.11.027.

[103] K. S. Kakaria and B. L. de Bivort, "Ring Attractor Dynamics Emerge from a Spiking Model of the Entire Protocerebral Bridge," *Front. Behav. Neurosci.*, vol. 11, p. 236 548, 2017. DOI: 10.3389/fnbeh.2017.00008.

[104] E. Ghazizadeh and S. Ching, "Slow manifolds within network dynamics encode working memory efficiently and robustly," *PLoS computational biology*, vol. 17, no. 9, e1009366, 2021.

[105] R. Viviani, G. Grön, and M. Spitzer, "Functional principal component analysis of fMRI data," *Hum. Brain Mapp.*, vol. 24, no. 2, p. 109, 2005. DOI: 10.1002/hbm.20074.

[106] D. Kobak *et al.*, "Demixed principal component analysis of neural population data," *eLife*, vol. 5,

M. C. van Rossum, Ed., e10989, 2016. DOI: 10.7554/eLife.10989.

[107] Y. Nakatsukasa and N. J. Higham, "Stable and efficient spectral divide and conquer algorithms for the symmetric eigenvalue decomposition and the svd," *SIAM Journal on Scientific Computing*, vol. 35, no. 3, A1325–A1349, 2013. DOI: 10.1137/120876605.

[108] A. J. King, "The superior colliculus," *Curr. Biol.*, vol. 14, no. 9, R335–R338, 2004. DOI: 10.1016/j.cub.2004.04.018.

[109] R. Soetedjo, C. R. S. Kaneko, and A. F. Fuchs, "Evidence that the superior colliculus participates in the feedback control of saccadic eye movements," *J. Neurophysiol.*, vol. 87, no. 2, pp. 679–695, 2002. DOI: 10.1152/jn.00886.2000.

[110] W. Y. Choi and D. Guitton, "Responses of Collicular Fixation Neurons to Gaze Shift Perturbations in Head-Unrestrained Monkey Reveal Gaze Feedback Control," *Neuron*, vol. 50, no. 3, pp. 491–505, 2006. DOI: 10.1016/j.neuron.2006.03.032.

[111] R. de Kleijn, D. Sen, and G. Kachergis, "A Critical Period for Robust Curriculum-Based Deep Reinforcement Learning of Sequential Action in a Robot Arm," *Topics in Cognitive Science*, vol. 14, no. 2, p. 311, 2022. DOI: 10.1111/tops.12595.

[112] A. Dey, "Perceptual characteristics of visualizations for occluded objects in handheld augmented reality," Advisors: C. Sandor, B. Thomas, Ph.D. dissertation, University of South Australia, 2013.

[113] J. D. Schall, W. Zinke, J. D. Cosman, M. S. Schall, M. Paré, and P. Pouget, "On the Evolution of the Frontal Eye Field: Comparisons of Monkeys, Apes, and Humans," in *Evolutionary Neuroscience (Second Edition)*, Academic Press, 2020, pp. 861–890, ISBN: 978-0-12-820584-6.

[114] J. M. Samonds, V. Choi, and N. J. Priebe, "Mice Discriminate Stereoscopic Surfaces Without Fixating in Depth," *J. Neurosci.*, vol. 39, no. 41, pp. 8024–8037, 2019. DOI: 10.1523/JNEUROSCI.0895-19.2019.

[115] G. Zhou, J. Wu, C. Zhang, and Z. Zhou, "Minimal Gated Unit for Recurrent Neural Networks," *arXiv*, 2016. DOI: 10.48550/arXiv.1603.09420.

[116] J. Brownlee, "Train Neural Networks With Noise to Reduce Overfitting - MachineLearningMastery.com," *MachineLearningMastery*, 2019. [Online]. Available: https://machinelearningmastery.com/train-neural-networks-with-noise-to-reduce-overfitting.

[117] M. Zhou, T. Liu, Y. Li, D. Lin, E. Zhou, and T. Zhao, "Towards Understanding the Importance of Noise in Training Neural Networks," *arXiv*, 2019. DOI: 10.48550/arXiv.1909.03172.

[118] L. Johnston, *Tracking and pre-training task examples - Google Drive*, 2023. [Online]. Available: https://drive.google.com/drive/folders/1-bTlo7YY4J5dQCq_meE9VrOlt_0_iFfR.

[119] A. Nayebi, R. Rajalingham, M. Jazayeri, and G. R. Yang, "Neural Foundations of Mental Simulation: Future Prediction of Latent Representations on Dynamic Scenes," *arXiv*, 2023. DOI: 10.48550/arXiv.2305.11772.

[120] R. Schaeffer, M. Khona, T. Ma, C. Eyzaguirre, S. Koyejo, and I. R. Fiete, "Self-supervised learning of representations for space generates multi-modular grid cells," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2023. DOI: 10.48550/arXiv.2311.02316.

[121] *KL Divergence: Forward or Reverse - DKL(p||q) or DKL(q||p) - Which Selected?* [Online; accessed 5. Dec. 2023], 2019. [Online]. Available: https://www.tutorialexample.com/kl-divergence-forward-or-reverse-dklpq-or-dklqp-which-selected.

[122] J. Wen, S. Kumar, R. Gummadi, and D. Schuurmans, "Characterizing the Gap Between Actor-Critic and Policy Gradient," *arXiv*, 2021. DOI: 10.48550/arXiv.2106.06932.

[123] C. Stringer, M. Pachitariu, N. Steinmetz, M. Carandini, and K. D. Harris, "High-dimensional geometry of population responses in visual cortex," *Nature*, vol. 571, pp. 361–365, 2019. DOI: 10.1038/s41586-019-1346-5.

[124] J. D. Rivero-Ortega *et al.*, "Ring attractor bioinspired neural network for social robot navigation," *Front. Neurorob.*, vol. 17, p. 1 211 570, 2023. DOI: 10.3389/fnbot.2023.1211570.

[125] W. Skaggs, J. Knierim, H. Kudrimoti, and B. McNaughton, "A model of the neural basis of the rat's sense of direction," in *Advances in Neural Information Processing Systems*, G. Tesauro, D. Touretzky, and T. Leen, Eds., vol. 7, MIT Press, 1994.

[126] M. H. Johnson, "Functional brain development in humans," *Nat. Rev. Neurosci.*, vol. 2, pp. 475–483, 2001. DOI: 10.1038/35081509.

[127] M. H. Johnson, "Cortical Maturation and the Development of Visual Attention in Early Infancy,"

J. Cognit. Neurosci., vol. 2, no. 2, pp. 81–95, 1990. DOI: 10.1162/jocn.1990.2.2.81.

[128] J. Atkinson, "Human visual development over the first 6 months of life. A review and a hypothesis," *Hum. Neurobiol.*, vol. 3, no. 2, pp. 61–74, 1984.

[129] G. Hennequin, T. P. Vogels, and W. Gerstner, "Non-normal amplification in random balanced neuronal networks," *Phys. Rev. E*, vol. 86, p. 011 909, 1 2012. DOI: 10.1103/PhysRevE.86.011909.

[130] L. D. Sun and M. E. Goldberg, "Corollary Discharge and Oculomotor Proprioception: Cortical Mechanisms for Spatially Accurate Vision," *Annu. Rev. Vision Sci.*, vol. 2, p. 61, 2016. DOI: 10.1146/annurev-vision-082114-035407.

[131] J. B. Fileta *et al.*, "Efficient Estimation of Retinal Ganglion Cell Number: a Stereological Approach," *J. Neurosci. Methods*, vol. 170, no. 1, p. 1, 2008. DOI: 10.1016/j.jneumeth.2007.12.008.

[132] U. Grünert and P. R. Martin, "Cell types and cell circuits in human and non-human primate retina," *Prog. Retin. Eye Res.*, vol. 78, p. 100 844, 2020. DOI: 10.1016/j.preteyeres.2020.100844.

[133] U. S. Kim, O. A. Mahroo, J. D. Mollon, and P. Yu-Wai-Man, "Retinal Ganglion Cells—Diversity of Cell Types and Clinical Relevance," *Front. Neurol.*, vol. 12, p. 661 938, 2021. DOI: 10.3389/fneur.2021.661938.

[134] Y. Xu, Q. Kong, Q. Huang, W. Wang, and M. D. Plumbley, "Convolutional Gated Recurrent Neural Network Incorporating Spatial Features for Audio Tagging," *arXiv*, 2017. DOI: 10.48550/arXiv.1702.07787.

[135] T. Pasternak, J. W. Bisley, and D. Calkins, "Visual Processing in the Primate Brain," in *Handbook of Psychology*, John Wiley & Sons, Inc, 2003, pp. 139–185, ISBN: 978-0-47126438-5.

[136] J. D. Victor, "Temporal aspects of neural coding in the retina and lateral," *Network*, vol. 10, no. 4, R1, 1999. DOI: 10.1088/0954-898X/10/4/201.

[137] T. Serre, "Hierarchical Models of the Visual System," in *Encyclopedia of Computational Neuroscience*, Springer, 2014, pp. 1–12, ISBN: 978-1-4614-7320-6.

[138] H. Kafaligonul, "Vision: A Systems Neuroscience Perspective," *Journal of Neurobehavioral Sciences*, vol. 2, no. 1, p. 1, 2014. DOI: 10.5455/JNBS.1395935766.

[139] S. Hochstein and M. Ahissar, "View from the Top: Hierarchies and Reverse Hierarchies in the Visual System," *Neuron*, vol. 36, no. 5, pp. 791–804, 2002. DOI: 10.1016/S0896-6273(02)01091-7.

[140] D. C. Van Essen, C. H. Anderson, and D. J. Felleman, "Information Processing in the Primate Visual System: An Integrated Systems Perspective," *Science*, vol. 255, no. 5043, pp. 419–423, 1992. DOI: 10.1126/science.1734518.

[141] J. H. Siegle *et al.*, "Survey of spiking in the mouse visual system reveals functional hierarchy," *Nature*, vol. 592, pp. 86–92, 2021. DOI: 10.1038/s41586-020-03171-x.

[142] G. De Franceschi and S. G. Solomon, "Visual response properties of neurons in the superficial layers of the superior colliculus of awake mouse," *J. Physiol.*, vol. 596, no. 24, pp. 6307–6332, 2018. DOI: 10.1113/JP276964.

[143] S. Ito, D. A. Feldheim, and A. M. Litke, "Segregation of Visual Response Properties in the Mouse Superior Colliculus and Their Modulation during Locomotion," *J. Neurosci.*, vol. 37, no. 35, pp. 8428–8443, 2017. DOI: 10.1523/JNEUROSCI.3689-16.2017.

[144] C. Quaia, H. Aizawa, L. M. Optican, and R. H. Wurtz, "Reversible Inactivation of Monkey Superior Colliculus. II. Maps of Saccadic Deficits," *J. Neurophysiol.*, 1998. DOI: 10.1152/jn.1998.79.4.2097.

[145] G. L. Craig, L. B. Stelmach, and W. J. Tam, "Control of reflexive and voluntary saccades in the gap effect," *Percept. Psychophys.*, vol. 61, no. 5, pp. 935–942, 1999. DOI: 10.3758/BF03206907.

[146] E. M. Reingold and D. M. Stampe, "Saccadic Inhibition in Voluntary and Reflexive Saccades," *J. Cognit. Neurosci.*, vol. 14, no. 3, pp. 371–388, 2002. DOI: 10.1162/089892902317361903.

[147] D. J. Mort *et al.*, "Differential cortical activation during voluntary and reflexive saccades in man," *Neuroimage*, vol. 18, no. 2, pp. 231–246, 2003. DOI: 10.1016/S1053-8119(02)00028-9.

[148] S. Everling, M. C. Dorris, and D. P. Munoz, "Reflex Suppression in the Anti-Saccade Task Is Dependent on Prestimulus Neural Processes," *J. Neurophysiol.*, 1998. DOI: 10.1152/jn.1998.80.3.1584.

[149] S. Everling, M. C. Dorris, R. M. Klein, and D. P. Munoz, "Role of Primate Superior Colliculus in Preparation and Execution of Anti-Saccades and Pro-Saccades," *J. Neurosci.*, vol. 19, no. 7,

pp. 2740–2754, 1999. DOI: 10.1523/JNEUROSCI.19-07-02740.1999.

[150] D. P. Munoz and S. Everling, "Look away: the anti-saccade task and the voluntary control of eye movement," *Nat. Rev. Neurosci.*, vol. 5, pp. 218–228, 2004. DOI: 10.1038/nrn1345.

[151] A. K. Moschovakis, G. G. Gregoriou, and H. E. Savaki, "Functional imaging of the primate superior colliculus during saccades to visual targets," *Nat. Neurosci.*, vol. 4, pp. 1026–1031, 2001. DOI: 10.1038/nn727.

[152] Z. M. Hafed and C. Chen, "Sharper, Stronger, Faster Upper Visual Field Representation in Primate Superior Colliculus," *Curr. Biol.*, vol. 26, no. 13, pp. 1647–1658, 2016. DOI: 10.1016/j.cub.2016.04.059.

[153] Z. M. Hafed and L. Goffart, "Gaze direction as equilibrium: more evidence from spatial and temporal aspects of small-saccade triggering in the rhesus macaque monkey," *J. Neurophysiol.*, vol. 123, no. 1, pp. 308–322, 2020. DOI: 10.1152/jn.00588.2019.

[154] A. Nurmikko and L. Hochberg, *Pulse-step model for saccade generation*, Slide from "Neuroengineering: Control of Eye Movement", 2007. [Online]. Available: https://www.brown.edu/Departments/Engineering/Courses/122JDD/Lcturs/PulseStep.html.

[155] V. Vasudevan, R. Padhi, and A. Murthy, "Characterization of pulse-step input for saccadic eye movements," in *2016 Indian Control Conference (ICC)*, IEEE, 2016, pp. 4–6. DOI: 10.1109/INDIANCC.2016.7441099.

[156] D. A. Robinson, "Neurophysiology, pathology and models of rapid eye movements," in *Progress in Brain Research*, 1, vol. 267, Walthm, MA, USA: Elsevier, 2022, pp. 287–317. DOI: 10.1016/bs.pbr.2021.10.014.

[157] E. L. Keller and J. A. Edelman, "Use of interrupted saccade paradigm to study spatial and temporal dynamics of saccadic burst cells in superior colliculus in monkey," *J. Neurophysiol.*, vol. 72, no. 6, pp. 2754–2770, 1994. DOI: 10.1152/jn.1994.72.6.2754.

[158] E. L. Keller, N. J. Gandhi, and S. V. Sekaran, "Activity in deep intermediate layer collicular neurons during interrupted saccades," *Exp. Brain Res.*, vol. 130, no. 2, pp. 227–237, 2000. DOI: 10.1007/s002219900239.

[159] S. Matsuo, A. Bergeron, and D. Guitton, "Evidence for Gaze Feedback to the Cat Superior Colliculus: Discharges Reflect Gaze Trajectory Perturbations," *J. Neurosci.*, vol. 24, no. 11, p. 2760, 2004. DOI: 10.1523/JNEUROSCI.5120-03.2004.

[160] D. S. Zee, L. M. Optican, J. D. Cook, D. A. Robinson, and W. K. Engel, "Slow saccades in spinocerebellar degeneration," *Arch. Neurol.*, vol. 33, no. 4, pp. 243–251, 1976. DOI: 10.1001/archneur.1976.00500040027004.

[161] N. J. Gandhi and H. A. Katnani, "Motor Functions of the Superior Colliculus," *Annu. Rev. Neurosci.*, vol. 34, p. 205, 2011. DOI: 10.1146/annurev-neuro-061010-113728.

[162] H. A. Katnani, A. J. Van Opstal, and N. J. Gandhi, "Evaluating models of decoding superior colliculus activity for saccade generation," in *Soc. Neurosci. Abstr.*, 2009. DOI: 10.1152/jn.00265.2011.

[163] A. J. Van Opstal, J. A. Van Gisbergen, and A. C. Smit, "Comparison of saccades evoked by visual stimulation and collicular electrical stimulation in the alert monkey," *Exp. Brain Res.*, vol. 79, no. 2, pp. 299–312, 1990. DOI: 10.1007/BF00608239.

[164] M. Carandini and D. J. Heeger, "Summation and division by neurons in primate visual cortex," *Science*, vol. 264, no. 5163, pp. 1333–1336, 1994. DOI: 10.1126/science.8191289.

[165] J. M. Groh, "Converting neural signals from place codes to rate codes," *Biol. Cybern.*, vol. 85, no. 3, pp. 159–165, 2001. DOI: 10.1007/s004220100249.

[166] M. Behan and N. M. Kime, "Intrinsic circuitry in the deep layers of the cat superior colliculus," *Visual Neurosci.*, vol. 13, no. 6, pp. 1031–1042, 1996. DOI: 10.1017/s0952523800007689.

[167] P. H. Lee, M. C. Helms, G. J. Augustine, and W. C. Hall, "Role of intrinsic synaptic circuitry in collicular sensorimotor integration," *Proc. Natl. Acad. Sci. U.S.A.*, vol. 94, no. 24, pp. 13 299–13 304, 1997. DOI: 10.1073/pnas.94.24.13299.

[168] M. Takahashi, Y. Sugiuchi, and Y. Shinoda, "Topographic organization of excitatory and inhibitory commissural connections in the superior colliculi and their functional roles in saccade generation," *J. Neurophysiol.*, vol. 104, no. 6, pp. 3146–3167, 2010. DOI: 10.1152/jn.00554.2010.

[169] R. W. Anderson, E. L. Keller, N. J. Gandhi, and S. Das, "Two-Dimensional Saccade-Related Population Activity in Superior Colliculus in Monkey," *J. Neurophysiol.*, 1998. DOI: 10.1152/jn.1998.80.2.798.

[170] N. L. Port, M. A. Sommer, and R. H. Wurtz, "Multielectrode evidence for spreading activity across the superior colliculus movement map," *J. Neurophysiol.*, vol. 84, no. 1, pp. 344–357, 2000. DOI: 10.1152/jn.2000.84.1.344.

[171] R. Soetedjo, C. R. S. Kaneko, and A. F. Fuchs, "Evidence against a moving hill in the superior colliculus during saccadic eye movements in the monkey," *J. Neurophysiol.*, vol. 87, no. 6, pp. 2778–2789, 2002. DOI: 10.1152/jn.2002.87.6.2778.

[172] H. Aizawa and R. H. Wurtz, "Reversible Inactivation of Monkey Superior Colliculus. I. Curvature of Saccadic Trajectory," *J. Neurophysiol.*, 1998. DOI: 10.1152/jn.1998.79.4.2082.

[173] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge: MIT Press, 2016, ISBN: 9780262035613.

[174] J. Guo, *AI Notes: Initializing neural networks - deeplearning.ai*, 2024. [Online]. Available: https://www.deeplearning.ai/ai-notes/initialization/index.html.

[175] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proc. 13th AISTATS*, JMLR Workshop and Conference Proceedings, 2010, pp. 249–256.

[176] D. P. Kingma and M. Welling, "Auto-Encoding Variational Bayes," *arXiv*, 2013. DOI: 10.48550/arXiv.1312.6114.

[177] J. Bradbury *et al.*, *JAX: Composable transformations of Python+NumPy programs*, Version 0.3.13, 2018. [Online]. Available: http://github.com/google/jax.

[178] DeepMind *et al.*, *The DeepMind JAX Ecosystem*, 2020. [Online]. Available: http://github.com/google-deepmind.

[179] G. Van Rossum and F. L. Drake Jr, *Python tutorial*. Centrum voor Wiskunde en Informatica Amsterdam, The Netherlands, 1995.

[180] D. Kobak *et al.*, "Demixed principal component analysis of neural population data," *eLife*, vol. 5, 2016. DOI: 10.7554/elife.10989.

[181] A. Paszke *et al.*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," *arXiv*, 2019. DOI: 10.48550/arXiv.1912.01703.

Appendix A

Mathematical Models of SC Vector Decoding

This appendix briefly details mathematical forms for the models used to decode SC activity to a saccadic movement vector, as discussed in Section 1.2. Note a detailed treatment of the downstream neural mechanisms responsible for enacting the decoded vector - involving interactions between neural control signals, ocular dynamics, and the oculomotor neural integrator (OMN) - are ommitted for brevity. Such interactions are often described in the literature via a model known as the *pulse-step model* [48, 154–156].

A.1 Static Models

In the following models, the SC's spatial code is considered to be *statically* (instantaneously) decoded to a saccade vector, which is then enacted by a separate downstream motor loop. Such motor feedback loops are a fundamental aspect of neural motor control, where movements are often considered to be governed by a closed loop involving some form of motor feedback error signal. In the case of saccades, perturbation studies of gaze trajectories provide evidence for such a controller to ensure the accuracy of saccadic movements [157–159], where the error signal involved is considered to be formed via efference copy feedback of eye displacement and velocity [50, 51][1].

A.1.1 Static Vector Averaging

In its simplest form, the averaging model [42, 43] proposes that deep SC activity is decoded to a resultant saccade vector as

$$\vec{S} = \frac{\sum_{n=1}^{N} r_n \vec{m}_n}{\sum_{n=1}^{N} r_n} \tag{A.1}$$

where $\vec{S}$ denotes the resultant saccade, r_n is the mean firing rate of cell n in the motor map, and $\vec{m}_n$ is the vector encoded by that neuron[2]. As one of the first models used to describe this process, such a model successfully accounted for a number of experimental results - for example producing saccades with amplitude and direction correctly predicted by a weighted average from multiple sites of stimulation [8, 162]. To better account for results demonstrating a sigmoidal dependence of saccade amplitude with current intensity [163], this model is sometimes appended with an additional parameter [49]

$$\vec{S} = \frac{\sum_{n=1}^{N} r_n \vec{m}_n}{K + \sum_{n=1}^{N} r_n} \tag{A.2}$$

[1] Visual pathways are considered prohibitively slow to feature in such a loop [160].

[2] Note the use of explicit vector notation as opposed to the bold notation previously used in this work to denote vectors, ensuring consistency with how these models are defined in the literature [161].

where K is a constant considered to reflect an 'inhibitory threshold', reducing saccade amplitude for low population activity to better fit experimental data. The main flaw with this model lies in the biological feasibility of an averaging mechanism. Though some network architectures which can accomplish such a normalising operation have been proposed [164, 165], there is little evidence to suggest such an averaging computation could be physiologically implemented within the oculomotor system.

A.1.2 Static Vector Summation

The vector summation model considers a resultant vector described as

$$\vec{S} = \alpha \sum_{n=1}^{N} r_n \vec{m}_n \tag{A.3}$$

where $\vec{m}_n$ is the vector contribution of cell n with firing rate weighting r_n, and α is a fixed scaling constant. Under this model, each active neuron now contributes a vector weighted by the mean firing rate of the cell. This model is simpler and more intuitive than the averaging model, and is hence often considered a more plausible mechanism for decoding motor commands [46]. It also has a number of shortcomings however. For example, it struggles to represent behaviour resulting from multiple sites of stimulation - requiring the additional incorporation of intracollicular connectivity features such as local excitation and distal inhibition to do so [166–168].

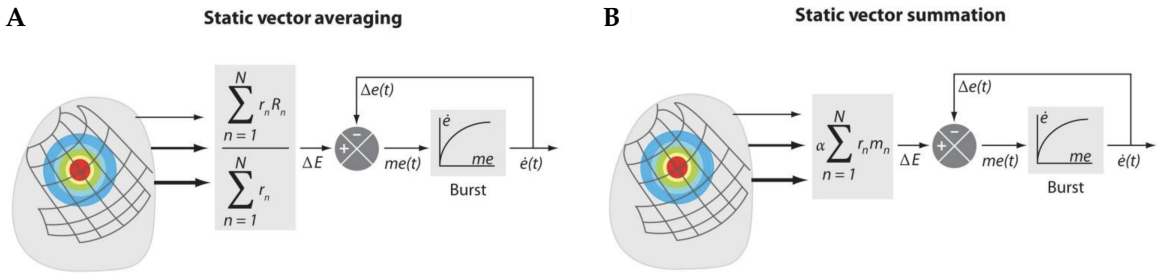


Figure A.1: **Static vector averaging and summation models of SC decoding. (A)** Static vector averaging. Intended saccade is statically decoded to a desired eye displacement as an average vector based on the formula in eq. (A.1), before implementation via a downstream motor feedback loop. **(B)** Vector summation model. Similar to the averaging model, intended saccade is now obtained via a sum of vector contributions from each cell. Model parameters: ΔE, desired eye displacement (equivalent to $\vec{S}$ in eqs. (A.1-A.3)); $\Delta e(t)$, current eye displacement; $me(t)$, dynamic motor error (eye displacement error); $\dot{e}(t)$, current eye velocity; "Burst", brainstem burst generator (acting as a form of differential controller in the oculomotor loop). Note these diagrams omit the 'neural eye displacement integrator' in the feedback loop, responsible for integrating current eye velocity to obtain current eye displacement, as well as the downstream oculomotor system (plant) which enacts the saccade via a 'pulse-step' mechanism. Reused from Gandhi & Katnani (2011) [161], originally adapted from Goossens & Van Opstal (2006) [48].

A.2 Dynamic Models

Increasing evidence, however, suggests the superior colliculus recruits both spatial and temporal information to encode the ensuing saccade, as discussed in the main text. A range of models have since been developed which incorporate temporal aspects to better model experimental data. This temporal signal then dynamically modulates the activity of the downstream motor loop which performs the saccade.

A.2.1 Dynamic Vector Averaging

In the case of vector averaging, the previous static model has been expanded to now consider instantaneous firing rate of SC activity to modulate gain of the brainstem burst generator in the downstream motor feedback loop (Figure A.2) [44, 45]. Such a mechanism reflects the *dual coding hypothesis* proposed based on experimental data [44], where saccade direction and dynamics are encoded by activity location and rate respectively. While this implementation successfully demonstrates both the metrics and kinematics of resulting movements can be explained by such an averaging scheme, some analyses of this model suggest the true kinematics of saccades are still not accurately captured [48, 49].

A.2.2 Dynamic Vector Summation

For vector summation, a 'dynamic ensemble coding' model has been developed where the integral of activity over time describes intended saccade trajectory, dynamically governing movement kinematics. This model hence considers a time-varying saccade modelled as

$$\vec{S}(t) = \sum_{k=1}^{N_A} \sum_{n=1}^{N_S} \vec{s}_k \delta(t - \tau_{k,n}) \tag{A.4}$$

where $\vec{s}_k$ is a scaled eye-displacement vector generated by a single spike of the k^{th} neuron, and $\tau_{k,n}$ denotes the the time of the n^{th} spike of neuron k. N_A describes the total number of neurons in the population, and N_S the number of spikes produced by neuron k (counted from 20 ms before onset to 20 ms before saccade offset). Each spike from an SC cell therefore now adds a site-specific 'mini' vector $\vec{s}_k$ contribution to the movement command, as specified by the delta function term. While simulations from this model demonstrate some saccade properties other models are unable to incorporate, a significant shortcoming is that it always yields a single vector summation locus when tested with two sites of activity, whereas similar conditions ethologically have been shown to generate activity at multiple locations [38, 39].

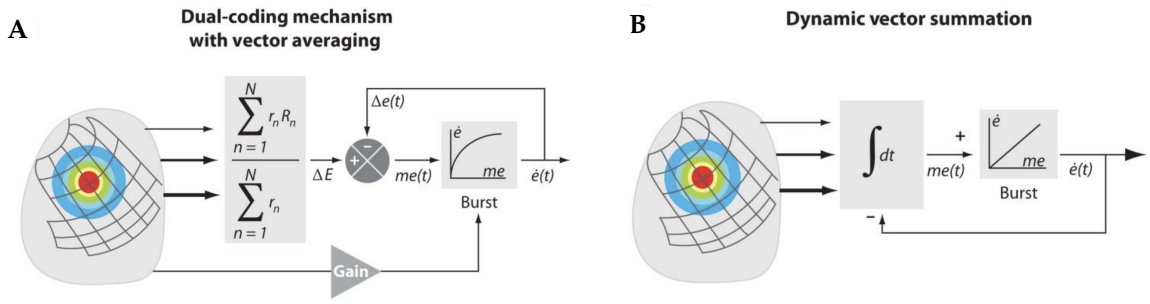


Figure A.2: **Dynamic vector averaging and summation models of SC decoding. (A)** Dynamic vector averaging. Based on a 'dual coding' mechanism, SC activity now dynamically governs the saccade via a temporal signal which modulates gain of the brainstem burst generator in the downstream motor loop. **(B)** Dynamic vector summation model. Saccade displacement is now described by a dynamic summation across time from the active population, where the accumulating activity specifies intended trajectory. Model parameters are equivalent to those in Figure A.1. Reused from Gandhi & Katnani (2011) [161], originally adapted from Goossens & Van Opstal (2006) [48].

A.2.3 Dynamic Motor Error Models

Finally, there exist alternative models of the SC's spatiotemporal code in which the SC is situated directly *within* the motor feedback loop governing saccades [57, 58, 109]. In such models SC motor

activity directly encodes instantaneous motor error (i.e. difference between the current gaze position and saccade goal), effectively acting as a comparator within the wider motor loop. An important experimental observation which provides evidence for this is the observed decay of motor layer neuron discharge within the saccadic period [57, 169], appearing to naturally correlate with motor error.

One such theory - the 'moving hill' model - suggests motor error to be dynamically represented by a locus of activity which moves across the motor map during the peri-saccadic period. The initial position of this locus therefore represents intended saccade vector, and the final position corresponds to zero error at the end of the saccade [58]. Another hypothesis - the 'decaying hill' model - similarly suggests this locus of activity encodes motor error, albeit via its activity rate, which is hypothesised to decay statically during the saccade [57]. While both models succeed in replicating many characteristic properties of ethological saccade dynamics [170], multiple other investigations cast doubt on these theories, citing experimental data that appears to conflict with the precise patterns of activity predicted by these models [144, 171, 172]. As such, the consensus appears to be that activity on the motor map does not dynamically encode gaze position error.

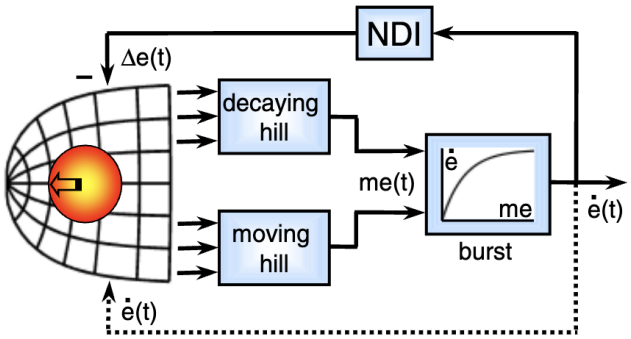


Figure A.3: **Dynamic motor error models of SC decoding.** Top loop: The 'decaying hill' model considers instantaneous firing rate to directly encode dynamic motor error, via a mechanism where eye feedback (top, solid) has an inhibitory effect on the population. Bottom loop: The 'moving hill' model instead considers that eye velocity feedback (bottom, dotted) causes a caudal-to-rostral movement (arrow) of the population locus, location of which encodes dynamic motor error. In both models, the SC now directly represents motor error by effectively acting as a comparator in the loop - as opposed to specifying a saccade 'goal' as a signal to a separate downstream loop. "NDI" in the top feedback loop denotes the neural eye displacement integrator, which integrates current eye velocity to obtain current eye displacement. Model parameters are once again equivalent to those in Figure A.1. Reused from Goossens & Van Opstal (2006) [48].

Appendix B

Supplementary Mathematics

B.1 Glorot Initialisation

Care must be taken when initialising network weights θ for the optimisation algorithms described in the main text, as poor initialisation can lead to slow, unstable learning which may not reach an optimal solution [173, 174]. Many initialisation rules therefore exist to prevent this problem and improve training performance. This work uses a method known as Glorot initialisation [175], which we briefly derive below. We start by making a few initial assumptions. First, that we consider a standard fully connected network of form

$$\mathbf{z}^{(l)} = \mathbf{W}^{(l)}\mathbf{a}^{(l-1)} + \mathbf{b}^{(l)} \tag{B.1}$$

$$\mathbf{a}^{(l)} = \sigma(\mathbf{z}^{(l)}) \tag{B.2}$$

and secondly, that our activation function is symmetric and has approximately unit derivative at 0. Finally, we assume weights to be randomly initialised and are independent of the input activations on which they act.

We then consider the reasonable heuristic goal for our initialisation that we want to keep variance of the network's output at each layer in line with that of the previous layer - both in the forward pass, to keep computations within a stable range, and in the reverse pass, preventing vanishing or exploding gradients. These two constraints can be mathematically formalised as[1]

$$\text{Var}[z^{(l)}] = \text{Var}[z^{(l-1)}] \qquad \text{(Forward pass)} \tag{B.3}$$

$$\text{Var}\left[\frac{\partial L}{\partial z^{(l)}}\right] = \text{Var}\left[\frac{\partial L}{\partial z^{(l+1)}}\right] \qquad \text{(Backward pass)} \tag{B.4}$$

For the forward pass condition, we first assume the biases contribute minimally to total variance given the small magnitude of their initialisations, and the activation similarly contributes minimally given network activity is considered to fall largely in this function's linear region. Ignoring both, we therefore

[1]Note we simplify the mathematics involved by considering scalar variables (instead of their composite vectors or matrices), where each scalar element is assumed to be sampled from some underlying distribution.

have pre-activation values in the forward pass with variance described as

$$\begin{aligned}
\mathrm{Var}\big[z^{(l)}\big] &\approx \mathrm{Var}\left[\sum_j w_{\cdot,j}^{(l)} a_j^{(l-1)}\right] && \text{(B.5)}\\
&\approx \sum_j \mathrm{Var}\left[w_{\cdot,j}^{(l)}\right] \mathrm{Var}\left[a_j^{(l-1)}\right] && \text{(B.6)}\\
&\approx n_{\text{in}}^{(l)} \mathrm{Var}\big[w^{(l)}\big] \mathrm{Var}\big[a^{(l-1)}\big] && \text{(B.7)}\\
&\approx n_{\text{in}}^{(l)} \mathrm{Var}\big[w^{(l)}\big] \mathrm{Var}\big[z^{(l-1)}\big] && \text{(B.8)}
\end{aligned}$$

where $n_{\text{in}}^{(l)}$ describes the number of inputs at that layer (i.e. length of $\mathbf{a}^{(l-1)}$). Similarly for the reverse pass, we recall the initial assumptions made enabling us to ignore the gradient of the activation and hence obtain

$$\begin{aligned}
\mathrm{Var}\big[\delta^{(l)}\big] &\approx \mathrm{Var}\left[\sum_j w_{j,\cdot}^{(l+1)} \delta_j^{(l+1)}\right] && \text{(B.9)}\\
&\approx \sum_j \mathrm{Var}\left[w_{j,\cdot}^{(l+1)}\right] \mathrm{Var}\left[\delta_j^{(l+1)}\right] && \text{(B.10)}\\
&\approx n_{\text{out}}^{(l+1)} \mathrm{Var}\big[w^{(l+1)}\big] \mathrm{Var}\big[\delta^{(l+1)}\big] && \text{(B.11)}\\
&\approx n_{\text{out}}^{(l+1)} \mathrm{Var}\big[w^{(l+1)}\big] \mathrm{Var}\big[\delta^{(l+1)}\big] && \text{(B.12)}
\end{aligned}$$

where $\delta^{(l)}$ denotes the partial derivative of the loss with respect to the output at that layer, and $n_{\text{out}}^{(l+1)}$ describes the number of outputs (i.e. length of $\mathbf{z}^{(l+1)}$). Subbing in our conditions from eqs. (B.3, B.4), we have

$$\begin{aligned}
n_{\text{in}}^{(l)} \mathrm{Var}\big[w^{(l)}\big] &\approx 1 && \text{(B.13)}\\
n_{\text{out}}^{(l)} \mathrm{Var}\big[w^{(l)}\big] &\approx 1 && \text{(B.14)}
\end{aligned}$$

As a reasonable compromise we can then take the average between both conditions, obtaining

$$\mathrm{Var}\big[w^{(l)}\big] \approx \frac{2}{n_{\text{in}}^{(l)} + n_{\text{out}}^{(l)}} \tag{B.15}$$

We can then finally draw initial weights from some distribution with a variance that satisfies this. In this work we use a normal distribution, with initial weights sampled as

$$w_{i,j}^{(l)} \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}}^{(l)} + n_{\text{out}}^{(l)}}\right) \tag{B.16}$$

Such initialisation aims to balance the variance of activations and their gradients across layers, in doing so mitigating the common problems of vanishing or exploding gradients in RNNs and helping to stabilise training. While the above result makes assumptions that are not directly comparable to the GRU networks used in our work (which feature complex gating and sigmoid activations), this rule nevertheless produces strong results empirically.

B.2 Policy Gradients for a Partially Observed Markov Decision Process

In the main text we use the *log-trick* to consider a policy gradient expression which in expectation has the form

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\nabla_\theta \log p_\theta(\tau) R(\tau) \right] \tag{B.17}$$

However to obtain the sample-based gradient estimator used in our simulations (Section 2.3.2, eq. (2.30)), we must carefully examine the $\nabla \log p_\theta(\tau)$ term to ensure such gradients are obtainable in a sample-based manner. We first consider the fully observed case, writing the distribution of trajectories $p_\theta(\tau)$ explicitly as a function of both our policy π_θ and state transition probability $p(\mathbf{s}_{t+1}, |\mathbf{s}_t, \mathbf{a}_t)$ as

$$p_\theta(\tau) = p_\theta(\mathbf{s}_1, \mathbf{a}_1, ..., \mathbf{s}_T, \mathbf{a}_T) = p(\mathbf{s}_1) \prod_{t=1}^{T} \pi_\theta(\mathbf{a}_t|\mathbf{s}_t) p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t) \tag{B.18}$$

where hence

$$\log p_\theta(\tau) = \log p(\mathbf{s}_1) + \sum_{t=1}^{T} \log \pi_\theta(\mathbf{a}_t|\mathbf{s}_t) + \log p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t) \tag{B.19}$$

noting reward is not considered a parameter of the trajectory distribution, given it is a deterministic function of the other quantities. Importantly, we have also assumed the Markov property for our policy - that it is independent of past states given the current state. Taking the derivative of each term with respect to θ we observe the initial state probability and transition probabilities are not functions of θ and as such can be ignored, leaving us with

$$\nabla_\theta \log p_\theta(\tau) = \nabla_\theta \left[\sum_{t=1}^{T} \log \pi_\theta(\mathbf{a}_t|\mathbf{s}_t) \right] \tag{B.20}$$

Substituting this back into our gradient term, we obtain

$$\nabla_\theta J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\mathbf{a}_t|\mathbf{s}_t) \right) \left(\sum_{t=1}^{T} r(\mathbf{s}_t, \mathbf{a}_t) \right) \right] \tag{B.21}$$

where it is clear all terms in this expression can be obtained for some sampled trajectory. As such we can estimate this expression via Monte Carlo methods, where

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t}) \right) \left(\sum_{t=1}^{T} r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \right) \tag{B.22}$$

which represents the final form of the un-modified policy gradient estimator for the fully observed case.

Turning now to the partially observed case as used in our work, we first acknowledge that in the partially observed case our agent typically makes decisions which are not solely functions of the current observation but also take into account previous observations, actions, and rewards, as contained in the task history $H_t = (\mathbf{o}_{1:t}, \mathbf{a}_{1:t-1}, r_{1:t-1})$. We therefore no no longer assume the Markov property. However by once again considering the concept of a *belief state* - capturing all information from this history as necessary for decision making at each timestep - a similar derivation is possible. We start by considering a verbose expression for our partially observed trajectories, which now incorporates observations $\mathbf{o}_t$ to

take the form[2]

$$p_\theta(\tau) = p_\theta(\mathbf{s}_1, \mathbf{o}_1, \mathbf{a}_1, ..., \mathbf{s}_T, \mathbf{o}_T, \mathbf{a}_T) = p(\mathbf{s}_1) \prod_{t=1}^{T} p(\mathbf{o}_t|\mathbf{s}_t)\pi_\theta(\mathbf{a}_t|\mathbf{o}_{1:t}, \mathbf{a}_{1:t-1}, r_{1:t-1})p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t) \tag{B.23}$$

where our agent's policy no longer obeys the Markov property, and is instead conditioned on the entire history of past observations, actions and rewards at each timestep. We also now consider the probability of a given state observation, alongside the policy and state transition probabilities. Given our agent never fully observes a given state, a more appropriate form of trajectory probability is that which marginalises out all states to consider only observations and actions - denoted the *observation-action trajectory*

$$p_\theta(\tau) = p_\theta(\mathbf{o}_1, \mathbf{a}_1, ..., \mathbf{o}_T, \mathbf{a}_T) = \int_{s_{1:T}} p(\mathbf{s}_1) \prod_{t=1}^{T} p(\mathbf{o}_t|\mathbf{s}_t)\pi_\theta(\mathbf{a}_t|\mathbf{o}_{1:t}, \mathbf{a}_{1:t-1}, r_{1:t-1})p(\mathbf{s}_{t+1}|\mathbf{s}_t, \mathbf{a}_t)\, ds_{1:T} \tag{B.24}$$

To enable a similar treatment of this expression as with the fully observed case, we introduce the belief state $b(\mathbf{s}_t)$ to encapsulate information contained in all past observations and actions via a theoretical distribution over all possible states given these quantities. This distribution takes the theoretical form

$$b(\mathbf{s}_t) = \eta\, p(\mathbf{o}_t|\mathbf{s}_t) \int_{s_{t-1}} p(\mathbf{s}_t|\mathbf{s}_{t-1}, \mathbf{a}_{t-1})b(\mathbf{s}_{t-1})\, ds_{t-1} \tag{B.25}$$

where η is a normalising factor

$$\eta^{-1} = \int_{s_t} p(\mathbf{o}_t|\mathbf{s}_t) \int_{s_{t-1}} p(\mathbf{s}_t|\mathbf{s}_{t-1}, \mathbf{a}_{t-1})b(\mathbf{s}_{t-1})\, ds_{t-1} ds_t \tag{B.26}$$

though in practise, these quantities are' not explicitly computed by our RL agent. Using this belief state, denoted with shorthand b_t, we can rewrite the observation-action trajectory in a simpler form as

$$p_\theta(\tau) = p_\theta(\mathbf{o}_1, \mathbf{a}_1, ..., \mathbf{o}_T, \mathbf{a}_T) = p(b_1) \prod_{t=1}^{T} \pi_\theta(\mathbf{a}_t|b_t)p(\mathbf{o}_{t+1}|b_t, \mathbf{a}_t) \tag{B.27}$$

Therefore despite our original partially observed process being non-Markovian, conditioned on such a belief state - updated at each timestep to incorporate past information - our process retains the Markov property. We can therefore perform similar analysis to the fully observed case, obtaining a gradient term of form

$$\nabla_\theta \log p_\theta(\tau) = \nabla_\theta \left[\sum_{t=1}^{T} \log \pi_\theta(\mathbf{a}_t|b(\mathbf{s}_t)) \right] \tag{B.28}$$

In practise in the main text, we represent the policy in the partially observed case as $\pi_\theta(\mathbf{a}_t|\mathbf{h}_t)$. This is because at code level the agent's policy is conditioned solely on the hidden state $\mathbf{h}_t$ of its recurrent neural network, and so we consider all information contained within the implicit belief state $b(\mathbf{s}_t)$ (or alternatively the history H_t) relevant to the agent's decision making process to be contained within this rolling hidden state. Substituting in the reward term, the resulting sample-based policy gradient

[2]Evaluating this expression, we note that the right terms only neatly combine to give the joint trajectory probability in the case agent policy is also conditioned on *state*, which we know this is not the case in our partially observed setting. We justify this equality by considering that the observation and history together form a 'partial state representation' that carries all state information necessary to form the agent's policy. This trajectory is however less directly meaningful than a trajectory for the fully observed case, as state information is no longer directly incorporated into the policy.

estimator in the partially observed case then takes the final form seen in the main text

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\left(\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{h}_{i,t})\right)\left(\sum_{t=1}^{T} r(\mathbf{s}_{i,t},\mathbf{a}_{i,t})\right) \tag{B.29}$$

B.3 Supplementary Details for Demixed Principle Component Analysis

B.3.1 Marginalisation

As described in Section 2.5.2 of the main text, a key element of demixed principle component analysis (dPCA) involves the separation of the data into a sum of marginalised datasets, as $\mathbf{X} = \sum_\phi \mathbf{X}_\phi$. For the example parameter set $S = \{t, s, d\}$, the simple case previously given for a single variable was

$$\mathbf{X}_t = \langle \mathbf{X} \rangle_{sd} \tag{B.30}$$

For up to two parameters, the total marginalised decomposition can be therefore be obtained by computing $\mathbf{X}_\phi$ as so for each individual parameter, then obtaining the final 'interaction' term as the remainder after each of these is subtracted from the original data. The general case for more than two parameters is however somewhat more complex, and is clarified below[3]. We start by generalising our decomposition for some set of parameters S as

$$x(S) = \sum_{\phi \subseteq S} z(\phi) \tag{B.31}$$

where $x(S)$ represents the decomposition we seek and $z(\phi)$ is some function which computes the marginalised dataset for each possible subset of S. We then introduce the following formula for computing each marginalised average

$$z(\phi) = \langle x(S) \rangle_{S \setminus \phi} - \sum_{\tau \subset \phi} z(\tau) \tag{B.32}$$

where the angle bracket term denotes the averaging of $x(S)$ over all parameters $y \in S$ that are not also elements of $y \notin \phi$. For $\phi = t$ this clearly reduces to eq. (B.30). For an example 'interaction component' between decision and stimulus $\phi = \{d, s\}$, we then have

$$z(\phi) = \langle \mathbf{X} \rangle_t - z(d) - z(s) \tag{B.33}$$

As before, the highest order interaction term is clearly formed as the remainder by subtracting all lower order terms. Note that in the original dPCA paper [106] the formula given for calculating the time-decision interaction term is

$$\mathbf{X}_{dt} = \langle \mathbf{X} - \mathbf{X}_t \rangle_s \tag{B.34}$$

which is effectively a reduction of the corresponding formula obtained from eq. (B.28), which emerges by pooling $\mathbf{X}_d, \mathbf{X}_{dt} \to \mathbf{X}_{dt}$ and making certain other assumptions[4].

[3] This is important as in this work (Section 4.4), dPCA is used to compute a marginalised dataset with respect to the variables time, decision (i.e. to move or plan), angle, and speed (the latter two referring to the task target) - in other words we have $S = \{t, d, a, s\}$. This allows us to consider the relationship between solely planning related activity and solely target speed-related activity, as in Figure 4.5.

[4] Specifically, that time and stimulus are independent (i.e. $\langle \mathbf{X}_t \rangle_s = \mathbf{X}_t$), and that our averaging operation exhibits linearity.

B.3.2 Regularisation

As briefly discussed in the main text, the full objective for demixed PCA includes a regularisation term λ, as

$$\mathcal{L}_{\text{dPCA}} = \sum_{\phi} \mathcal{L}_{\phi} = \sum_{\phi} \left(\|\mathbf{X}_{\phi} - \mathbf{F}_{\phi}\mathbf{D}_{\phi}\mathbf{X}\|^2 + \lambda \|\mathbf{D}_{\phi}\|^2 \right) \tag{B.35}$$

to prevent overfitting and thus improve validation set performance. This prevents our PCs from being overly sensitive to small variations in the data or capturing noise, common problems when fitting high-dimensional data.

We can explain this problem in more detail by considering how instability arises during the fitting process. In the (common) scenario in which the number of neurons and data points are of similar order, there is insufficient data to accurately estimate the relationships between neurons - our data matrix is *ill-conditioned*. That is, many eigenvectors have eigenvalues close to 0, explaining minimal variance in the data and largely capturing noise. The distribution of eigenvalues can hence be considered to peak at zero. This can create problems, as with no regularisation the space of eigenvectors with eigenvalues close to zero - the *kernel*[5] - will typically overlap with the decoder (i.e. the dPCs computed) for individual marginalisations. In other words without regularisation, the decoder directions computed for each marginalisation will typically lie within the kernel such that they overlap with the leading PC directions for one marginalisation (i.e. as we seek) but are orthogonal to all other sources of variance in the data. Many directions will hence have very small variance, and as such the encoder columns associated with reconstructing these low-variance directions will have large column norms to compensate. Therefore with only slight instabilities of the kernel subspace (i.e. for validation data with a different noise profile), this same encoder will strongly amplify noise on a validation set, leading to large cross validation errors.

We therefore add a regularisation term to prevent this problem, as in eq. (B.31). The regularisation term λ is obtained via cross-validation, where we hold out one trial for each neuron in each condition to form the dataset $\mathbf{X}_{\text{test}}$, and average the remaining data to form $\mathbf{X}_{\text{train}}$ (which has the same dimensions). We can then perform dPCA on $\mathbf{X}_{\text{train}}$ for varying values of λ to obtain $\mathbf{F}_{\phi}(\lambda)$ and $\mathbf{D}_{\phi}(\lambda)$, quantifying performance on the test data by computing the fraction of training set variance not explained by the held-out data via as

$$R(\lambda) = \frac{\sum_{\phi} \|\mathbf{X}_{\text{train},\phi} - \mathbf{F}_{\phi}(\lambda)\mathbf{D}_{\phi}(\lambda)\mathbf{X}_{\text{test}}\|^2}{\|\mathbf{X}_{\text{train}}\|^2} \tag{B.36}$$

where $R(\lambda)$ is the *reconstruction error*, which we attempt to minimise. Repeating this procedure across multiple train-test splits and averaging results, we obtain the optimal value of λ for use with our dPCA algorithm.

Finally, we can discuss the algorithmic procedure for implementing dPCA for $\lambda \neq 0$, showing this can be simplified to a reduced-rank regression problem as detailed in the main text for the $\lambda = 0$ case. First, we note our minimisation objective for each ϕ is now formalised as

$$\mathbf{A}_{\phi} = \underset{\mathbf{A}}{\text{argmin}} \|\mathbf{X}_{\phi} - \mathbf{A}\mathbf{X}\|^2 + \lambda \|\mathbf{A}\|^2, \quad \text{s.t} \quad \text{rank}(\mathbf{A}) \leq K \tag{B.37}$$

[5]Note that while the kernel typically refers to the set of vectors mapped to zero under that matrix transformation (explaining zero variance in the data), in this context we expand this definition to also include vectors which explain close to zero variance - for example those solely capturing noise in the data.

Given that we assume the encoder matrix $\mathbf{F}$ to be orthogonal[6], we also have

$$\|\mathbf{A}\|^2 = \|\mathbf{FD}\|^2 = \mathrm{Tr}(\mathbf{FDD}^\top\mathbf{F}^\top) \tag{B.38}$$

$$= \mathrm{Tr}(\mathbf{F}^\top\mathbf{FDD}^\top) = \mathrm{Tr}(\mathbf{DD}^\top) \tag{B.39}$$

$$= \|\mathbf{D}\|^2 \tag{B.40}$$

We can hence reform this optimisation problem into a more compact form. Given $\mathbf{A} \in \mathbb{R}^{N\times N}$, we concatenate $\mathbf{X}$ with the $N \times N$-dimensional identity matrix $\mathbf{I}_N$ to obtain $[\mathbf{X}|\mathbf{I}_N]$, and concatenate $\mathbf{X}_\phi$ with the $N \times N$-dimensional zero matrix $\mathbf{0}_N$ to obtain $[\mathbf{X}_\phi|\mathbf{0}_N]$, in both cases along the column indices. We can hence rewrite our regularised objective in eq. (B.33) as

$$\mathbf{A}_\phi = \underset{\mathbf{A}}{\mathrm{argmin}}\|[\mathbf{X}_\phi|\mathbf{0}_N] - \mathbf{A}[\mathbf{X}|\lambda\mathbf{I}_N]\|^2, \quad \text{s.t} \quad \mathrm{rank}(\mathbf{A}) \leq K \tag{B.41}$$

where the problem has now reduced to that of the unregularised case and can similarly be solved as a reduced rank regression problem, as detailed in Algorithm 3 in Section 2.5.2 of the main text.

B.4 Reparameterization trick

In Section 3.3 the first term of our RL objective has form

$$\Psi(\theta) = \mathbb{E}_{\tau\sim p_\theta(\tau)}\left[R\left(\tau\right)\right] \tag{B.42}$$

which we consider to be fully differentiable at code-level when estimated as a sample-based objective $\hat{\Psi}(\theta)$, despite the inclusion of additive 'motor noise' in our agent's actions (Section 3.2). To formally demonstrate this differentiability, we can use a principle known as the *reparameterization trick*. Such a trick allows us to show any independent, additive noise injected into our system can be considered instead as an *input* to the system, avoiding the need to differentiate directly through the noise itself.

As a preliminary, we motivate the decision for direct differentiation of our estimated objective when compared to other methods for gradient estimation. To recap, eq. (B.38) has analytic form

$$\Psi\left(\theta\right) = \int_\tau p_\theta(\tau)R(\tau)d\tau \tag{B.43}$$

for which we can obtain an unbiased estimate via Monte Carlo methods as

$$\hat{\Psi}\left(\theta\right) = \frac{1}{N}\sum_{i=1}^{N} R(\tau_i) \tag{B.44}$$

$$= \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T} r(\mathbf{s}_{i,t}) \tag{B.45}$$

given $p_\theta(\tau)$ can be (independently) sampled from, and $r(\mathbf{s}_t)$ is obtainable. We also recall trajectories τ are implicitly functions of θ. For our gradient term, we hence obtain a similar analytic form

$$\nabla_\theta\Psi(\theta) = \nabla_\theta\left[\int_\tau p_\theta(\tau)R(\tau)d\tau\right] \tag{B.46}$$

$$= \int_\tau \nabla_\theta p_\theta(\tau)R(\tau)d\tau \tag{B.47}$$

[6]While we relax the orthogonality constraint (as in PCA) on the decoder matrix to ensure full demixing between parameters is possible, this constraint is kept for the encoder matrix.

however we note this no longer takes the form of an expectation, and as such cannot be estimated as for the original objective above. This motivates the existence of policy gradient methods, which use the *log-trick* to rearrange this expression into a sampleable form (Section 2.3.2, Appendix B.2).

In our case, however, we have the additional option of estimating gradients by directly differentiating our *estimated* objective term $\hat{\Psi}(\theta)$. This is not usually considered as an option in RL problems, given the multiple sources of non-differentiability involved in computing return for each sampled trajectory $R(\tau)$, including (but not limited to):

- Non-differentiable reward function
- Probabilistic state transitions
- Probabilistic action sampling (i.e. stochastic policy)

However, for the renavigation task in this work each of these problems can be accounted for - our reward function is differentiable, and state transitions are deterministic conditioned on actions. Despite an obstensively stochastic policy due to additive motor noise (Section 3.2, eq. (3.3)), the final term can also be accounted for via the reparameterization trick, which we now return to.

Naively attempting to directly differentiate $\hat{\Psi}(\theta)$, it is not clear if $r(\mathbf{s}_t)$ has a well-defined differentive with respect to θ. While our reward function is differentiable with respect to $\mathbf{s}_t$ (Section 2.4.2, eq. (2.55)), examining the dependencies of $\mathbf{s}_t$ on θ by tracing back through the computational graph of our RL task (Appendix B.5), this differentiability appears to break down due to the sampling of each action $\mathbf{a}_t$. As such, it appears not to be possible to differentiate through each trajectory τ.

To recap, our agent's actions are sampled from a Gaussian policy - a process that can equally be considered as the addition of noise to our readout 'mean', as

$$\mathbf{a}_t = \alpha \mathbf{V}\mathbf{h}_t + \boldsymbol{\epsilon}, \quad \epsilon \sim \mathcal{N}(0, \sigma^2_{\text{motor}}) \tag{B.48}$$

The key insight of the reparameterization trick is then to reframe the stochasticity of our system - in this case the additive noise involved in the sampling process - as an external variable which is an *input* to the system and therefore does not need to be differentiated through. In other words, we consider each trajectory as a deterministic, differentiable function of some pre-sampled noise and initial input $\mathbf{x}$[7]

$$\tau = f_\theta(\boldsymbol{\epsilon}, \mathbf{x}), \quad \mathbf{x} = \{\mathbf{s}_0, \mathbf{h}_0\} \tag{B.49}$$

where in our case, $\mathbf{x}$ describes the initial environment and network state. Each noise value is then pre-sampled from a standard normal distribution

$$\epsilon \sim \mathcal{N}(0, 1) \tag{B.50}$$

and our overall objective can now be written as an expectation over this noise distribution, as

$$\Psi(\theta) = \mathbb{E}_{\epsilon \sim \mathcal{N}(\cdot)} \left[R(f_\theta(\boldsymbol{\epsilon}, \mathbf{x})) \right] \tag{B.51}$$

The code-level Monte Carlo estimate of our objective can hence be considered differentiable, with form

$$\nabla_\theta \hat{\Psi}(\theta) = \frac{1}{N} \sum_i^N \nabla_\theta R(f_\theta(\boldsymbol{\epsilon}_i, \mathbf{x}_i)) \tag{B.52}$$

[7] Adopting notation from the Variational Autoencoder (VAE) literature where this technique was originally employed [176].

Having demonstrated the differentiability of our estimated objective, we can finally comment on to what degree this is a practical estimator for the gradients required by our optimisation algorithm. Specifically we can compare this gradient estimator to that obtained via policy gradient methods. While both methods yield identical gradients in the limit of large N[8], overall we consider this direct differentiation approach to be superior, and it is therefore used at code-level in our work (Appendix D.1). The two main reasons for this can be summarised as follows:

- Lower variance: Policy gradient methods are notorious for their high variance due to the strong variation in returns across sampled trajectories, exacerbated by the log probability term. The direct differentiation method has considerably lower variance in comparison, considering only variance of the total returns themselves. It therefore requires fewer samples to be an effective estimator.
- Simpler implementation: While policy gradients require a complex algorithmic implementation, often with separate actor and critic components (Section 2.3.4, Algorithm 2), the direct differentiation approach involves simply backpropagating through the estimated objective itself. As such, it is much easier to finetune and implement.

B.5 Differentiability of RL Objective for Navigation Task

The objective used in the navigation task in Chapter 3

$$J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_t^T \left(r(\mathbf{s}_t) + \lambda_1 e^{\sum[\cos(\mathbf{x}_t^{\text{reward}} - \hat{\mathbf{x}}_t^{\text{reward}})]} - \lambda_2 \sum_j y_j \log \hat{y}_j \right) \right] \tag{B.53}$$

is fully differentiable, given it is assembled from differentiable component functions and the additive 'motor noise' injected does not impede this thanks to the reparameterization trick (Appendix B.4). As an exercise, we can also show this property explicitly by tracing back through the computations involved in assembling the full task objective, to show the full gradient tensor is composed of elements each with closed form differential expressions. We start by considering the full, verbose form of the *estimated* task objective implemented at code level

$$\hat{J}(\theta) = \frac{1}{N} \sum_{n=1}^{N} \sum_{t=1}^{T} \left(e^{\kappa_r (\sum[\cos(\mathbf{x}_t^{\text{reward}})] - 2)} + \lambda_1 e^{\sum[\cos(\mathbf{x}_t^{\text{reward}} - \hat{\mathbf{x}}_t^{\text{reward}})]} - \lambda_2 \sum_j y_j \log \hat{y}_j \right) \tag{B.54}$$

Considering each component of the loss separately, we then have via the chain rule[9]

$$\nabla_\theta \hat{J}(\theta) = \frac{1}{N} \sum_{n=1}^{N} \sum_{t=1}^{T} \left(\frac{\partial J_{\text{reward}}}{\partial \theta} + \frac{\partial J_{\text{predict}}}{\partial \theta} + \frac{\partial J_{\text{crossentropy}}}{\partial \theta} \right) \tag{B.55}$$

$$= \frac{1}{N} \sum_{n=1}^{N} \sum_{t=1}^{T} \left(\frac{\partial J_{\text{reward}}}{\partial \mathbf{x}_t^{\text{reward}}} \frac{\partial \mathbf{x}_t^{\text{reward}}}{\partial \theta} + \left(\frac{\partial J_{\text{predict}}}{\partial \mathbf{x}_t^{\text{reward}}} \frac{\partial \mathbf{x}_t^{\text{reward}}}{\partial \theta} + \frac{\partial J_{\text{predict}}}{\partial \hat{\mathbf{x}}_t^{\text{reward}}} \frac{\partial \hat{\mathbf{x}}_t^{\text{reward}}}{\partial \theta} \right) + \frac{\partial J_{\text{crossentropy}}}{\partial \theta} \right) \tag{B.56}$$

The first two losses have closed form gradients with respect to $\mathbf{x}_t^{\text{reward}}$ as

$$\frac{\partial J_{\text{reward}}}{\partial \mathbf{x}_t^{\text{reward}}} = \kappa_r e^{\kappa_r (\sum[\cos(\mathbf{x}_t^{\text{reward}})] - 2)} \left(-\sin(\mathbf{x}_t^{\text{reward}}) \right) \tag{B.57}$$

[8] More specifically, with an infinite number of samples, both estimators would converge to the 'true' gradient of our objective.

[9] Note the slight abuse of notation - the J notation typically refers to a theoretical objective, where in this case we use it to simply represent the formula for each component of the loss, i.e. with no estimation taken.

and

$$\frac{\partial J_{\text{predict}}}{\partial \mathbf{x}_t^{\text{reward}}} = \lambda_1 e^{\sum\left[\cos(\mathbf{x}_t^{\text{reward}} - \hat{\mathbf{x}}_t^{\text{reward}})\right]} \left(-\sin(\mathbf{x}_t^{\text{reward}})\right) \tag{B.58}$$

where in the latter case the derivative with respect to $\hat{\mathbf{x}}_t^{\text{reward}}$ also takes a similar form. We next consider the $\mathbf{x}_t$ gradient term with respect to θ. To recap from Chapter 3, we have

$$\mathbf{x}_t^{(j)} = \mathbf{x}_0^{(j)} - \mathbf{p}_t, \quad \forall j \tag{B.59}$$

$$\mathbf{p}_t = \mathbf{p}_{t-1} + \mathbf{a}_{t-1} \tag{B.60}$$

$$\mathbf{a}_t = \mathbf{V}\mathbf{h}_t + \sigma_{\text{motor}}\boldsymbol{\epsilon} \tag{B.61}$$

and therefore

$$\frac{\partial \mathbf{x}_t^{\text{reward}}}{\partial \theta} = -\frac{\partial \mathbf{p}_t}{\partial \theta} \tag{B.62}$$

$$= -\left(\frac{\partial \mathbf{p}_{t-1}}{\partial \theta} + \frac{\partial \mathbf{a}_{t-1}}{\partial \theta}\right) \tag{B.63}$$

$$= -\left(\frac{\partial \mathbf{p}_0}{\partial \theta} + \sum_{t'=0}^{t-1} \frac{\partial \mathbf{a}_{t'}}{\partial \theta}\right) \tag{B.64}$$

$$= -\sum_{t'=0}^{t-1} \frac{\partial \mathbf{a}_{t'}}{\partial \theta} \tag{B.65}$$

The dependence of $\mathbf{a}_t$ on θ is implicit from eq. (B.58) above, however to be verbose we can explicitly trace back the dependence of $\mathbf{a}_t$ on each element of theta

$$a_{t,i} = \sum_j v_{i,j} h_{t,j} + \sigma_{\text{motor}} \epsilon_i \tag{B.66}$$

where as per the reparameterization trick we do not consider our action as sampled from an implicit Gaussian, but instead as subject to additive noise which can be ignored at the point of differentiation. We hence have via the chain rule

$$\frac{\partial a_{t,i}}{\partial \theta_k} = \sum_j v_{i,j} \frac{\partial h_{t,j}}{\partial \theta_k} + h_{t,j} \frac{\partial v_{i,j}}{\partial \theta_k} \tag{B.67}$$

where each gradient term is an element of a wider gradient tensor $G^{a\theta}$[10] as

$$G_{i,k}^{a\theta} = \frac{\partial a_{t,i}}{\partial \theta_k} \tag{B.68}$$

and the derivative of each element $v_{i,j}$ is

$$\frac{\partial v_{i,j}}{\partial \theta_k} = \delta_{v_{i,j},\theta_k} \tag{B.69}$$

where $\delta_{v_{i,j}}$ is the Kronecker delta, denoting this derivative is one for $v_{i,j} = \theta_k$, and zero otherwise. The derivative of $h_{t,j}$ can then be examined by considering the GRU equations in a functional form i.e

$$\mathbf{h}_t = \text{GRU}(\mathbf{o}_t, \mathbf{h}_{t-1}; \theta) \tag{B.70}$$

[10]Specifically, we refer to this gradient tensor at time t.

Considering gradient contributions from each variable, including θ, we then have

$$\frac{\partial \mathbf{h}_t}{\partial \theta} = \frac{\partial \text{GRU}}{\partial \mathbf{o}_t}\frac{\partial \mathbf{o}_t}{\partial \mathbf{x}_t}\frac{\partial \mathbf{x}_t}{\partial \theta} + \frac{\partial \text{GRU}}{\partial \mathbf{h}_{t-1}}\frac{\partial \mathbf{h}_{t-1}}{\partial \theta} + \frac{\partial \text{GRU}}{\partial \theta} \tag{B.71}$$

where $\mathbf{o}_t$ represents the stacked vector of observations $\mathbf{o}_t^{(i)}$, and $\mathbf{x}_t$ is similarly a vector of all objects $\mathbf{x}_t^{(j)}$. The GRU gradient terms are clearly differentiable given the fact all operations in the GRU are either linear operations (with similar gradients to those in eq. (B.64)) or differentiable activation functions (Section 2.1.2). A more involved examination to determine a closed form expression for these gradients is therefore omitted. For the $\mathbf{o}_t$ term, we first consider recall eq. (2.53) from Section 2.4.2

$$\mathbf{o}_t^{(i)} = \sum_{j=1}^{N_x}\left(e^{\kappa\left(\sum\left[\cos(\mathbf{x}_t^{(j)} - \boldsymbol{\mu}^{(i)})\right]-2\right)}\right)\mathbf{c}^{(j)} + \boldsymbol{\epsilon} \tag{B.72}$$

We then obtain the following expression for a given element of the gradient tensor $G_{i,j}^{ox}$

$$\frac{\partial \mathbf{o}_t^{(i)}}{\partial \mathbf{x}_t^{(j)}} = \kappa\left(e^{\kappa\left(\sum\left[\cos(\mathbf{x}_t^{(j)} - \boldsymbol{\mu}^{(i)})\right]-2\right)}\right)\mathbf{c}^{(j)} \otimes (-\sin(\mathbf{x}_t^{(j)} - \boldsymbol{\mu}^{(i)})) \tag{B.73}$$

where $\otimes$ is the outer product[11]. Next examining the $\mathbf{h}_{t-1}$ gradient term in eq. (B.67) we note the recursive structure to these gradients, which are hence clearly differentiable by a similar argument to eqs. (B.58-B.60), given the gradient of $\mathbf{h}_0$ cancels. Finally, we acknowledge the derivative of the $\mathbf{x}$ term is formed from the derivatives of each $\mathbf{x}^{(j)}$ as described by eq. (B.62), which eventually leads to a recursive form based on its dependence on $\mathbf{h}_t$ eq. (B.62 - B.67). Next we examine the gradients of the $\hat{\mathbf{x}}^{\text{reward}}$ component with respect to θ

$$\frac{\partial \hat{\mathbf{x}}_t^{\text{reward}}}{\partial \theta} = \frac{\partial \hat{\mathbf{x}}_t^{\text{reward}}}{\partial \mathbf{x}_t^{\text{readout}}}\frac{\partial \mathbf{x}_t^{\text{readout}}}{\partial \theta} \tag{B.74}$$

where $\hat{\mathbf{x}}_t^{\text{reward}}$ is a 2-vector which attempts to predict the location of $\mathbf{x}_t^{\text{reward}}$, and $\mathbf{x}_t^{\text{readout}}$ is a readout 4-vector taken from the hidden state as

$$\mathbf{x}_t^{\text{readout}} = \mathbf{D}\mathbf{h}_t \tag{B.75}$$

which we consider to have form $\mathbf{x}_t^{\text{readout}} = [\sin(\hat{x}_t), \cos(\hat{x}_t), \sin(\hat{y}_t), \cos(\hat{y}_t)]$, and is then used to make predictions as[12]

$$\hat{\mathbf{x}}_t^{\text{reward}} = \left(\arctan 2\left(\frac{\sin(\hat{x}_t)}{\cos(\hat{x}_t)}\right), \arctan 2\left(\frac{\sin(\hat{y}_t)}{\cos(\hat{y}_t)}\right)\right) \tag{B.76}$$

The differential term with respect to $\mathbf{x}_t^{\text{readout}}$ is then a 2×4 Jacobian of form

$$\frac{\partial \hat{\mathbf{x}}_t^{\text{reward}}}{\partial \mathbf{x}_t^{\text{readout}}} = \begin{bmatrix} \frac{\cos(\hat{x}_t)}{\sin(\hat{x}_t)^2+\cos(\hat{x}_t)^2} & \frac{-\sin(\hat{x}_t)}{\sin(\hat{x}_t)^2+\cos(\hat{x}_t)^2} & 0 & 0 \\ 0 & 0 & \dots & \dots \end{bmatrix} \tag{B.77}$$

$$\approx \begin{bmatrix} \cos(\hat{x}_t) & -\sin(\hat{x}_t) & 0 & 0 \\ 0 & 0 & \cos(\hat{y}_t) & -\sin(\hat{y}_t) \end{bmatrix} \tag{B.78}$$

[11]This derivative represents the Jacobian formed by differentiating $\mathbf{x}_t^{(j)} \in \mathbb{R}^3$ with respect to $\mathbf{o}_t^{(i)} \in \mathbb{R}^2$, which in this case can be expressed with the outer product form above.

[12]This is a predictive readout where we assign labels $\sin(\hat{x})$ etc. to each element but the readout does not apriori have any notion of sin or cos: over the course of training these readout values better approximate the true $\sin()$ and $\cos()$ of the predicted targets location.

The $\mathbf{x}_t^{\text{readout}}$ gradient term with respect to θ then has a similar form to that of eq. (B.62), where each element of the differential can be considered as part of the gradient tensor $G_{i,k}^{x\theta}$ as

$$G_{i,k}^{x\theta} = \frac{\partial x_{t,i}}{\partial \theta_k} \tag{B.79}$$

Finally, we consider the gradient of the cross entropy loss term. Noting each $\hat{y}_j$ term in the loss is computed from the softmax activation of a hidden state readout, we have

$$\hat{\mathbf{y}}_t = \text{Softmax}(\mathbf{z}_t) \tag{B.80}$$

$$\mathbf{z}_t = \mathbf{W}\mathbf{h}_t + \mathbf{b} \tag{B.81}$$

and hence considering contributions to the gradient from both $\mathbf{h}_t$ and network parameters, we have

$$\frac{\partial J_{\text{crossentropy}}}{\partial \theta} = \frac{\partial J}{\partial \hat{\mathbf{y}}} \frac{\partial \hat{\mathbf{y}}}{\partial \mathbf{z}} \frac{\partial \mathbf{z}}{\partial \mathbf{h}_t} \frac{\partial \mathbf{h}_t}{\partial \theta} \tag{B.82}$$

Working backwards, the final term was already explored in eq. (B.67) and the second last term clearly evaluates to $\mathbf{W}$. For the second term, we can consider the mathematical form of the softmax function, denoted $\sigma(z_i)$, as

$$\hat{y}_i = \sigma(z_i) = \frac{e^{z_i}}{\sum_j e^{z_j}} \tag{B.83}$$

Carefully differentiating this term via the quotient rule, we obtain

$$\frac{\partial \hat{y}_i}{\partial z_j} = \frac{\delta_{i,j} e^{z_i} \sum_k e^{z_k} - e^{z_i} e^{z_j}}{\left(\sum_k e^{z_k}\right)^2} = \frac{\delta_{i,j} e^{z_i}}{\sum_k e^{z_k}} - \frac{e^{z_i} e^{z_j}}{\left(\sum_k e^{z_k}\right)^2} \tag{B.84}$$

$$= \frac{e^{z_i}}{\sum_k e^{z_k}} \left(\delta_{i,j} - \frac{e^{z_j}}{\sum_k e^{z_k}} \right) \tag{B.85}$$

$$= \sigma(z_i) \left(\delta_{i,j} - \sigma(z_j) \right) \tag{B.86}$$

and can hence assemble the full Jacobian as required. Finally, given the summation form of the (multi-class) cross entropy loss in eq. (B.50), we can obtain an expression for the first term in eq. (B.79)

$$J_{\text{crossentropy}} = -\sum_i y_i \log \hat{y}_i \tag{B.87}$$

$$\frac{\partial J}{\partial \hat{y}_i} = -y_i \frac{1}{\hat{y}_i} \tag{B.88}$$

$$\frac{\partial J}{\partial \hat{\mathbf{y}}} = -\frac{\mathbf{y}}{\hat{\mathbf{y}}} \tag{B.89}$$

We have then considered all elements of the derivatives for each of the functions in the overall loss term in eq. (B.50), showing there exists a closed form expression for the gradient of our sampled objective $\hat{J}(\theta)$, which can hence be directly optimised with backpropagation.

Appendix C

Computational Implementation

C.1 Automatic Differentiation

All simulations in this work were performed with Jax 0.2-0.4 [177] (alongside Optax [178] and Chex [178]) in Python 3.8 [179]. The Python library dPCA [180] was used for dPCA analysis. Jax is a machine learning framework which implements automatic differentiation ('autodiff') alongside XLA - a linear algebra compiler enabling GPU acceleration. Together with its set of useful function transformations, it is a powerful library for accelerating the optimisation of neural networks. For example $\mathrm{vmap}()$ enables efficient vectorisation, with associated performance improvements, and $\mathrm{jvp}()$ elegantly implements autodiff for a multivariate function, providing derivatives for each output. The latter is particularly useful when dealing with actor and critic losses which may be updated asynchronously, as in our PPO implementation. Some examples of Jax code used in our simulations are given in Appendix D.

C.2 Optimisation Details

This section provides a more detailed account of the optimisation procedure for each task, omitted from the main work for brevity. These training processes often involve iteratively finetuning a number of simulation parameters - both training hyperparameters, such as learning rate and batch size, and environment parameters, such as reward function length scale and magnitude of additive noise.

C.2.1 Navigation Task

The optimisation process for the navigation task (Chapter 3) is relatively simple given the fully differentiable form of its objective (Appendix B.4,B.5). In terms of parameter finetuning, most parameters were chosen after brief preliminary testing and fixed throughout the optimisation process; a selection of these are given below (Table C.1). The only parameters for which training was extensively optimised over were reward function length scale κ_r, and 'motor noise' σ_{motor}. For the reward function length scale experiments were done to implement scheduling for this term, slowly reducing its value over training such that the agent receives a stronger training signal earlier during training, which slowly decreases as the agent obtains more experience. For the motor noise magnitude, experiments were similarly performed over a range of values to determine the optimal additive noise for exploration, as visualised in Figure C.1.

Optimisation was performed via backpropagation and gradient descent using the 'Adamw' optimiser, a

modified version of the Adam optimiser (Section 2.2.1) incorporating weight decay, helping to stabilise training in lieu of a more explicit regularisation term in the objective[1].

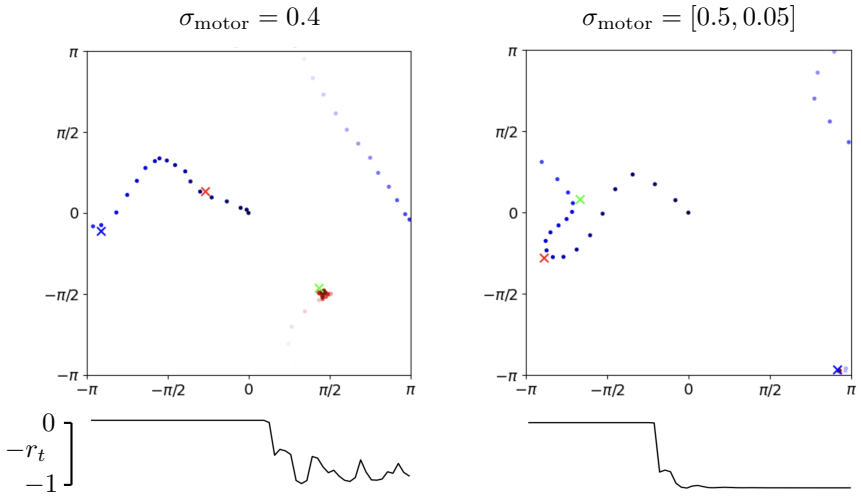


Figure C.1: **Effect of motor noise during navigation task.** Left: Post-training behaviour under constant high motor noise, $\sigma_{\text{motor}} = 0.4$. As in the main text, we describe the agent's behaviour across a single trial with a blue-red colourmap, and reward timeseries is appended below the figure. Motor noise magnitude must initially be set high - if too low (e.g. $\sigma_{\text{motor}} < 0.1$) the agent learns slowly due to poor exploration of the state space. However if σ_{motor} remains high throughout training we obtain poor post-training behaviour, where the agent struggles to 'settle' on the discovered target due to additive noise perturbing the agent at each timestep, as depicted. This problem is overcome by scheduling σ_{motor}, where we begin with high noise magnitude to encourage adequate exploration and lower this across training to ensure strong post-training performance. Right: Post-training behaviour for scheduled motor noise, $\sigma_{\text{motor}} = [0.5, 0.05]$. We see a clear improvement in performance for a post-training trial obtained under scheduled noise, where the agent exhibits macroscopically smoother behaviour and is now able to remain static at the discovered reward-maximising location.

Table C.1: **Simulation parameters for the navigation task.** Symbols match those in the main text, where applicable. Interval values for reward function scale κ_r and motor noise σ_{motor} reflect their chosen scheduling ranges across training, implemented via exponential decay (or growth).

Category	Parameter	Value
Hyperparameters	Hidden units, H	80
	Training epochs, E	$10,000$
	Trial length, T	60
	Learning rate	$5e-4$
	Batch size, N	600
	Weight decay	$2e-4$
	Predictive loss weight, λ_1	0.05
	Cross entropy loss weight, λ_2	0.02
Environment Parameters	Number of dots, N_x	3
	Number of neurons, N_o	100
	Activation fnc. scale, κ	2
	Reward fnc. scale, κ_r	$[1, 2.5]$
	Motor noise, σ_{motor}	$[0.5, 0.05]$
	Step size, α	0.04

[1]Such a method provides more effective regularisation when using an adaptive optimiser - such as Adam - where weight decay (shrinkage of the weights by a small amount at each iteration) can be decoupled from adaptive learning rates for each parameter.

C.2.2 Renavigation Task

For the renavigation task, training was performed only for a short period to ensure any observed task behaviour could be attributed largely to its capacity for adaptation, as built during the previous task. Specifically, we use a heuristic training length of a factor of 10 less than that of navigation task training, as well as removed the contribution of each auxiliary predictive loss during this task, to ensure our agent's adaptive behaviour leveraged representations learned during the initial training period. Such a paradigm enables us to consider the *meta-learning* capability of our agent.

Task parameters were largely kept identical to the tracking task, and reward function length scale was held constant at its final scheduled value. However, trial length was increased from 60 to 200 timesteps, to enable multiple teleportations and renavigations per trial. All modified and additional hyperparameters used in this task are displayed in Table C.2.

Optimisation was, however, performed over the parameters of the probabilistic teleportation function introduced in this task, of form

$$p(\text{teleport}|r_t) = \alpha_1 r_t - \alpha_2 r_t^2, \quad \text{where} \quad \alpha_1 > \alpha_2, \quad r_t \propto e^{-\frac{\kappa_r}{2}\|\mathbf{x}_t^{\text{reward}}\|^2} \tag{C.1}$$

This was necessary to find optimal values for α_1 and α_2 which ensured the agent would teleport roughly at the point of reaching the rewarded target during each renavigation sub-trial, whilst still incentivising direct navigation to the rewarded location (Chapter 3, Figure 3.2). A visual comparison of this parameter tuning process is depicted in Figure C.2.

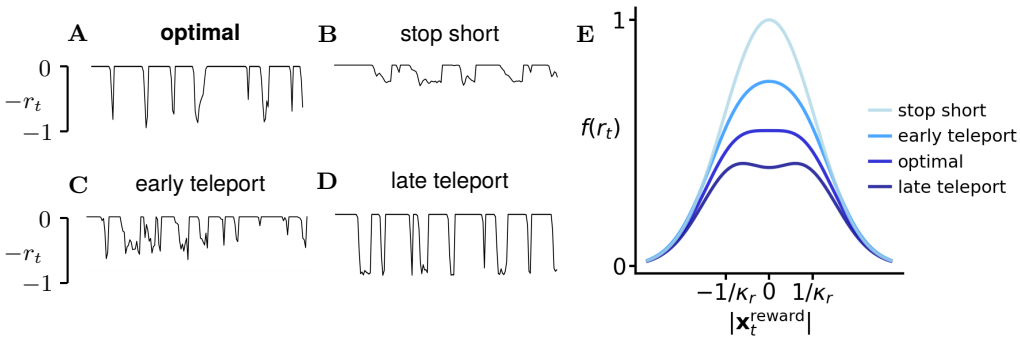


Figure C.2: **Parameter tuning of the teleportation function.** The parameters α_1, α_2 of the probabilistic teleportation function $p(\text{teleport}|r_t)$ (eq. (C.1)) must be carefully chosen to produce the desired behaviour, as described in the main text. Each plot in the 2×2 arrangement on the left ($\mathbf{A} - \mathbf{D}$; arbitrary order) displays an example of the reward timeseries obtained post training under a certain parameter setting. The graph on the right hand side **(E)** plots $p(\text{teleport}|r_t)$ corresponding to each parameter setting. **(A)** Optimal setting ($\alpha_1 = 1.1, \alpha_2 = 0.55$). This setting results in consistent teleportations as the agent navigates directly to the centre of the reward, as discussed in Figure 3.2 of the main text. **(B)** Suboptimal - agent *stops short* of the goal ($\alpha_1 = 1.1, \alpha_2 = 0.1$). If α_2 is decreased significantly from the optimal value (i.e. if the probability curve becomes increasingly proportional to reward) the relative probability of teleporting is increased at the centre of the reward, decentivising direct navigation to the target. As such, our agent learns the unintended behaviour of stopping short of the target center, obtaining lower reward magnitude but over a longer period of time in each trial, thereby resulting in fewer teleports triggered per trial. **(C)** Suboptimal - agent *teleports early* ($\alpha_1 = 1.2, \alpha_2 = 0.45$). If α_1 is increased or α_2 is decreased (or both), the probability of teleportation becomes too high resulting in 'early' teleportation - when the agent teleports on approach, before the it reaches the centre of the target. **(D)** Suboptimal - agent *teleports late* ($\alpha_1 = 1.0, \alpha_2 = 0.6$). If α_1 is decreased or α_2 is increased (or both), the probability of teleportation similarly becomes too low, resulting in 'late' teleportation. This is when the agent navigates directly to the target but teleportation takes a few timesteps to trigger, once again representing unintended behaviour.

Table C.2: **Simulation parameters for the renavigation task**. Trial length and training time are changed, and auxiliary losses now ignored. Previously scheduled parameters are kept at their final values.

Category	Parameter	Value
Hyperparameters	Training epochs, E	1000
	Trial length, T	200
	Predictive loss weight, λ_1	0
	Cross entropy loss weight, λ_2	0
Environment Parameters	Reward fnc. scale, κ_r	2.5
	Motor noise, σ_{motor}	0.05
	Teleportation fnc. parameter 1, α_1	1.1
	Teleportation fnc. parameter 2, α_2	0.55

C.2.3 Visual Model Pre-training

For the tracking task in Chapter 4, we initially pre-train a visual predictive model to give our agent the option of covertly *planning* the outcome of a movement as an alternative to making an overt movement (Figure 4.1). This model is then considered to represent the agent's internal, *simulated* environment. Training for this model is performed across a number of offline pre-generated timeseries[2], as described in the main text and Figure 4.2. The agent's role was then to predict visual activations across the full space at each timestep, based on partial within-aperture observations up until that point. These could consist of either 'true' basis function activations or predicted activations, based on if the agent made a movement or plan at each timestep (as dictated by the pre-generated timeseries).

Two main problems were found during the training of this model. First, to limit unnecessary complexity the model initially chosen for this task was a vanilla RNN (Section 2.1.1) with $H = 300$, assuming this relatively simple prediction task - predicting next state from an internal representation formed across previous (partial) state observations - should not explicitly require gating. However a significant problem became clear with this approach. To preface this, it is important to clarify that when internally simulating the environment (Figure 4.1) the agent simulates the visual activations it would encounter at a given sequence of *simulated* locations, however the subsequent movement made post-planning is made relative to the agent's *true* position, kept constant during the plan. The problem encountered was then that the agent's prediction performance would drop significantly for the first movement following a planning phase, due to the difficulty involved with switching context from planning back to movement - where the agent must integrate a new movement command with its 'old' true position (Figure C.3). In other words, to successfully perform the prediction task the agent must learn to internalise its true position when planning, then combine this with the subsequent movement command received post-planning to correctly predict visual activations in the new location following this movement. This becomes an increasingly difficult task for longer planning periods, where the vanilla RNN model struggles to learn this long term dependency and successfully retain the initial position information. To solve this problem we therefore introduced a Minimal Gated Unit (MGU) model, a reduced complexity version of the full GRU where the update and reset gates are merged to a single 'forget' gate [115]. Such a model exhibits the gating required to successfully learn the long term dependencies necessary for this prediction task.

An additional problem encountered was that despite learning well for the case of no movement refractory period ($P = 0$), when a refractory period was introduced the agent's training performance would become stuck in a local minima, plateauing after a certain point and never reaching the optima. Observing model performance empirically[3] the issue became clear: the agent was learning to accurately predict visual

[2]That is, we use static, pre-generated timeseries for training that are assembled before training begins. Such a method provides significant performance improvements given Jax's efficient computation of static data.

[3]As in Figure 4.2, based on the fit of the predictive ellipse.

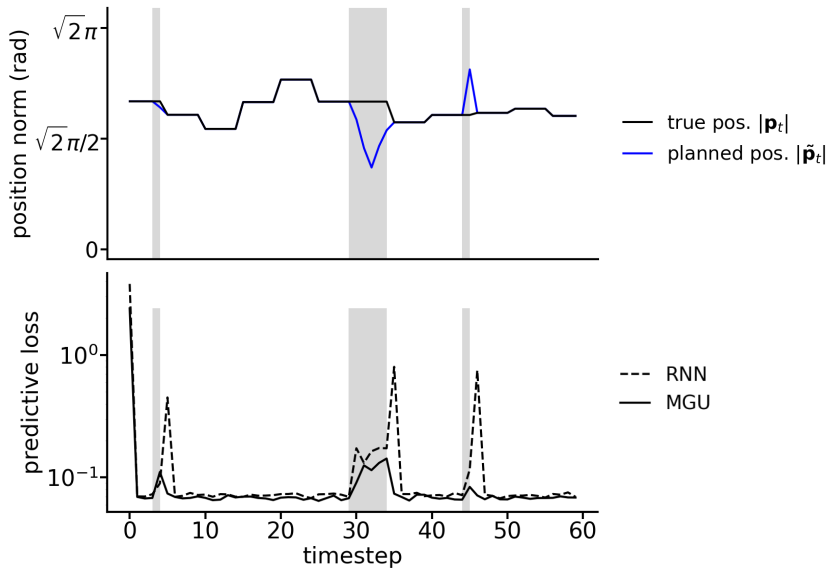


Figure C.3: **Visual model pre-training timeseries.** Top: Timeseries describing the training paradigm. As described in the main text, offline training timeseries consist of randomly sampled movements and plans as dictated by a random variable d_t, with fixed probability α of switching between the two at each non-refractory period timestep. The top plot visualises this process, showing sampled plans $d_t = 1$ with grey shading. Y-axis describes agent position as Euclidean norm with respect to $(0, 0)$ on the torus. The two plots describe both 'true' position (black), which changes only during overt movements, and planned position (blue), identical to the former aside from during planning phases, at which point it deviates. Refractory periods can be observed as static periods in between each movement. Bottom: Timeseries describing the corresponding predictive loss. Loss is plotted for both vanilla RNN and MGU models, post-training. Loss is calculated as the summed squared error over all neuron activations, for each of the 60 steps in the timeseries. During planning periods we notice a steady increase in error, as might be expected given input information is now fictitious, and so error slowly compounds. As discussed in the main text, we observe the MGU model (solid line) solves the issue where predictive performance for the RNN (dashed line) drops significantly for the first post-plan prediction, due to difficulties integrating old positional information with a new movement command during the plan-move context switch.

activations describing target movement only during the refractory period following a movement, where the agent would remain 'static' for $P - 1$ timesteps. This corresponds to an easier task: by adjusting its predicted activations at each timestep in the refractory period to reflect target movement, the agent would need to learn only the target velocity, and never need to learn the motor map from one-hot motor commands to movements vectors as required to predict post-movement activations. Given a refractory period of reasonable length, the agent would hence obtain sufficiently strong predictive performance at this easier task to fall into a local minima[4], never learning the harder task of predicting post-movement activations by learning the motor map. To solve this problem, a curriculum learning approach was employed, where the model was first trained with no refractory period (where movements and plans took a single step alike) to ensure it successively learnt the motor map between one-hot commands and movements as necessary to predict post-movement activations. After training under the refractory period paradigm was then re-introduced, at which point the model rapidly converged to optimal performance. Parameters for this model's training process are displayed below in Table C.3.

[4]In other words, this suggests the objective in Chapter 4 (eq. (4.3)) can be minimised faster initially with this simple strategy, than with the intended motor map-based strategy.

Table C.3: **Simulation parameters for visual model pre-training.** Bracketed value for number of neurons represents activations observed within the agent's aperture, out of the total spanning the space.

Category	Parameter	Value
Hyperparameters	Hidden units (predictive network), H_{predict}	300
	Training epochs, E	$10,000$
	Trial length, T	80
	Learning rate	$8e-4$
	Batch size, N	200
	Move-plan switching probability, α,	0.2
Environment Parameters	Number of neurons, N_o	144 (32)
	Action space cardinality, $\lvert S_a \rvert$	81
	Activation fnc. scale, κ	2
	Refractory period, P	5
	Aperture width	$\pi/2$
	Target speed ratio, R	1.5

C.2.4 Tracking Task

Optimisation of the agent's policy for the tracking task also posed a number of challenges. For the code level implementation, difficulties were found adapting Jax methods - which have a preference for static computations[5] - to the dynamic graphs required by this task, where decisions to move, plan, or be in a refractory period are made dynamically at *runtime*. The implementation used involved nesting multiple `lax.cond()` statements together (each acting as a conditional 'fork') to account for the multiple decisions available at each timestep. Each potential trajectory is then accounted for by a pre-compiled static graph capable of dynamically choosing the correct runtime branches. This approach however suffered from memory issues, where memory allocation grew with number of training epochs until the process ran out of memory. This problem was solved by manually clearing the compile cache every 100 epochs[6].

For our optimisation algorithm we used PPO (Section 2.3.4), a decision motivated by two main factors. First, the inherently high variance of policy gradient methods often leads to unstable optimisation, something improved upon by PPO. As detailed in the main text, the implicit trust region introduced by the clipped surrogate objective prevents large, potentially unstable updates[7] and was found empirically to provide better performance than other methods (Figure C.4A). Additionally, as described above our dynamic task structure does not benefit from the same performance improvements within Jax as with the static computations utilised by other optimisation tasks in this work. Therefore sample efficiency is particularly important, given the expensive cost of collecting trajectories. PPO excels in this regard, where multiple optimisation steps can be made for a single batch of 'old' trajectories collected. The 'new' trajectories required for each mini-batch update can then be quickly collected, given they are identical to the old trajectories and can hence be statically scanned through (i.e. with no runtime control flow).

A few key modifications were also made to our optimisation process to further improve performance, as detailed in Section 4.3.2 of the main text. First, Kullback-Leibler regularisation was incorporated to regularise the agent's decision to plan or act by penalising deviations in the agent's policy away from

[5] I.e. the transformations enabling large performance gains, such as just-in-time compilation with `@jit`, work best for static computations. In hindsight, a framework which better deals with dynamic graphs - e.g. PyTorch [181] - would have been more appropriate for this task.

[6] After profiling memory usage during the process, we hypothesise this issue was due to cached intermediate results created from certain `lax.cond()` branches gradually accumulating in memory. Clearing the compile cache could have then indirectly resulted in more efficient memory management, and these results being garbage collected.

[7] While gradient descent in supervised learning is often conceptualised as descending through parameter space to a global minima, policy gradients methods instead perform gradient ascent through a parameter space with steep curvature either side of the ascent path. A single poor step may therefore never be recovered from, so ensuring update stability is of utmost importance.

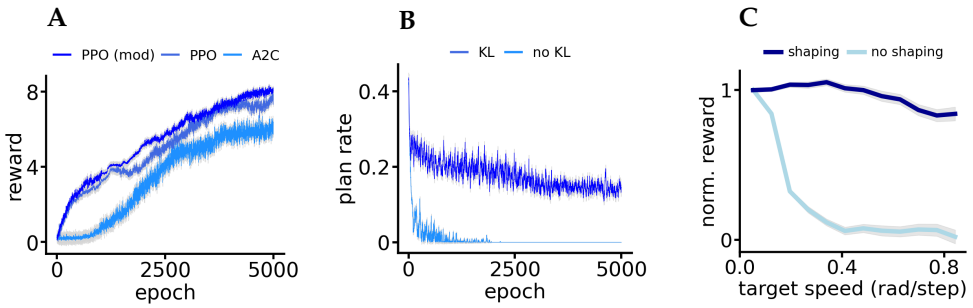


Figure C.4: **Tracking task optimisation. (A)** A2C vs PPO comparison. Average reward vs training epoch for A2C and PPO algorithms during preliminary testing over 5000 epochs. Plots show A2C (light blue), PPO (medium blue), and PPO with a modified critic (dark blue) - as detailed in the main text. Grey shading denotes standard error. We observe PPO outperforms A2C, exhibiting stronger initial performance and characteristically stable updates. PPO with a modified critic further improves performance and was selected for full training over $15,000$ epochs using parameters in Table C.4. Note while outperforming A2C, PPO proved much more challenging to fine-tune due to its high sensitivity to hyperparameter variation [97]. **(B)** KL regularisation for plan-move decision. Plots depict mean plan rate during preliminary testing for both an objective including a KL regularisation term over the move-plan decision (dark blue) and with no KL term (light blue). We see the KL plot initially hover around the 0.2 mass placed on planning in the KL prior, before slowly deviating. The curve with no KL term however quickly reduces to zero, indicating planning is of minimal use for the agent early in training. **(C)** Reward Shaping. Mean, normalised reward post-training (normalised with respect to reward obtained for the slowest target speed bin) plotted against target speed in the case of no reward shaping (constant reward across target speeds), and reward shaping as described in eq. (C.5). Optimising over the parameters of this function, we implement a reward shaping curve with $\alpha_1 = 9, \alpha_2 = 2.5$, such that similar mean reward is obtained across target speeds, post-training.

a specified prior distribution. For the vector component of the policy distribution we use a uniform prior, encouraging our agent to explore the full movement vector space when determining the optimal policy[8]. For the decision to move or plan, we heuristically place a $[0.8, 0.2]$ categorical prior over moving and planning, penalising deviations from a policy which wishes to move with 80% probability and plan with 20% probability (ignoring refractory periods). The logic is then that by encouraging a degree of uncertainty over this decision, the agent will naturally sample movements and plans in different contexts. Eventually, the agent should then learn the optimal contexts to move or plan and hence consistently choose the optimal action at each timestep irrespective of the prior[9]. This is important as without such regularisation there is no guarantee the agent will explore the full movement vector space or learn to appropriately leverage planning[10]. With this in mind, we can revisit the RL objective for this task and consider the full form of gradient updates. The full actor loss implemented at code level is then

$$J_{\text{actor}}(\theta) = J_{\text{PPO}}(\theta) + \lambda_1 \text{KL}(\pi_\theta^{\text{vec}} || \mathbf{p}^{\text{vec}}) + \lambda_2 \text{KL}(\pi_\theta^{\text{dec}} || \mathbf{p}^{\text{dec}}) \tag{C.2}$$

with corresponding actor gradient update

$$\Delta\theta_{\text{actor}} \propto \sum_k \left[\nabla_\theta \min\left(\cdot\,, \text{clip}(\cdot) \right)_k + \lambda_1 \sum_j \left(\log \frac{\pi_{k,j}^{\text{vec}}}{p_{k,j}^{\text{vec}}} + 1 \right) \nabla_\theta \pi_{k,j}^{\text{vec}} + \lambda_2 \sum_j \left(\log \frac{\pi_{k,j}^{\text{dec}}}{p_{k,j}^{\text{dec}}} + 1 \right) \nabla_\theta \pi_{k,j}^{\text{dec}} \right] \tag{C.3}$$

[8]The use of a uniform prior means this is effectively equivalent to an entropy regularisation term.

[9]In other words the KL term initially implicitly encourages a certain level of uncertainty, but as the agent learns the optimal policy it should learn to override the KL contribution and choose actions which maximise expected reward. This is observed across training, where policy entropy (i.e. uncertainty) slowly decreases alongside task performance as the agent becomes more confident in its actions.

[10]Given planning does not obtain reward, their is no apriori 'motivation' to employ this behaviour if it is not encouraged in some way, such as via KL regularisation - which imposes an explicit prior - or other techniques such as entropy regularisation.

where index k denotes datapoint i, t across trajectories in a mini-batch. Similarly for critic updates

$$\Delta\theta_{\text{critic}} \propto \sum_k \delta_k \nabla_\theta \delta_k, \quad \text{where} \quad \delta_t = r_t + \hat{V}_{t+1}^{\text{GAE}} - \hat{V}_t^{\text{GAE}} \tag{C.4}$$

To further stabilise training and improve performance we also introduce a number of modifications to our critic ('PPO (mod)', Figure C.4A). First, we use a multi-layer value estimate as

$$\hat{V}(\mathbf{h}_t) = \mathbf{V}_2(\text{ReLU}(\mathbf{V}_1\mathbf{h}_t + \mathbf{b}_1)) \tag{C.5}$$

Alternative critic structures were also tested - such as with an additional nonlinearity and bias term - but the structure described above represented the best tradeoff between performance and complexity (more complex variants were noticeably more challenging to train). A 'reverse counter' term

$$t' = 1 - \frac{t}{T} \tag{C.6}$$

was also appended to the pre-readout hidden state to help decorrelate time within trial from the value estimation, empirically improving performance early in training. Additionally, we introduced multiple small critic updates for each actor update to further stabilise training. The combined effect of each of the above modifications was to significantly improve critic performance, resulting in improved, stabilising training resulting overall (Figure C.4B). Hyperparameters used for training are displayed in Table C.4.

A final notable modification introduced when optimising the tracking task was that of reward shaping, where reward received by the agent is modified to be a function of target speed as

$$r_{\max} = 1 + \alpha_1 \|\mathbf{v}\|^{\alpha_2} \tag{C.7}$$

assigning greater value to faster targets, which may only be encountered a few times per trial. This was crucial in ensuring the agent learned a similarly optimal policy across target speeds (Figure C.4C).

Table C.4: **Simulation parameters for the tracking task.** We now consider separate hyperparameters governing actor and critic updates. Bracketed number for batch size denotes size of each mini-batch, and for training epochs denotes number of critic updates (a ratio of 3 compared to actor updates).

Category	Parameter	Value
Hyperparameters	Hidden units (policy network), H_{policy}	100
	Training epochs, E	$15,000$ $(45,000)$
	Trial length, T	60
	Learning rate (actor)	$3e-4$
	Learning rate (critic)	$8e-4$
	Weight Decay (actor)	0
	Weight Decay (critic)	$1e-6$
	Batch size, N (B)	1000 (200)
	Vector KL loss weight, λ_1	$[0.1, 0.01]$
	Decision KL loss weight, λ_2	$[0.5, 0.05]$
Environment Parameters	Number of neurons, N_o	144 (32)
	Action space cardinality, $\|S_a\|$	81
	Activation fnc. scale, κ	2
	Reward fnc. scale, κ_r	$[1, 2.5]$
	Refractory period, P	5
	Aperture width	$\pi/2$
	Target speed ratio, R	1.5

Appendix D

Example Code

The two code examples below - for navigation task and tracking task training - display examples of Jax code used to optimise the RL agents in this work, written in a functional style. The outer training loop and a few additional functions are shown for both examples, as well as an example main routine.

D.1 Training Loop for Navigation Task

```
### outer training loop and other primary functions ###

def full_loop(params, weights):
    """ full training loop for (fully differentiable) navigation task objective.

    Args:   params: training parameters (hyperparameters and environment parameters), pytree of scalars.
            weights: initial agent network weights, pytree of arrays.

    Returns:    loss_arrays: arrays to store losses, pytree of arrays.
                sem_arrays: arrays to store standard errors of the mean, pytree of arrays.
                opt_state: network optimiser, optimiser state.
                weights: trained agent network weights, pytree of arrays.
    """
    loss_keys = ['loss_tot', 'loss_predict', 'loss_CE']
    sem_keys = ['sem_tot', 'sem_predict', 'sem_CE']
    loss_arrays = {key: jnp.zeros((params["TOT_EPOCHS"],)) for key in loss_keys} # initialise loss arrays
    sem_arrays = {key: jnp.zeros((params["TOT_EPOCHS"],)) for key in sem_keys} # initialise standard error arrays

    optimiser = optax.adamw(learning_rate=params["LEARNING_RATE"], weight_decay=params["WEIGHT_DECAY"])
    opt_state = optimiser.init(weights) # initialise optimiser

    for epoch in range(params["TOT_EPOCHS"]):

        scan_params = new_scan_params(epoch) # generate new random environments

        (loss_total, aux), grads_ = get_losses_and_grads_vmap(scan_params["H0"], scan_params["DOTS"], scan_params["SELECT"], scan_params["EPSILON_N"], weights, params)
        grad_mean = jax.tree_util.tree_map(lambda x: jnp.mean(x,axis=0), grads_) # find mean gradient over batch
        opt_update, opt_state = optimiser.update(grad_mean, opt_state, weights)
```

```
        weights = optax.apply_updates(weights, opt_update)

        loss_total_mean = jnp.mean(loss_total)
        sem_total = jnp.std(loss_total) / jnp.sqrt(loss_total.shape[0])

        losses, _ = aux
        losses_mean = (jnp.mean(jnp.sum(x, axis=1)) for x in losses)
        sems = (jnp.std(jnp.sum(x, axis=1)) / jnp.sqrt(x.shape[0]) for x in losses)

        loss_arrays = {key: loss_arrays[key].at[epoch].set(losses_mean[key]) for key in loss_arrays.keys()} # populate loss arrays
        sem_arrays = {key: sem_arrays[key].at[epoch].set(sems[key]) for key in sem_arrays.keys()} # populate standard error arrays

        losses = jax.tree_map(lambda x: x.block_until_ready(), losses) # ensure computations complete before printing; saving memory profile
        sems = jax.tree_map(lambda x: x.block_until_ready(), sems)

        (mean_reward, loss_predict, loss_CE) = losses_mean
        (sem_rewards, sem_predict, sem_CE) = sems

        print(f"epoch={epoch} reward={mean_reward:.3f} loss_total={loss_total_mean:.3f} loss_predict={loss_predict:.3f} loss_CE={loss_CE:.3f}") # print information for debugging (optional)
        print(f"sem_rewards={sem_rewards:.3f} sem_total={sem_total:.3f} sem_predict={sem_predict:.3f} sem_CE={sem_CE:.3f}")

    return loss_arrays, sem_arrays, opt_state, weights

@partial(jax.jit, static_argnums=(5,))
def get_trajectory(h0, dots, select, epsilon_n, weights, params):
    """ calculate the total loss for a single trajectory.

    Args:   h0: initial hidden state for the agent network, array [H,].
            dots: locations of the 3 RGB objects, array [3,2].
            select: one-hot vector denoting the rewarded object in that trajectory, array [3,].
            epsilon_n: pre-generated motor noise for each step of the trajectory, array [T,2].
            weights: current agent network weights, pytree of arrays.
            params: training parameters (hyperparameters and environment parameters), pytree of scalars.

    Returns:    loss_total: total loss summed over a single trajectory, scalar.
                (losses, debug): individual losses and additional debugging information, tuple of tuples.
    """
    pos_t = jnp.array([0,0], dtype=jnp.float32) # initial position
    args_0 = (pos_t, h0, dots, select, weights, params)

    args_final, (losses, debug) = jax.lax.scan(single_step, args_0, epsilon_n)
    rewards, loss_predict, loss_CE = losses
    loss_total = - jnp.sum(rewards) - params["LAMBDA_PREDICT"] * jnp.sum(loss_predict) + params["LAMBDA_CE"] * jnp.sum(loss_CE) # assemble total (negative) loss

    return loss_total, (losses, debug)

get_loss_and_grad = jax.value_and_grad(get_trajectory, argnums=4, allow_int=True, has_aux=True)
```

```
get_losses_and_grads_vmap = jax.vmap(get_loss_and_grad, in_axes=(0, 0, 0, 0, None, None),
    out_axes=(0, 0)) # the first 4 arguments in get_trajectory() are vmap'd across their
    leading axis, length N

def single_step(args, epsilon):
    """ calculate the total loss for a single trajectory.

    Args:   args: current trajectory step carry, tuple of multiple types.
            epsilon: pre-generated motor noise for current step, array [2,].

    Returns:    args: next trajectory step carry, tuple of multiple types.
                (losses, debug): individual losses and additional debugging information, tuple
     of tuples.
    """
    pos_t, h_t, dots, select, weights, params = args

    activations_t1 = neuron_act(pos_t, dots, params) # neuron activations

    h_t1 = GRU_step(activations_t1, h_t, weights) # GRU equations

    v_t1 = params["ALPHA"] * (weights["V"] @ h_t1 + params["SIGMA_NOISE"] * epsilon) #
    velocity readout and 'motor noise'
    pos_t1 = pos_t + v_t1 # update position; new state

    rdot_hat_t1 = weights["D"] @ h_t1 # predict rewarded object location
    sel_hat_t1 = weights["S"] @ h_t1 # predict rewarded object index

    reward_t1 = get_reward(pos_t1, dots, select, params) # reward
    loss_predict_t1 = get_loss_predict(rdot_hat_t1, pos_t1, dots, params) # aux prediction
    loss
    loss_cross_entropy_t1 = get_loss_cross_entropy(sel_hat_t1, select, params) # aux decision
    loss

    args = (pos_t1, h_t1, dots, select, weights, params)
    losses = (reward_t1, loss_predict_t1, loss_cross_entropy_t1)
    debug = (activations_t1, pos_t1, rdot_hat_t1, sel_hat_t1)

    return args, (losses, debug)

### main routine ###

params = {
    # initialise simulation parameters including training hyperparameters (number of epochs,
    learning rates etc.) and environment parameters (noise magnitude, activation function
    scale etc.)
}

init_weights = {
    # initialise agent GRU network weights with normal glorot initialisation
}

start_time = datetime.now()
loss_arrays, sem_arrays, opt_state, final_weights = full_loop(params, init_weights)
print(f"Sim time: {datetime.now() - start_time}
s/epoch= {((datetime.now() - start_time) / params['TOT_EPOCHS']).total_seconds()}")

save_checkpoint((opt_state, final_weights), 'navigation_task_checkpoint_') # checkpoint
    training
```

```
125 save_outputs((loss_arrays, sem_arrays), 'navigation_task_outputs_') # save other outputs to
    separate file
```

Example Code D.1: **Training loop and other selected functions from `navigation_task_training.py`.** In this case the objective (line 71) is fully differentiable (Appendix B.5) and so our loop serves largely as a wrapper around the gradient-obtaining function `get_losses_and_grads_vmap` which calculates gradients across batched (vmapped) trajectories obtained using Jax's efficient `lax.scan` method for unrolling loops. The decorator `@jit` indicates a function should be compiled to a form optimised for efficient execution on a GPU (Appendix C.1). The `@partial` decorator provides additional flexibility to compile functions where the inputs are not all Jax arrays, by marking non-array arguments as *static* such that they are not traced by Jax during compilation. Note in Jax any nested combination of lists, tuples or dictionaries are referred to as *pytrees*, hence many inputs and outputs are denoted as such in the function descriptions. A 'pytree of arrays [N,T]' then refers to such a container where each leaf is a Jax array of shape [N,T]. Note finally use of `block_until_ready()` to force waiting for all (often asynchronous) operations to complete before non-jax operations are performed such as printing and memory profiling.

D.2 Training Loop for Tracking Task

```
1  ### training loop and other primary functions ###
2
3  def full_loop(params, init_weights_p ,weights_v):
4      """ full PPO training loop: outer loop over epochs, inner loop over mini batch updates.
5
6      Args:   params: training hyperparameters and environment parameters, pytree of scalars.
7              init_weights_p: initial policy network weights, pytree of arrays.
8              weights_v: (static) pre-trained visual model network weights, pytree of arrays.
9
10     Returns:    loss_arrays: arrays to store losses, pytree of arrays.
11                 sem_arrays: arrays to store standard errors of the mean, pytree of arrays.
12                 actor_opt_state: actor optimiser, optimiser state.
13                 critic_opt_state: critic optimiser, optimiser state.
14                 new_weights_p: trained policy network weights, pytree of arrays.
15     """
16     loss_keys = ['loss_tot', 'loss_actor', 'loss_critic', 'loss_kl_vectors', '
       loss_kl_decisions', 'mean_reward', 'mean_plan_rate']
17     sem_keys = ['sem_tot', 'sem_actor', 'sem_critic', 'sem_kl_vectors', 'sem_kl_decisions', '
       sem_rewards', 'sem_plan_rate']
18     TOT_EPOCHS = params["EPOCHS"] * params["VMAPS"] // params["BATCH_SIZE"]
19     loss_arrays = {key: jnp.zeros((TOT_EPOCHS,)) for key in loss_keys} # initialise loss
       arrays
20     sem_arrays = {key: jnp.zeros((TOT_EPOCHS,)) for key in sem_keys} # initialise standard
       error arrays
21
22     old_weights_p = copy.deepcopy(init_weights_p) # initialise old policy
23     new_weights_p = copy.deepcopy(init_weights_p) # initialise new policy
24
25     actor_optimiser = optax.chain(
26         optax.clip_by_global_norm(params["GRAD_CLIP"]),
27         optax.adam(learning_rate=params["ACTOR_LR"])
28         )
29     actor_opt_state = actor_optimiser.init(new_weights_p) # initialise actor optimiser
30
31     critic_optimiser = optax.chain(
32         optax.clip_by_global_norm(params["GRAD_CLIP"]),
33         optax.adamw(learning_rate=params["CRITIC_LR"], weight_decay=params["CRITIC_WD"])
```

```
        )
    critic_opt_state = critic_optimiser.init(new_weights_p) # initialise critic optimiser

    for epoch in range(params["EPOCHS"]): # outer optimisation loop

        scan_params = new_scan_params(epoch) # generate new batch of random environments

        old_trajectories, _ = get_old_trajectories_vmap(scan_params, params, old_weights_p, weights_v) # get total batch of old trajectories

        for b, batch in enumerate(range(0, params["VMAPS"], params["BATCH_SIZE"])): # inner mini batch update loop

            old_trajectories_batch = jax.tree_map(lambda x: x[batch: batch + params["BATCH_SIZE"]], old_trajectories) # select mini batch of old trajectories
            scan_params_batch = jax.tree_map(lambda x: x[batch: batch + params["BATCH_SIZE"]], scan_params)

            for _ in range(params["CRITIC_UPDATES"]): # multiple critic updates for each mini-batch of old/new trajectories

                isolated_batch_loss = lambda new_weights_p: get_batch_losses(old_trajectories_batch, scan_params_batch, params, new_weights_p, weights_v) # get_batch_losses as a univariate function of new_weights_p
                (loss_actor, loss_critic), get_actor_critic_grads, (losses, sems) = jax.vjp(isolated_batch_loss, new_weights_p, has_aux=True) # obtain jacobian-returning function

                critic_grad, = get_actor_critic_grads((0.0, 1.0)) # critic grads using vjp
                critic_update, critic_opt_state = critic_optimiser.update(critic_grad, critic_opt_state, new_weights_p)
                new_weights_p = optax.apply_updates(new_weights_p, critic_update) # critic update

            actor_grad, = get_actor_critic_grads((1.0, 0.0)) # actor grads using vjp
            actor_update, actor_opt_state = actor_optimiser.update(actor_grad, actor_opt_state)
            new_weights_p = optax.apply_updates(new_weights_p, actor_update) # actor update

            ind = epoch * (params["VMAPS"] // params["BATCH_SIZE"]) + b
            loss_arrays = {key: loss_arrays[key].at[ind].set(losses[key]) for key in loss_arrays.keys()} # populate loss arrays
            sem_arrays = {key: sem_arrays[key].at[ind].set(sems[key]) for key in sem_arrays.keys()} # populate sem arrays

            losses = jax.tree_map(lambda x: x.block_until_ready(), losses) # ensure computations complete before printing; saving memory profile
            sems = jax.tree_map(lambda x: x.block_until_ready(), sems)

            (loss_tot, loss_actor, loss_critic, loss_kl_vectors, loss_kl_decisions, mean_reward, mean_plan_rate) = losses
            (sem_tot, sem_actor, sem_critic, sem_kl_vectors, sem_kl_decisions, sem_reward, sem_plan_rate) = sems

            print(f"epoch={epoch}.{b} reward={mean_reward:.3f} sem_reward={sem_reward:.3f} loss_tot={loss_tot:.3f} sem_tot={sem_tot:.3f}")
            print(f"loss_actor={loss_actor:.3f} sem_actor={sem_actor:.3f} loss_critic={loss_critic:.3f} sem_critic={sem_critic:.3f}")
            print(f"loss_kl_vec={loss_kl_vectors:.3f} sem_kl_vec={sem_kl_vectors:.3f} loss_kl_dec={loss_kl_decisions:.3f} sem_kl_dec={sem_kl_decisions:.3f} plan={mean_plan_rate:.3f} sem_plan={
```

```
    sem_plan_rate:.3f}")  # print information for debugging (optional)
74          jax.profiler.save_device_memory_profile(f"memory_profile_epoch{epoch}.prof") #
    save memory profile for debugging (optional)
75
76          if ind > 0 and (ind % 100) == 0:
77              print("*clearing cache*")
78              jax.clear_caches() # clear compile cache every 100 iterations
79
80      old_weights_p = copy.deepcopy(new_weights_p) # set old weights to the weights
81
82  return loss_arrays, sem_arrays, actor_opt_state, critic_opt_state, new_weights_p
83
84 def get_batch_losses(old_trajectories_batch, scan_params_batch, params, new_weights_p,
    weights_v):
85  """ obtain actor and critic losses by comparing new and old trajectories.
86
87  Args:   old_trajectories_batch: batch of old trajectories, pytree of arrays [B,T].
88          scan_params_batch: parameters to batch over, pytree of arrays [B,T].
89          params: training hyperparameters and environment parameters, pytree of scalars.
90          new_weights_p: new policy network weights, pytree of arrays.
91          weights_v: (static) pre-trained visual model network weights, pytree of arrays.
92
93  Returns:    (loss_actor, loss_critic): actor and critic losses, tuple of scalars.
94              (losses, sems): auxiliary debugging information, tuple of pytrees.
95  """
96  new_trajectories, _ = get_new_trajectories_vmap(old_trajectories_batch, scan_params_batch,
     params, new_weights_p, weights_v) # get new trajectories
97  log_probs_new, rewards, values, kl_vectors, kl_decisions = new_trajectories["log_probs"],
    new_trajectories["rewards"], new_trajectories["values"], new_trajectories["kl_vectors"],
    new_trajectories["kl_decisions"]
98
99  log_probs_old, mask_array, decisions = old_trajectories_batch["log_probs"],
    old_trajectories_batch["mask_array"], old_trajectories_batch["decisions"]
100
101 advantages = compute_gae(rewards, values, params["GAMMA"], params["LAMBDA_"]) # compute
    advantages
102 loss_ppo, sem_ppo = compute_ppo_loss(advantages, log_probs_old, log_probs_new, mask_array,
     params["EPSILON"]) # compute ppo loss
103
104 kl_vectors_masked = kl_vectors * mask_array # mask out refactory periods from KL losses
105 loss_kl_vectors = jnp.sum(kl_vectors_masked) / jnp.sum(mask_array)
106 sem_kl_vectors = jnp.std(kl_vectors_masked[kl_vectors_masked != 0]) / jnp.sum(mask_array)
    ** 0.5
107 kl_decisions_masked = kl_decisions * mask_array
108 loss_kl_decisions = jnp.sum(kl_decisions_masked) / jnp.sum(mask_array)
109 sem_kl_decisions = jnp.std(kl_decisions_masked[kl_decisions_masked != 0]) / jnp.sum(
    mask_array) ** 0.5
110
111 loss_actor = - loss_ppo - params["LAMBDA_KL_VECTORS"] * loss_kl_vectors - params["
    LAMBDA_KL_DECISIONS"] * loss_kl_decisions # losses negated as optimiser minimises
112 sem_actor = jnp.sqrt((params["LAMBDA_KL_VECTORS"] * sem_kl_vectors) ** 2 +
113                      (params["LAMBDA_KL_DECISIONS"] * sem_kl_decisions) ** 2 +
114                      sem_ppo ** 2)
115
116 critic_array_masked = (advantages * mask_array) ** 2 # mask out refactory periods from
    critic losses
117 loss_critic = jnp.sum(critic_array_masked) / jnp.sum(mask_array)
118 sem_critic = jnp.std(critic_array_masked[critic_array_masked != 0]) / jnp.sum(mask_array)
    ** 0.5
```

```
119
120     loss_tot = loss_actor + params["LAMBDA_CRITIC"] * loss_critic # total loss
121     sem_tot = (sem_actor ** 2 + (params["LAMBDA_CRITIC"] * sem_critic) ** 2) ** 0.5
122
123     tot_reward = jnp.mean(jnp.sum(rewards, axis=1)) # mean total reward
124     sem_reward = jnp.std(jnp.sum(rewards, axis=1)) / rewards.shape[0] ** 0.5
125     plan_rate_masked = decisions * mask_array # mean plan rate via multiplication of decision
        and mask binary arrays
126     mean_plan_rate = jnp.sum(plan_rate_masked) / jnp.sum(mask_array)
127     sem_plan_rate = jnp.std(plan_rate_masked[plan_rate_masked != 0]) / jnp.sum(mask_array) **
        0.5
128
129     losses = (loss_tot, loss_actor, loss_critic, loss_kl_vectors, loss_kl_decisions,
        tot_reward, mean_plan_rate)
130     sems = (sem_tot, sem_actor, sem_critic, sem_kl_vectors, sem_kl_decisions, sem_reward,
        sem_plan_rate)
131
132     return (loss_actor, loss_critic), (losses, sems)
133
134 @partial(jax.jit, static_argnums=(1,)) # params contains non-array values so is marked static
135 def get_old_trajectory(scan_params_i, params, old_weights_p, weights_v):
136     """ obtain single trajectory under old policy.
137
138     Args:   scan_params_i: i'th scan parameters, pytree of arrays [T,].
139             params: training hyperparameters and environment parameters, pytree of scalars.
140             old_weights_p: old policy network weights, pytree of arrays.
141             weights_v: (static) pre-trained visual model network weights, pytree of arrays.
142
143     Returns:    old_trajectory: single old trajectory, pytree of arrays [T,].
144                 old_debug: additional debugging information, pytree of arrays [T,].
145     """
146     scan_args_0 = get_initial_scan_args(scan_params_i) # initialise tuple of scanning
        arguments (initial observation/reward, etc.)
147     _, (old_trajectory, old_debug) = jax.lax.scan(old_trajectory_step, (scan_args_0, params,
        old_weights_p, weights_v), None, params["TRIAL_LENGTH"])
148
149     return old_trajectory, old_debug
150
151 get_old_trajectories_vmap = jax.vmap(get_old_trajectory, in_axes=(0, None, None, None),
        out_axes=0) # scan_params input is a pytree of arrays [N,T]
152
153 @partial(jax.jit, static_argnums=(2,))
154 def get_new_trajectory(old_trajectory_j, scan_params_j, params, new_weights_p, weights_v):
155     """ obtain single trajectory under new policy using equivalent old trajectory.
156
157     Args:   old_trajectory_j: j'th old trajectory, pytree of arrays [T,].
158             scan_params_j: j'th scan parameters from old trajectory, pytree of arrays [T,].
159             params: training hyperparameters and environment parameters, pytree of scalars.
160             new_weights_p: new policy network weights, pytree of arrays.
161             weights_v: (static) pre-trained visual model network weights, pytree of arrays.
162
163     Returns:    new_trajectory: single new trajectory, pytree of arrays [T,].
164                 new_debug: additional debugging information, pytree of arrays [T,].
165     """
166     scan_args_0 = get_initial_scan_args(scan_params_j) # initialise tuple of scanning
        arguments (initial observation/reward, etc.)
167     _, (new_trajectory, new_debug) = jax.lax.scan(new_trajectory_step, (scan_args_0, params,
        new_weights_p, weights_v), old_trajectory_j, params["TRIAL_LENGTH"])
168
```

```
169     return new_trajectory, new_debug
170 
171 get_new_trajectories_vmap = jax.vmap(get_new_trajectory, in_axes=(0, 0, None, None, None),
        out_axes=0) # old_trajectories_batch and scan_params_batch inputs are pytrees of arrays [B
        ,T]
172 
173 ### main routine ###
174 
175 params = {
176     # initialise simulation parameters including training hyperparameters (number of epochs,
        learning rates etc.) and environment parameters (aperture size, activation function scale
        etc.)
177 }
178 
179 init_weights_p = {
180     # initialise policy GRU network weights with normal glorot initialisation
181 }
182 
183 _,(*_,weights_v) = load_('/path/to/visual_model_weights.pkl') # load pre-trained visual model
184 
185 faulthandler.enable() # detailed traceback for debugging memory errors
186 start_time = datetime.now()
187 loss_arrays, sem_arrays, actor_opt_state, critic_opt_state, final_weights_p = full_loop(params
        , init_weights_p ,weights_v)
188 print(f"Sim time: {datetime.now() - start_time}
189 s/epoch= {((datetime.now() - start_time) / params['TOT_EPOCHS']).total_seconds()}")
190 
191 save_checkpoint((actor_opt_state, critic_opt_state, final_weights_p), '
        tracking_task_checkpoint_') # checkpoint training; saving optimiser state is key
192 save_outputs((loss_arrays, sem_arrays), 'tracking_task_outputs_') # save other outputs to
        separate file
```

Example Code D.2: **Training loop and other selected functions from `tracking_task_training.py`.** The training loop consists of an outer loop over epochs to generate a full batch of old trajectories, and an inner loop over mini-batch updates, where smaller batches of old trajectories are sequentially run under a new policy and the two are compared to produce an optimisation update (Section 2.3.4, algorithm 2). Trajectories are once again obtained via `lax.scan()`, however now with complex control flow in the stepping function via use of nested `lax.cond()` statements (not shown) to govern the transitions between movements, plans and refractory periods. Note new trajectories are much faster to compute, given the actions $\mathbf{a}_t^{(k,d)}$ (vectors and decisions) made at each timestep are now fed as inputs to the scan and hence no control flow is required in the scan function. Note also use of `copy.deepcopy()` to ensure the new and old policy weights represent distinct objects in memory (i.e. do no share a reference). This script initially suffered from memory issues due to the use of nested `lax.cond()` statements within each scanned trajectory (not shown), as detailed in the main text. Manually clearing the compile cache periodically (line 80) was found to prevent these issues.