Computational Modelling of Childhood Trauma (Complex PTSD)

Jeffrey Scholes

Master of Science

Data Science

School of Informatics

University of Edinburgh

2023

Abstract

Post-traumatic stress disorder (PTSD) is a psychiatric disorder that can develop after experiencing traumatic, usually life-threatening events. The disorder is characterised through various symptom clusters, each of which can cause extreme distress and result in significant impairment within many aspects of an individuals life. Complex PTSD (cPTSD) is a recently proposed distinction of the disorder, resulting from prolonged, ongoing trauma that can cause additional symptoms, and typically precipitates lasting damage to an individuals perception of themselves and the world. Current knowledge of the underlying mechanisms of this disorder remains limited, as does knowledge of clear distinctions between PTSD types. Through methods of computational psychiatry, we model the fear learning aspect of the disorder, hoping to improve this understanding. Several reinforcement learning models of PTSD are summarised. A recent TD-Momentum model is then re-implemented and evaluated in terms of describing cPTSD and implications to treatments. Several extensions are then proposed that can add value to the model, these include concepts of associability, outcome-sensitivity, and valence partitioning which are evaluated in terms of their implications.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Jeffrey Scholes*)

Acknowledgements

First of all, I would like to extend immense thanks to my supervisor, Dr. Peggy Seriès, for the continued support, valuable feedback, and for introducing me to computational psychiatry. I am extremely grateful to have had the opportunity to make my own contribution to this very meaningful field under your excellent guidance.

I am also very thankful to Filippo Ferrari for the extended support and expert insight throughout this project.

I would like to take this opportunity to thank Dr. Iain Murray, whose kind words at a time when I was struggling this year meant a great deal to me.

Finally, I am extremely grateful for the love and support of all my family and friends, particularly my mum and dad.

Table of Contents

Chapter 1

Introduction

Post-traumatic stress disorder (PTSD) is a psychiatric disorder which effects all ages and can develop after an individual experiences or witnesses a traumatic event. Although not all who suffer from PTSD have experienced a dangerous life event, the risk of developing PTSD drastically increases after exposure to serious trauma. The context of such traumatic events can vary greatly, but will likely include feelings of horror, extreme fear or helplessness, e.g. being injured or witnessing another person being injured or killed. The disorder is thought to effect around 6–8% of the general population, with this figure increasing significantly up to 25% if individuals have experienced severe psychological trauma (combat veterans, war refugees) [22].

The Diagnostic and Statistic Manual of Mental Disorders (DSM-5) [4] classifies PTSD as a Trauma- and Stressor-Related Disorder, rather than an anxiety disorder and organises symptoms into clusters, where the duration of disturbances must be more than one month and cause significant distress and/or impairment to an individuals social or occupational life (which cannot be attributed to effects of substance abuse or another medical condition). Symptom clusters include: **intrusion symptoms**, **avoidance symptoms**, **negative alterations in cognition/mood**, and **alterations in arousal and reactivity**. However, the DSM-5 does not separate type-1 and type-2 PTSD (also known as complex PTSD, or cPTSD), so no clinical diagnosis of cPTSD currently exists. cPTSD refers to PTSD caused by repeated, long-lasting exposure to traumatic experiences (at any age) that may not be life threatening themselves, but are still extremely traumatic and often come with an increased sense of helplessness and powerlessness (e.g. domestic abuse, sexual abuse).

In contrast to the DSM-5, the International Classification of Diseases (ICD-11) [29] does provide a distinction for PTSD types. The ICD-11 lists PTSD symptoms to be: **re-experiencing symptoms** (intrusive, distressing memories, dissociative reactions e.g. flashbacks, intense distress at exposure to cues reminiscent of the trauma, re-experiencing trauma in the here and now), **avoidance symptoms** (deliberate avoidance of memories and external reminders related to trauma, e.g. people and places), **alterations in arousal and reactivity** (emotional outbursts, persistent perceptions of heightened fear, hypervigilance, hyperarousal). Additional clinical features are noted, including dissociative symptoms, suicidal ideation, substance abuse, social withdrawal, and anxiety. Emotional responses include sadness, shame, humiliation, and guilt (also survivor guilt). The criteria for cPTSD includes all of the above, and additionally: **severe problems with affect regulation** (increased emotional reactivity, self-destructive behaviour, emotional numbing, dissociation), **persistent diminished feelings of oneself** (worthlessness, guilt from not having escaped the trauma/preventing it for others), **persistent difficulties in maintaining relationships** (avoiding relationships/social situations, a sustained view of the world being dangerous/that people cannot be trusted).

There is much debate as to whether these PTSD types should be distinctly separated. Some, like Cloitre et al. [8], believe that a clear distinction is necessary for progress in refining treatments and diagnoses. However, there are those, such as Achterhof et al. [2], who claim that statistical tests show that patient groups are not well separated, showing a lack of clear evidence to support such a distinction. Here, we take the stance of Cloitre at al., that a distinction between PTSD and cPTSD will benefit more tailored and effective treatment planning. Due to this distinction being a recent inclusion in the ICD-11, there is much ambiguity in the literature with regards to which PTSD type is being discussed, making it difficult to clearly identify if the literature is more relevant to cases of prolonged, ongoing trauma or sudden, more classical PTSD trauma.

Ultimately, this dissertation aims to provide insight on cPTSD processes and symptoms, and to provide avenues for further research that could help in progressing treatment. Our focus being on why treatments may not always be effective (treatment failure/dropout), and how cPTSD mechanisms may differ to more classical PTSD. We begin by reviewing typical approaches of computational PTSD models, we then review the literature of relevant models, discussing what they can explain and their shortcomings. We then re-implement the main models of Kaye et al. [16], reproducing results and noting the extent to which they can explain mechanisms of cPTSD. We also perform our own additional investigations on how model behaviour reacts to early life stress (ELS). Finally, we provide details on potential model extensions that can add value to explaining mechanisms of the disorder.

Chapter 2

Background

Many key questions remain in order for PTSD to be better understood. The underlying mechanisms of the disorder need to be clarified for treatments of each PTSD type to be improved, and for clinicians to be able to accurately predict which individuals are more susceptible to developing the disorder after experiencing trauma. Some important research questions that remain include [5]:

- Why (at a neurobiological level) do some individuals fully recover from traumatic experiences and others do not?
- Can we predict how different people will react to trauma and/or treatment?
- How does severity and duration of trauma affect likelihood and severity of PTSD?

There have been various different approaches taken in modelling PTSD mechanisms and framing these important questions. Much of the literature takes on one or more of these approaches: predictive coding, Bayesian inference, and reinforcement learning.

The predictive coding framework views the brain as a predicting machine that minimises prediction errors to correctly update predictions based on sensory input. PTSD is proposed to cause a breakdown in this framework that is reflected in symptom clusters. Aitchison and Lengyel note that while predictive coding offers a framework for implementing prediction error based learning updates, Bayesian inference provides a possible calculus for computing such predictions [3]. In Bayesian inference, prior beliefs about the environment/sensory input are tested and updated when new evidence is received, where the posterior updates follow Bayes theorem. The prior belief is combined with the likelihood of observed data given the prior belief, and is normalised by the overall probability of observing the data, giving a valid posterior. This process repeats using the latest posterior as the new prior, generating an endless cycle of belief updates about the environment. Intense trauma can change an individuals internal model such that powerful, maladaptive priors reflect PTSD symptom clusters.

Reinforcement learning (RL) models generate agents that learn a set task in real time through trial and error, along with delayed rewards. They are typically applied to model learning or decision making, e.g. avoidance or fear learning and are well suited to be fit to behavioural tasks, with their adaptability allowing us to identify differences in the neural functions of learning processes in individuals with PTSD. These models are typically combined with neuroimaging data from the areas of the brain associated with learning and decision-making (e.g. the amygdala), allowing for a deeper understanding of the mechanistic processes of learning related to the disorder [7]. However, current RL approaches are usually hindered by basic assumptions of outcomes in the related behavioural tasks, which may not account for the broad spectrum of outcomes in reality.

Predictive coding approaches to PTSD are typically theoretical and not applied to data due to the complexity of retrieving such information (e.g. how do we extract accurate information about ones internal belief system?). However, they do provide a thorough base for theory on the underlying mechanisms of PTSD and can be applied to explain a variety of symptom clusters [30]. The following literature review focuses on RL models of PTSD that relate more directly, and can be compared to, the main RL model explored in this dissertation.

2.1 Literature Review - Reinforcement Learning Models

Radell et al. suggest that a fully comprehensive model of PTSD should provide novel predictions and potential explanations for the underlying mechanisms of *all* symptom clusters of the disorder [19]. However, the complexity of the disorder, along with the current lack of understanding the underlying mechanisms, make this unifying model difficult to define. Most computational models of PTSD instead focus on adaptations within one of the symptom clusters: fear learning, hyperarousal, avoidance, cognition & mood, and intrusive memories. With RL models commonly applied to behavioural data, symptom clusters where it is easier to extract such data are favoured, namely: fear learning, hyperarousal, and avoidance. Models related to disorders sharing symptomatology with PTSD are included from which we can draw parallels to PTSD mechanisms. The lack of distinction between PTSD types within the literature makes it difficult to specify which models relate to cPTSD. Efforts have been made to include relevant literature with the potential to describe mechanisms associated with cPTSD.

2.1.1 Kaye et al. (2023) - TD-Momentum Threat Prediction Model

Kaye et al. [16] propose a fear learning model of estimating threat which incorporates associative and non-associative responses to threat across different contexts. They implement a Temporal Difference learning model (TD) with the addition of a *momentum* term, m_t. Threat in context c at time t is computed as follows.

$$T_{c,t} = T_{c,t-1} + \alpha(u_t - \gamma_1 T_{c,t-1}) + f m_t \tag{2.1}$$

$$m_t = m_{t-1} + \gamma_2 \sum_{c=\{A,B,\ldots\}} \alpha(u_t - T_{c,t-1}) \tag{2.2}$$

γ_1 is the decay rate of associative prediction errors, α is the learning rate. The momentum at time t, m_t, is computed from the sum of all decayed prediction errors across all contexts. Scaling constant f and momentum decay rate γ_2 control how much momentum influences learning.

Momentum was first proposed by Eldar et al. [10] as a way of representing mood in Bipolar Disorder (section 2.1.5). Eldar et al. proposed that mood corresponds to the overall momentum of recent outcomes, where its biasing influence on the perception of outcomes accounts for environmental dependencies, therefore “correcting” learning. Applied here by Kaye et al., momentum (Eq. 2.2) allows for threat prediction errors in any given context to affect predictions in all contexts. In other words, this term corresponds to the “mood” of an agent, given recent experiences.

Comparing to a basic TD model and fitting to real data from a contextual rodent study (discussed in chapter 3), the authors showed how this TD-Momentum model represented fear learning better than the simpler model. Kaye et al. also note the addition of this momentum term may provide potential reasons for PTSD treatment failure (exposure therapy). However, the model is only evaluated generally by the authors. In chapter 3 we re-implement the models and evaluate them with a focus on how early life stress can effect threat perception. We also explore any links to cPTSD.

2.1.2 Homan et al. (2019) & Brown et al. (2018) - Associability Models of Fear Learning and Hyperarousal

Homan et al. create a fear learning hyrbid model of Rescorla-Wagner/Pearce-Hall models [14] used to explain conditioned threat responses for two groups of combat veterans (PTSD and healthy controls). The task for both groups was to learn pairings of pictures of faces and electric shocks. The authors identified a correlation between the magnitude of prediction errors and the severity of symptoms in PTSD individuals, where highly symptomatic individuals were more influenced by larger prediction errors.

The associability term gives a value to the attention that each specific cue receives, depending on its historic accuracy of predicting the outcome. Associability effectively turns the learning rate from a constant into a dynamic parameter. Unreliable cues receive larger associability as time moves on, as they are more likely to be unreliable in the future, and are thus updated preferentially as new information comes in.

Brown et al. [7] propose a similar hybrid Rescorla-Wagner/Pearce-Hall model, focusing on hyperarousal symptoms and investigating potential reasons for disproportionate reactions to unexpected stimuli. The authors fit results from a loss learning task performed by a group of combat veterans who had to choose between two stimuli with monetary income and learn the "better" option over time. Brown et al. showed that veterans with PTSD have an increased learning response to surprising events, i.e. an increased learning rate for unexpected events that inherently cause larger prediction errors. PTSD individuals typically received increased associability weights for unexpected cues during loss learning, meaning that loss learning in these individuals was more heavily influenced by unexpected outcomes related to these cues. Similar to Homan et al., associability represents the attention that each cue is given by the participant. Thus, loss learning was more heavily influenced by attention given to surprising outcomes, with PTSD individuals more likely to allocate additional attention to unexpected outcomes, which could provide reason for exaggerated responses to unexpected stimuli (hyperarousal). The expected value of a stimulus A, Q^A, is updated as follows,

$$Q^A_{t+1} = Q^A_t + \alpha \cdot \kappa^A_t \cdot \delta_t \tag{2.3}$$

α represents the (fixed) learning rate, δ_t represents the prediction error at time t. The associability of stimulus A is represented by κ^A_t, updated as follows.

$$\kappa^A_{t+1} = (1-\eta)\kappa^A_t + \eta|\delta_t| \tag{2.4}$$

The associability weight parameter, $0 \leq \eta \leq 1$, controls how much historic prediction errors influence the future associability of each cue.

The model was found to predict participant choices better than models without associability, suggesting that attention-based learning may be a key aspect in the underlying mechanisms of PTSD and may assist in refining treatment targeting. Similar to the momentum term used by Kaye et al., associability allows for an agent to take advantage of prediction errors gathered across its lifetime. In other words, both associability and momentum terms add value to their respective models by allowing agents to utilise past experiences when generating future predictions.

Comparing these terms, momentum (Eq. 2.2) sums the decayed historic prediction errors of threat across all contexts to influence prediction, whilst maintaining a constant learning rate. Associability (Eq. 2.4) creates a more dynamic learning rate (seen in Eq. 2.3), where learning for more unreliable cues is updated preferentially. These terms provide us with an opportunity to explore the extent to which past experiences can influence future fear learning processes. In relation to cPTSD and childhood trauma specifically, investigating how these terms react to prolonged stressors in early life stages and how these influence the extinguishing of learned fear responses could prove to be key in developing the understanding of cPTSD. We propose a model combining associability with the momentum model of Kaye et al. in Chapter 4.

2.1.3 Ross et al. (2018) - General PTSD Learning Deficits

Ross et al. [23] explored how much PTSD is associated with general reinforcement learning, outside of the context of learned fear stimuli. The authors investigated variations in prediction errors between a group of PTSD individuals and a group of control individuals. Participants were shown two images of houses and were required to identify which was unlocked in order to maximise monetary reward. Results were fit to several adaptions of the classical Rescorla-Wagner model (Eq. (2.5)).

$$V_{t+1} = V_t + \delta * \alpha \tag{2.5}$$

The expected value of a choice is represented by V_t, the prediction error $\delta = (outcome_t - V_t)$, and α represents the learning rate ranging from $0-1$. The authors found that a risk-sensitive, anti-correlated model fitted participant behaviour best. “risk-sensitive” implies separate learning rates are used for positive and negative prediction errors. “anti-correlated” implies that updates to the expected value of the unchosen stimulus are in the opposite direction of the prediction error. Choice estimate values are then updated via Eq. (2.6) and Eq. (2.7), where $\alpha^{+/-} = \alpha^{+}$ if $\delta \geq 0$ or α^{-} otherwise.

$$V_{t+1}^{Chosen} = V_t^{Chosen} + \alpha_{Chosen}^{+/-}\delta \tag{2.6}$$

$$V_{t+1}^{Unchosen} = V_t^{Unchosen} - \alpha_{Unchosen}^{+/-}\delta \tag{2.7}$$

Although no significant differences in parameter values across these adult groups were found, previous cognitive flexibility investigations using this model have been performed on adolescents that found a significantly increased learning rate for negative prediction errors compared to adults [13]. Therefore, we propose that this model may have further implications (and provide more distinct results between groups) when applied to childhood trauma and cPTSD.

Comparing to previous models, this model is similar to the associability model of Brown et al., where associability can be portrayed as *weighting* learning such that unreliable cues are updated preferentially and influence learning more than others. Ross et al. do not include this weighted learning as such, although there are separate learning rates for positive and negative prediction errors, no specific cues are updated preferentially. The momentum model is also the only approach to explicitly incorporate non-associative learning (via the momentum term).

2.1.4 Yanamori et al. (2023) - Anxiety Related Approach-Avoidance

Yanamori et al. [31] investigated anxiety-related approach-avoidance, proposing that anxiety may increase avoidance responses (decrease approach responses) in an approach-avoidance task, which may be explained by increased sensitivity to punishments rather than rewards. In other words, anxious people are more likely to avoid pursuing reward than to pursue a reward associated with potential punishment. Although based on anxiety-related avoidance, the concepts used have parallels to PTSD-related avoidance.

Yamamori et al. conducted a "Restless Bandit" behavioural task where participants aimed to maximise rewards while avoiding punishments. Participants chose one of two images that resulted in either a reward or no reward. Options were grouped as "safe" and "conflict" (unknown to participants), where "conflict" options sometimes resulted in screaming sounds (punishment) and could be any combination of reward/no reward and punishment/no punishment (4 possible outcomes). "safe" options never resulted in punishment and were, on average, less likely to produce reward over "conflict" options. "conflict" options thus represented more reward but also the potential of punishment.

Participant choices were fit to variations of the Q-learning algorithm [28]. The model with specific learning rates and specific outcome sensitivity parameters for rewards and punishments fit behavioural data best. Probability estimates of outcomes are progressively updated as action outcomes are observed. These estimates are used by participants to choose an option that maximises subjective value, e.g. option most likely to result in reward, or option most likely to result in no punishment. The probability estimates of observing an outcome, $o = \{reward, punishment\}$, of a chosen option,

$a = \{conflict, safe\}$, were updated via the below:

$$Q^o_{t+1}(a) = Q^o_t(a) + \alpha \cdot [o^t - Q^o_t(a)] \tag{2.8}$$

Learning rate α is split into reward- and punishment-specific learning rates, α^r and α^p that imply a similar risk-sensitive approach to Ross et al., however, learning rates are chosen based on outcome, rather than sign of prediction error. Probability estimates are then merged into action weights, W, at every trial, with outcome sensitivity parameter, β, separated into reward- and punishment-specific parameters, β^r and β^p.

$$W = \beta^r \cdot Q^r - \beta^p \cdot Q^p \tag{2.9}$$

The action weights then form the basis of which choice is made between options, modelled with a softmax function,

$$P(a) = \frac{W(a)}{\sum_i W(i)} \tag{2.10}$$

The sensitivity parameters β^r, β^p capture the extent to which each outcome impacts choice and allow for asymmetries in how individuals value reward and punishment to be captured. The "reward-punishment sensitivity index", β^r/β^p, provides a unique index for each individual as to where they lie on the approach vs avoidance spectrum, higher values correspond to approach preference, and lower values to avoidance preference.

The authors found that punishment learning rate α^p and reward-punishment sensitivity index β^r/β^p were negatively correlated with task-induced anxiety. This indicates that anxious individuals were slower in updating estimates of punishment probability, and that they placed more weight on punishment relative to reward when choosing between options (reward was biased by potential for punishment). This explains the effect of task-induced anxiety on avoidance behaviours, and may be useful in providing further insight into mechanisms that drive avoidance. As anxiety disorders are often comorbid with PTSD [6], similar experiments could be conducted on PTSD individuals exploring if parameter values for PTSD individuals are similar to those in this anxiety-related study. Differences in parameter values may provide insight on differences between the mechanisms of these disorders and provide potential explanation for comorbidity.

2.1.5 Eldar et al. (2016) - Momentum Model of Mood in Bipolar Disorder

Here we detail the momentum model by Eldar et al. [10], which is implemented by Kaye et al. in the main model of this dissertation. This model aims to describe mood in a computational model of bipolar disorder (BD). It is known that BD can stem from prolonged childhood trauma (cPTSD) and individuals often suffer from both disorders [6]. Also, cPTSD is often misdiagnosed as BD due to similar symptoms, as such we can draw parallels to cPTSD mechanisms.

Eldar et al. propose that experiences affect mood, which in turn influences future experiences. They suggest that mood is the overall momentum of recently experienced outcomes, and that its biased influence on outcome perception allows for the "correction" of learning, accounting for environmental dependencies. The model is based on an adjusted form of the Rescorla-Wagner model (Eq. (2.5)), this standard form assumes different states are independent of each other. However, Eldar et al. suggest that if different states are *not* independent of each other, and multiple states can be similarly affected by some environmental factor, then an adjustment is required that updates expectations of all states affected by this factor, when a prediction error is experienced in any one of the states. The *momentum* term is introduced to represent this:

$$m_{t+1} = m_t + \alpha((outcome_t - V_t^s) - m_t) \tag{2.11}$$

Momentum assumes that states close in space and time are affected similarly. The term is combined with Eq. (2.5) to create the momentum model for mood (Eq. 2.12), where f_t is a scaling factor. This model allows for expectation of reward in a given state to reflect outcomes experienced in all states.

$$v_{t+1}^s = v_t^s + \alpha(f_t * m_t + (outcome_t - v_t^s)) \tag{2.12}$$

The authors apply behavioural data (gathered by Eldar and Niv [9]) from BD individuals who played a different slot machine before and after a wheel-of-fortune draw that they either won or lost. Participants who reported high emotional instability were asked to report how their mood varied throughout the experiment. The wheel-of-fortune outcome was shown to influence mood in these individuals. Those who won the wheel-of-fortune preferred the second slot machine (after the win, played in a better mood), while those who lost the wheel-of-fortune preferred the first slot machine (before the loss, played while in a better mood), showing how outcomes that may not be directly related to future stimuli can influence interpretation of future stimuli (i.e. mood influences learning).

Kaye et al. implement this concept of mood into a model investigating the fear learning aspect of PTSD, whereby they propose that trauma experienced in any context should hold some influence over threat predictions in other contexts that are close in space and time. We now move on to the evaluation of this main paper.

Chapter 3

Re-implementation & Evaluation of Kaye et al. (2023)

The majority of previous RL models of PTSD focus on associative learning, describing the disorder as a deficiency in extinction learning of conditioned fear responses thought to rely on RL mechanisms, i.e. future predictions are influenced by errors between predictions and actual outcomes (prediction errors). Many studies build upon the classical Rescorla-Wagner model [21] which accurately describes associative learning. However, Kaye et al. note that these studies do not incorporate non-associative learning, and thus fail to account for how repeated trauma (across different contexts) can influence future threat predictions. There is currently a lack of knowledge surrounding how non-associative learning in PTSD can be described computationally, as well as how this process may be implemented neurobiologically.

The authors propose a novel RL model to advance this knowledge, combining the basic Temporal Difference (TD) RL model and the mood model by Eldar et al. [10] (section 2.1.5). Kaye et al. suggest that the estimation of *frequency* of trauma may be influential to PTSD adaptations in the brain, instead of simply threat associations with specific cues. The authors first propose a Bayesian model for estimating frequency of threat, to understand how well an agent can perform based on its own experiences. Two RL models are then created; the simpler TD model, and the TD-Momentum model (our main focus). These RL models are fit to data from a stress-enhanced fear-learning (SEFL, section 3.2) model of rodents to assess if the TD-Momentum model provides an improved fit to threat learning.

3.1 Bayesian Baseline Model

Kaye et al. propose that an agent in an environment must estimate future likelihood of trauma based upon previous experiences, which can be defined in a Bayesian process. This Bayesian model is used as a baseline for threat estimation from the perspective of an "ideal observer" that utilises all information of a stimulus (in the given context) to estimate threat as best as possible. This allows us to measure how much information an agent can accumulate over a lifetime when subjected to traumatic events.

A simple probabilistic model was used to test the performance of an ideal Bayesian observer in an information-poor environment, i.e. where agents have no previous knowledge of the environment and must learn from scratch. The model runs for 700 timesteps where, at each step, attacks occur with probability p_a and death occurs with probability p_d (given an attack occurs). This procedure is repeated for all timesteps, or until an attack results in death. If an agent survives a timestep, it continues to learn and this procedure repeats at the next timestep (model structure is shown in Appendix A, Fig. A.1). Agents have no power over the environment and do not know the fixed probabilities of attack (p_a) or death (p_d), these are set by researchers. During simulations, if the actual probability of death is set too high, agents will not gain enough experience over their lifetime in order to produce good estimates. Therefore, agents must maximise the information available in order to produce accurate estimates.

3.1.1 Bayesian Attack Rate Estimation Model

In the attack model for Bayesian estimation, attacks are binomially distributed over the agent lifetimes (700 timesteps), with a probability of attack $p_a = 0.2$ and a probability of death given attack $p_d = 0.01$. Both p_a and p_d agent estimates are calculated via Bayes theorem, based on a sequence of attacks, $\mathbf{x_t}$.

$$p(p_a, p_d|\mathbf{x_t}) = \frac{p(\mathbf{x_t}|p_a, p_d)p(p_a, p_d)}{\int p(\mathbf{x_t}, p_a, p_d)d\mathbf{x_t}} \tag{3.1}$$

Upon new observations, posterior estimates are updated through an affine invariant Markov Chain Monte Carlo (MCMC) sampler [11]. As time progresses, the previously evaluated posterior becomes the new prior and estimation is repeated. As agents have no starting knowledge, we set a flat initial prior at $t = 0$. The MCMC sampler was fit using the below likelihood function to estimate the posteriors of p_a and p_d.

$$ln(\mathcal{L}(\mathbf{x_t}, p_a, p_d)) = \sum_{attacks} ln(p_a(1-p_d)) + \sum_{non-attacks} ln(1-p_a) + \sum_{deaths} ln(p_a * p_d) \tag{3.2}$$

The affine invariant MCMC sampler, first proposed by Goodman and Weare [12], is applied due to its efficiency in exploring skewed posterior distributions. Here, agent posterior estimates are predicted to be positively skewed between 0 and 1, with more estimations being near the real value of p_a. This estimator generates the best estimate available to an ideal Bayesian observer, with the information available at each timestep

3.1.1.1 Bayesian Attack Rate Estimation Model - Implementation of Methods

In terms of implementing models, all code was written in Python (Version 3.9 [27]) using various packages. Attack sequences of 700 timesteps based on figure A.1 were simulated for use in the MCMC Sampler, code for these simulations is given in Appendix B.1. The *"EnsembleSampler"* function from the *"emcee"* [11] package was used to generate the MCMC sampler with 30 walkers, *"num_steps"*=60, *"burnin"* = 0.3 and a *"moves"* argument of *"Stretch_move"* with stretch parameter = 2. Code for the MCMC implementation and related plots can be found in Appendix B.2 - B.3.

Kernel Density Estimation (KDE) was applied to results of the MCMC sampler using the *"gaussian_kde"* function of the *"stats"* package [1]. Suitable bandwidths were selected for the p_a and p_d distributions to generate smooth posteriors. A *"gaussian_filter"* was applied so that contour plots of p_a against p_d were readable.

3.1.2 Auto-regressive Time Series

Auto-correlated attack rate time series were also generated to create attack sequences used as input to the Bayesian model, following the auto-regressive (AR) process below.

$$p_{a,t} = c p_{a,t-1} + \mathcal{N}(0, 0.01) \tag{3.3}$$

Attack rate at timestep t is denoted $p_{a,t}$, the correlation for successive timesteps is represented by c, additional noise $\mathcal{N}(0, 0.01)$ is sampled from a normal distribution with mean $\mu = 0$ and standard deviation $\sigma = 0.01$. The resulting time series of unique p_a at each timestep is used to generate attack and death sequences for input to the MCMC sampler. Whilst p_a varies at each timestep, $p_d = 0.01$ remains constant. This process generates attack sequences where $p_a \approx 0.01$, as $p_{a,0} = 0$ (initial p_a).

3.1.2.1 Auto-regressive Time Series - Implementation of Methods

AR time series simulations used the *"ArmaProcess"* function of the *"statsmodels.tsa"* package [25]. This function requires *"ar"* and *"ma"* arguments which represent coefficients for the auto-regressive and moving-average lag polynomials, respectively (zeroth lag always set to 1). To implement the AR process in Eq. (3.3), arguments are set to $ar = [1, -c]$ and $ma = [1]$, where c values must be negated in the ar argument, e.g. for $c = 0.7$ we set $ar = [1, -0.7]$. As we require a strictly non-negative probability of attack, p_a, in the binomial distribution, time series must be clipped between 0 and 1 before generating attack sequences (related code detailed in Appendix B.5).

We generated 10,000 separate lifetime (700 timesteps) attack rate time series, used to create 10,000 separate attack sequences consisting of 0's (no attack) and 1's (attack). Each sequence was used as input to the MCMC sampler, testing how the model reacts to varying AR correlation coefficients c (related code in Appendix B.6).

3.1.3 Bayesian Model - Re-implementation

Here we re-implement this Bayesian model and reproduce plots similar to those of Kaye et al.. Fig. 3.1 shows posteriors generated from MCMC sampler results of a typical sequence of attacks over an agent lifetime, created using a single AR time series of attack probability (correlation constant $c = 0.7$). This shows estimates of the probability of attack (p_a) to be very accurate, with real attack rates generated by the AR time series being $p_a \approx 0.01$, our p_a posterior estimates are clearly centred around this value, ranging between 0 - 0.02, indicating low variance. The accuracy of p_d posterior estimates is very poor in comparison, with estimates over a much wider range, showing non-convergence to a confident estimate and indicating the model is poor at estimating lethality of attacks. This is intuitive when considering the model structure; an agent cannot provide further information for posterior updates after death (Fig. A.1). This Bayesian model progressively improves its estimates over time, becoming more confident in estimating attack frequency throughout its life (seen in Appendix A Fig. A.2, variance in model estimates decreases over time, showing convergence for p_a estimates). The correlation in AR time series can influence precision of Bayesian model attack estimates, larger auto-correlation ($c = 0.999$) results in larger variance in p_a estimates (flatter, wider posteriors) and less accuracy in results. Lower auto-correlation ($c = 0.7$) results in lower variance in p_a estimates (sharper, thinner posteriors) and more accuracy in results (seen in Appendix A, Fig. A.3 showing "average" MCMC sampled densities for 10,000 simulated AR time series with various values of correlation).

Kaye et al. then briefly discuss links to childhood trauma, comparing an early life stress (ELS) scenario to one of lifetime stress. We re-create a similar plot in Fig. 3.2,

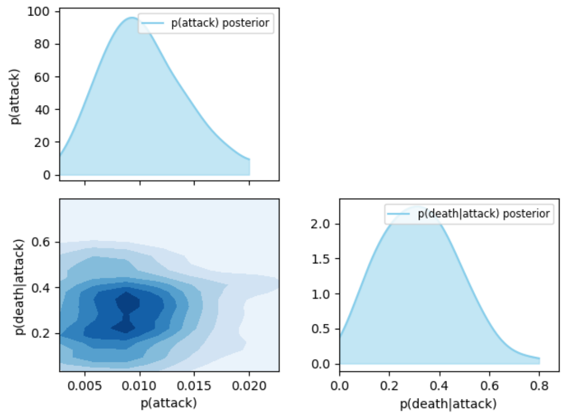

Figure 3.1: Estimated posteriors for p_a and p_d for typical attack sequence generated from AR time series of attack rates with $c = 0.7$. The Bayesian estimator approximates p_a very accurately ($p_a \approx 0.01$), however estimates lethality of attacks poorly, as agents gain no information after death.

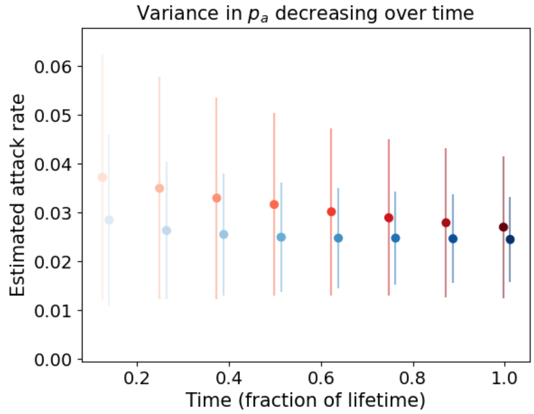

Figure 3.2: ELS scenario (red) with $n = 15$ attacks in early life and lifetime scenario (blue) of $n = 15$ attacks across life. The ELS scenario shows larger threat predictions and increased variance across its lifetime, showing the Bayesian model can naturally represent the impact of ELS.

where $n = 15$ attacks are uniformly distributed in the first half of life to represent ELS, and across the entire lifetime for the lifetime scenario. Attacks in early life can cause mean and overall variance of threat estimations to be larger (red) compared to attacks spread across a lifetime (blue), showing how the Bayesian model can naturally represent the disproportionate impact of early life trauma (see Appendix B.7 for related code).

3.1.4 Bayesian Model - Additional Investigations

We now extend upon the findings of Kaye et al. by applying our own ELS scenarios and analysing how the model reacts. We investigated the effects of ELS clusters of 3 attacks compared to random attacks over a lifetime (see Appendix B.8 for related code). For the ELS scenario, 5 equally spaced ELS clusters of 3 attacks were administered (over adjacent timesteps, totalling 15 attacks), while the lifetime scenario consisted of 5 attacks randomly over a lifetime. Clustered attacks represent more significant or prolonged traumatic ELS compared to single attacks. Fig. 3.3 shows the ELS scenario has larger mean and variance of threat estimations compared to the lifetime scenario.

However, this may be due to the ELS scenario simply having more attacks. We also compare the clustered ELS scenario to a lifetime scenario with the same number of attacks. Fig. 3.4 shows mean threat predictions in the lifetime scenario all increase slightly as expected. However, variance in lifetime threat estimation remains similar to lifetime variance in Fig. 3.3. The mean threat estimations in the ELS scenario, as well as variance, still remain significantly larger than those in the lifetime scenario.

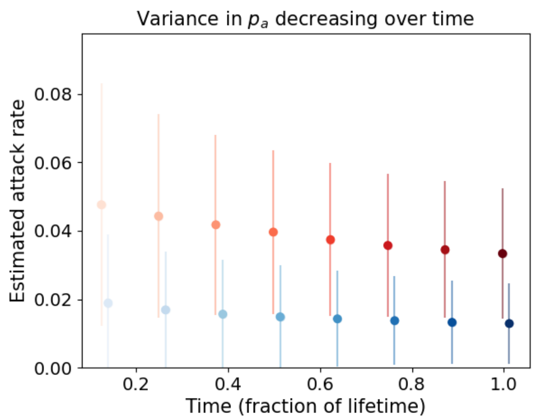

Figure 3.3: 5 equally spaced 3-clustered attacks (red-ELS), compared to 5 random attacks over lifetime (blue-lifetime)

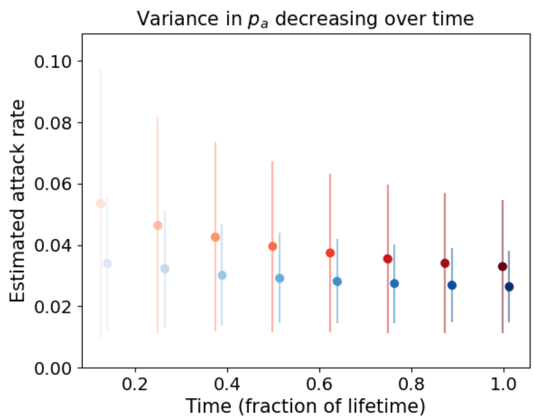

Figure 3.4: 5 equally spaced 3-clustered attacks (red-ELS), compared to 15 random attacks in lifetime (blue-lifetime)

Comparing the clustered ELS scenarios in Figs. 3.3 and 3.4 to random single ELS attacks in Fig. 3.2 (all red plots), we observe that the clustered scenario threat estimates maintain larger variance and mean in comparison to the single attack scenario. In other words, these simulations show that clustered attacks in early life can cause the Bayesian estimator to be less accurate and predict higher threat than random single attacks in early life. However, as these plots show single simulations, they may not generalise to every scenario of clustered vs single attacks. In any case, there is a clear disproportionate effect on threat estimates caused by any early life attacks, clustered or single.

3.1.4.1 Bayesian Model Implications for cPTSD

In terms of cPTSD implications, our investigations correspond to comparing types of early life trauma to trauma experienced over a lifetime. Findings suggest that individuals exposed to clusters of ELS predict higher threat across their lifetime compared to those who suffered less prolonged ELS, or less intense trauma spread across their lifetime. These increased threat predictions represent an increased perception of threat in the world throughout day-to-day life which may provide explanation for the dysfunctional physical and emotional responses prevalent in cPTSD, classified by symptom clusters.

Physical symptoms may be represented by an increased state of hyperarousal (startle response) and hypervigilance to sensory inputs. Emotional symptoms would be represented by emotional dysregulation during various situations. For example, increased emotional reactivity to small stressors, where an individual may react with intense negative emotion to a minor stressor (e.g. being easily overwhelmed by a minor inconvenience). Another possible representation may be a persistent, deep sense of mistrust of the world in general, caused by persistent elevated threat level. Symptoms may then lead to impairment in important areas of social functioning (occupational, family, educational) and may also be linked to difficulties maintaining relationships.

It could also be argued that this increased perception of threat may lead to avoidance of reminders of the trauma itself, or even to avoidance of small stressors unrelated to the original trauma that, when encountered, may trigger these aforementioned symptoms.

3.2 Stress-Enhanced Fear-Learning Model

Here we define the Stress-Enhanced Fear-Learning (SEFL) model of rodents which produces behavioural data used for fitting to the RL models of threat. The SEFL has been shown to result in long-lasting enhancement of fear behaviour in rodents [20]. Stressed mice were subject to 15 unpredictable footshocks in context A on day 1 over 90 minutes, with random inter-shock intervals of 4 to 8 minutes. Mice were transferred to context B (with completely different sensory characteristics) for 10 minutes on day 6 and administered a single footshock after 5 minutes. On day 7, mice were returned to context B for a further 10 minutes with no footshocks. A group of control mice were not exposed to any footshocks (unstressed) on day 1 in context A, but had the same experience as stressed mice on days 6 and 7. Freezing was defined to be complete cessation of movement (other than breathing), and was measured for all mice across all days. Stressed mice showed an increased average freezing percentage across days 6 and 7 compared to unstressed mice, shown in Figs. A.4 and A.5 of Appendix A.1.

“Freezing” is a voluntary defence mechanism in rodents where the animal is completely immobile in response to potential threat. The raw freezing time series created for each mouse is binary, where 1 marks observed freezing behaviour and 0 marks no freezing. This is transformed into “smoothed freezing”, used as input for the models. Smoothing is a pre-processing technique to assist in time series analysis of freezing in which the percentage of freezing within consecutive time intervals is calculated, giving a smoother view of freezing variation over time. Kaye et al. use a 15s sliding window smoothing, i.e. average freezing is calculated over 15s fixed intervals, the window is then moved on 1s and freezing percentage is calculated again, creating a time series of moving average values between 0 and 1 (an example of this smoothed freezing is shown in Fig. 3.7). We perform a similar smoothing to the raw freezing data, generating our own smoothed freezing sequences (related code can be found in Appendix C.2).

3.3 Reinforcement Learning (RL) Models

We now introduce the RL models applied by Kaye et al. that aim to explain the neural processes of fear learning in PTSD. The free parameters in each model allow for fitting to the stressed and control (unstressed) mice groups from the SEFL, allowing for hypotheses and conclusions to be drawn from comparisons across groups. We begin by re-implementing the models and reproducing similar plots before performing additional investigations on TD-Momentum model behaviour and response to ELS.

Fitting the TD and TD-Momentum models to the contextual SEFL experiment data, Kaye et al. tested if threat momentum was a source of PTSD symptoms, rather than just the specific association with traumatic events. Parameters for both models were fit via maximum likelihood, minimising the negative log likelihood (NLL) function of each model. The maximum likelihood fit for each model (of each mouse) was compared using BIC scores to identify the best fitting model for each mouse. Inputs to models were binary time series of attack sequences (1s indicating attacks, 0s no attack), threat was fit for both contexts across all three days, where the threat variable in models (output) was re-scaled to (0.1, 0.9) to match the real smoothed freezing probabilities.

3.3.1 Parameter Recovery

Parameter recovery was performed for both models prior to fitting. Generative model functions were built and two scenarios for parameter sampling were carried out: one set from multivariate normal distributions (with no covariance), and one set from random choices of potential parameter value in lists provided by Kaye et al..

The multivariate normal distribution had means equal to the midpoints of each respective parameter value list, with suitable variances to ensure sampled values were spread across the complete range of values for each parameter. Parameter values sampled from the lists were simply random choices from each list. 100 simulations were run for each sampling scenario and parameters were recovered via maximum likelihood fit, minimising the NLL function of each model using the *"differential_evolution"* function with a *"polish = True'* parameter from the *"scipy.optimize"* package [15]. The inclusion of *"polish"* means the *"L-BFGS-B"* method is applied to the best population member, slightly improving results. Differential evolution was found to be the best optimisation method for both models. Related code can be found in Appendices C.4-C.5 (TD model), and Appendix D.4-D.5 (TD-Momentum model).

Recovery was very successful for the basic TD model, with Pearson correlation coefficients between sampled and recovered parameter values for both α and γ_1 parameters being close to $r_\alpha \approx r_{\gamma_1} \approx 1$ (perfect recovery), showing the TD model function and related TD NLL function were implemented correctly (Figs. A.7, A.6 in Appendix A). Recovery for the TD-Momentum model was successful for α with correlation consistently returning values $r_\alpha \approx 1$. Recovery for γ_1 was not as successful, with correlation in the range $0.57 < r_{\gamma_1} < 0.87$. Recovery for γ_2 and f parameters was less successful with correlations coefficients in the ranges $0.38 < r_{\gamma_2} < 0.69$ and $0.2 < r_f < 0.61$ (Figs. A.8, A.9 in Appendix A), this weaker recovery may be due to the parameters having a similar effect on threat estimations, hindering the minimiser from accurately minimising the NLL, thus returning non-minimal values. No apparent correlation between fitted parameters was found, suggesting this poor recovery is not due to the parameters trading variance with each other. Kaye et al. do not note any parameter recovery in their study, it may be the case that recovery was not performed. In any case, we do not have any recovery figures or thresholds to compare our findings to. Overall, our findings show that both models and related NLL functions are correctly implemented.

3.3.2 TD Model

The Temporal Difference (TD) model, adjusted from Sutton and Barto [26], follows the update rule:

$$T_{c,t} = T_{c,t-1} + \alpha(u_t - \gamma_1 T_{c,t-1}) \tag{3.4}$$

Threat at time t in context c is denoted $T_{c,t}$, learning rate is represented by α and controls how much the prediction error $(u_t - \gamma_1 T_{c,t-1})$ updates future threat estimations, decay rate is represented by γ_1 and controls how much we discount the value of future rewards. Threat is learned from a sequence of unconditioned stimuli, u_t, as input, in this case a sequence of attacks (0 for no shock, 1 for shock). Parameter values are bounded as follows: $0.05 \leq \alpha \leq 0.9$ and $0.9 \leq \gamma_1 \leq 0.99999$. This update rule describes *associative* threat learning for PTSD.

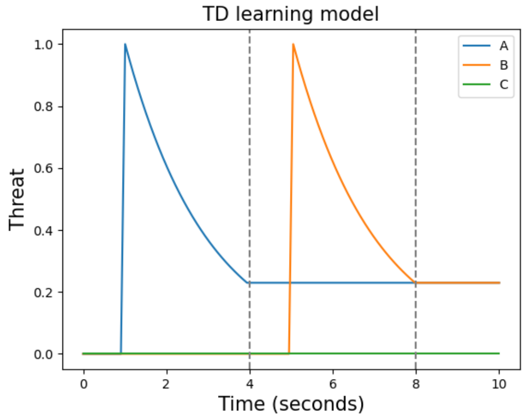

Figure 3.5: Typical TD threat estimation, single attack in context A and B, no attacks in C. Threat prediction for context C does not increase from 0. Dashed lines indicate moves from context A to B to C.

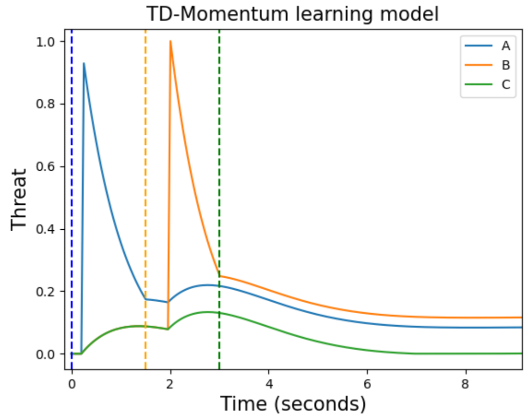

Figure 3.6: Typical TD-Momentum threat estimation for similar scenario. All threat estimations are affected when an attack occurs in any context. All threat in context C is due to attacks in other contexts.

3.3.3 TD Model - Re-implementation

The TD model allows us to model threat estimations for agents within the context in which attacks occur. This model assumes that threat prediction in each context is independent of other contexts, as such, there is no means for threat estimations to "carry across" contexts. Fig. 3.5 shows a simulation of a single attack in contexts A and B, with no attacks in C. Starting in context A, dashed lines represent transitions between contexts (A to B and B to C). There is no increase in contexts B and C threat when an attack occurs in A (the sharp increase in threat), similarly for A and C when an attack occurs in B. Predictions in any context remain constant upon leaving that context.

The TD model thus captures the *associative* fear learning of the disorder, allowing agents to associate traumatic experiences with the context in which they occurred. However, the simple TD model neglects the fact that an agent may have had previous experiences in other contexts that may influence future threat predictions across novel contexts. Kaye et al. argue that these past experiences influence an agents current view of the world (or "mood" as referred to by Eldar et al. [10]), which needs to be incorporated for threat estimation to represent both associative and non-associative learning. They present the TD-Momentum model to express this.

3.3.4 TD-Momentum Model

The TD-Momentum model is an extension of the TD model with the addition of a "momentum" term, introduced by Eldar et al. [10] (section 2.1.5). The inclusion of momentum allows for prediction errors in any context to influence estimations in novel contexts, incorporating *non-associative* fear learning. The update rule is defined as:

$$T_{c,t} = T_{c,t-1} + \alpha(u_t - \gamma_1 T_{c,t-1}) + f m_t \tag{3.5}$$

$$m_t = m_{t-1} + \gamma_2 \sum_{c=\{A,B,\ldots\}} \alpha(u_t - T_{c,t-1}) \tag{3.6}$$

Momentum at time t, m_t, is computed from the sum of decayed prediction errors across all contexts, with decay rate $0 \leq \gamma_2 \leq 0.8$. Scaling constant $0.01 \leq f \leq 3.0$ controls how much momentum influences threat prediction updates. Inputs are sequences of attacks and outputs are threat prediction sequences across all contexts.

3.3.5 TD-Momentum Model - Re-implementation

For the TD-Momentum model, threat estimations in unique contexts are no longer independent. Fig. 3.6 shows an example of an agent exposed to a similar simulation to Fig. 3.5. Comparing to the TD model in Fig. 3.5, attacks in any context influence estimations in all other contexts and Context C threat also increases, even though no attacks occur here; these effects are due to momentum. Estimations also plateau slowly when leaving a context, rather than abruptly plateauing like the TD model.

3.3.5.1 Re-implementation - Fitting to SEFL Data

We now reproduce the fitting and model comparison performed by Kaye et al. An example of fitting the smoothed freezing to the TD-Momentum model is shown in Fig. 3.7 (single stressed mouse ID G:34). This shows smoothed freezing behaviour observed on each day (top row), as well as the related TD-Momentum fitted threat (bottom row). The TD-Momentum model fits threat well, and shows the disproportionate freezing response of stressed mice to a single footshock on day 6 in context B. Kaye et al. note that this sensitised behaviour on day 6 is explained by the momentum term, which allows threat estimates to be linked across context A on day 1 and context B on day 6. For unstressed mice, TD and TD-momentum fits were very similar due to these mice not experiencing any shocks on day 1, thus not showing the same level of sensitised threat to the single shock on day 6 seen in stressed mice.

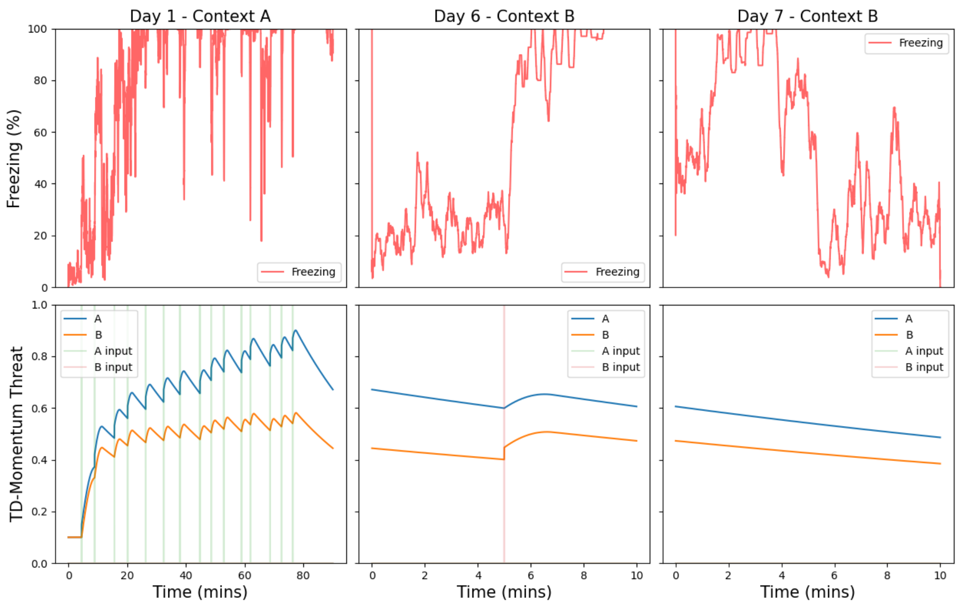

Figure 3.7: Observed smoothed freezing of stressed mouse (ID: G34) (top row) and corresponding TD-Momentum fitted threat (bottom row) across all days of the experiment. Blue and orange lines represent threat estimates in contexts A and B, respectively. Vertical lines represent input (footshocks) in contexts A (green) and B (red).

Fig. 3.8 shows the model comparison for all mice. Bayes Information Criterion (BIC) scores for maximum likelihood fits of each model were computed for each mouse and the difference ($BIC_{TD} - BIC_{TD\ Momentum}$) was taken to evaluate the preferred model. Stressed mice favoured the TD-Momentum model (14/15 mice), and control (unstressed) mice favoured the basic TD model (16/18 mice).

Taking the approach that the momentum model may be favoured as it allows for influence across contexts, we could argue that unstressed mice may have previously experienced threatening events in other contexts (before the experiment) that influence the freezing responses observed in the SEFL study. As such, it would be intuitive to hypothesise that *all* mice should have favoured the TD-Momentum model.

This leads to the question of how long momentum should persist after a series of events. Kaye et al. found this to depend on how long trauma itself persists. When attacks are correlated in time, including momentum provided a better estimation of true threat level in an environment. When attacks are uncorrelated, they found that momentum holds no advantage and the optimal $f = 0$ reduces to the basic TD model (Fig. A.10 Appendix A). As unstressed mice experience uncorrelated attacks (no attacks) on day 1, the f parameter tends towards 0, making the two models equivalent. As the TD-Momentum model has 4 parameters and the TD model has 2, the BIC

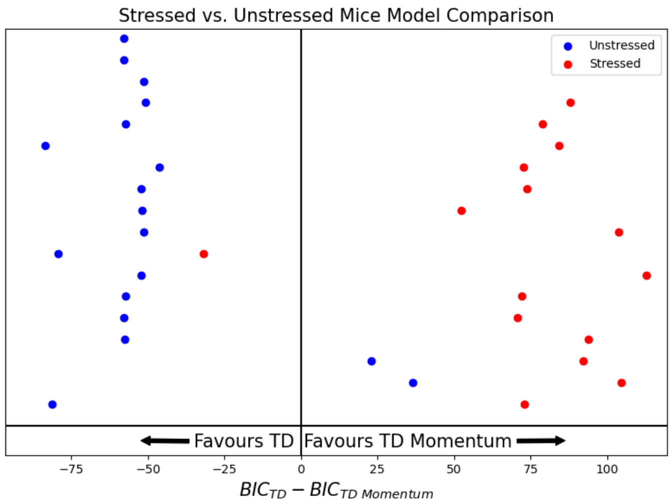

Figure 3.8: Model comparison for 15 stressed (red) and 18 unstressed (blue) mice. BIC scores were computed from the maximum likelihood fits of both models for each mouse.

scores for unstressed mice penalise the TD-momentum model more heavily, leading to the majority of control mice favouring the TD model. Stressed mice experienced correlated attacks on day 1, including the momentum term predicted increased freezing in the novel context (B) which resulted in a better fit, with momentum accounting for improved predictions compared to the TD model (related code in Appendices C.2-C.3 & D.2-D.3).

3.3.6 TD-Momentum Model - Additional Investigations

3.3.6.1 Additional Investigations - Model Behaviour

We now extend upon the findings of Kaye et al. with an analysis of TD-Momentum model behaviour, focusing on the additional parameters of momentum decay rate (γ_2) and scaling (f). Low values for these parameters encourage a slow summation of prediction errors across contexts and a normal observation of threat estimates summing to influence other contexts. However, larger values result in oscillatory behaviour of threat estimations across all contexts, with γ_2 being largely responsible for this as it controls the decay rate of summed prediction errors within the momentum term.

Fig. 3.9 shows TD-Momentum threat in response to 6 random ELS attacks. Typical values of $f = 0.2, \alpha = 0.07, \gamma_1 = 0.9999$ were used with the upper bound value of $\gamma_2 = 0.8$. This large γ_2 value causes context B threat to be larger in general. As momentum decreases after exiting context A, threat in both contexts reduces, threat

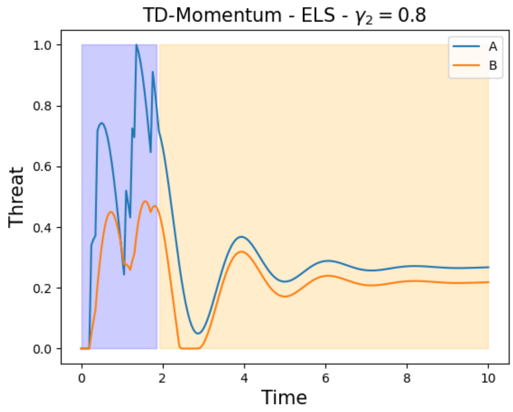

Figure 3.9: Maximum value of $\gamma_2 = 0.8$ causes oscillatory behaviour in threat estimations across all contexts. Attacks only occur in context A (blue highlight).

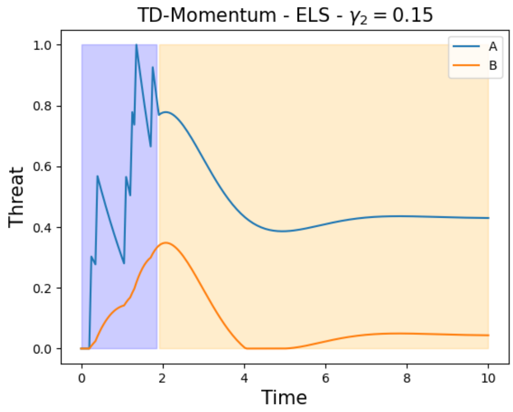

Figure 3.10: Similar to Fig 3.9, with $\gamma_2 = 0.15$. Small values of γ_2 can cause threat predictions to increase after reaching 0 for context B, even with no attacks.

in context B returns to 0 before a sudden increase into oscillatory behaviour. Fig. 3.10 shows that increasing γ_2 to as small as $\gamma_2 = 0.15$ can cause strange behaviour in estimations, where threat increases slightly after context B threat has reached zero. Maximising both parameters $f = 3$ and $\gamma_2 = 0.8$ results in extreme oscillatory behaviour for threat estimation which, in real life, would correspond to individuals experiencing cyclical periods of heightened sense of threat followed by periods of low sense of threat.

3.3.6.2 Additional Investigations - ELS Scenarios

Here we extend the findings of Kaye et al. by investigating the effects of repeated ELS on the TD momentum model. This has links to extinction learning (the basis for exposure therapy); the process by which associative threat responses are reduced by re-exposure to the original trauma context with no threat stimulus. This produces small, negative prediction errors that reduce threat in the trauma context over time.

Fig. 3.11 shows an ELS scenario of 6 random attacks in context A (none in context B). If the original context of repeated trauma is only visited briefly (e.g. for the duration of the trauma), after which it is exited, threat in both contexts plateaus at high level, explaining sensitisation (increase in response) to repeated threat. Threat level will not decrease further until the original trauma context is returned to and extinction learning can re-start, reducing threat in all contexts (via reduced momentum). Fig. 3.12 shows how threat estimations change when the original trauma context is re-entered, allowing for extinction learning to begin again. Threat for both contexts reduces more rapidly

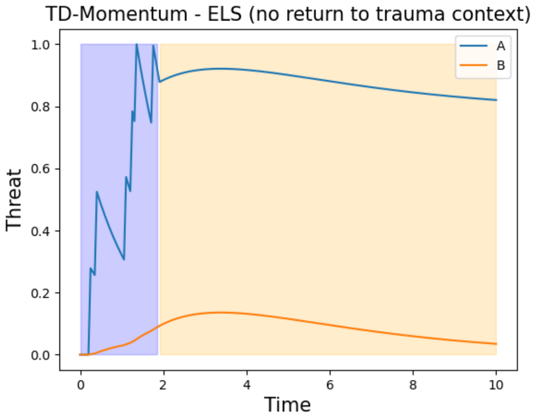

Figure 3.11: ELS of 6 random attacks in context A influences momentum and thus threat estimations in both contexts (blue for A, orange for B). Moving from context A to B, threat estimations both plateau.

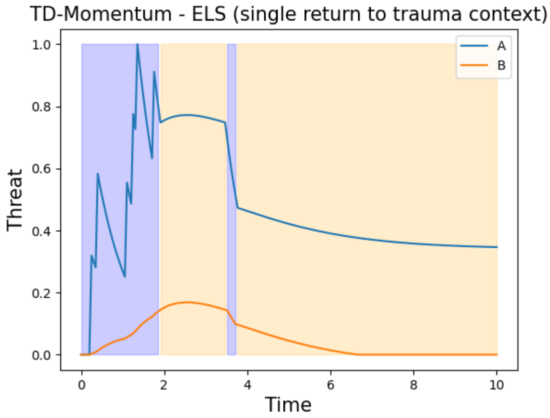

Figure 3.12: Similar to figure 3.11, however here we re-enter context A briefly before moving back to context B. Threat in both contexts decreases via momentum due to re-visiting trauma context A.

while in the original trauma context, even a short return to the original trauma context can have a large impact due to the larger prediction errors influencing momentum, context B threat actually returns to 0 following the re-entry. The rate of threat reduction depends on parameter values; larger learning rates, α, result in more rapid decrease.

We also investigate potential reasons for habituation, the phenomenon in which threat response decreases when exposed to repeated trauma. This has been shown to have links to numbing symptoms seen in cPTSD [17]. We compare ELS clusters followed by smaller singular threat with a scenario of singular, evenly spaced ELS. Fig. 3.13 shows threat estimates for an agent exposed to 5 singular, evenly spaced ELS instances in context A (none in context B). Threat in context A increases to around the same level after each attack, threat in context B slowly increases after each attack due to momentum. Upon entering context B, both threat levels reduce towards 0.

Fig. 3.14 shows the clustered ELS scenario where the first attack is replaced by a cluster of 4 attacks close in time, these represent more intense trauma in very early life. This cluster is followed by 4 single evenly spaced attacks. The initial cluster has a large impact on overall future threat estimations, whereby it creates a bias towards such intense threat and single attacks then result in a lower threat estimation in comparison to Fig. 3.13. This decrease in threat for prolonged trauma shows how habituation can occur in the model due to intensely clustered ELS. Momentum in the clustered ELS scenario influences context B threat largely, increasing in response to the cluster but

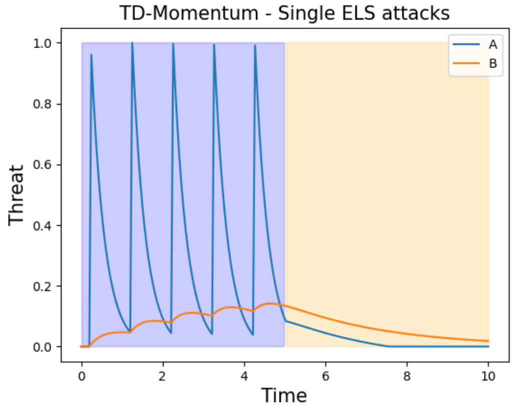

Figure 3.13: Single ELS scenario results in threat estimation in both contexts increasing with every attack (sudden spike).

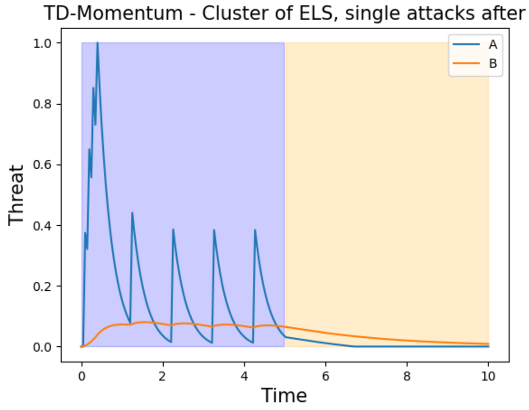

Figure 3.14: Clustered ELS scenario shows decreased response to single attacks, representing habituation in both contexts.

then slowly decreasing overall in response to singular attacks. This contrasts greatly to the single ELS scenario, where context B threat continually increases with every attack.

3.3.6.3 Additional Investigations - Implications for cPTSD

Here we review our previous findings with regards to implications for cPTSD and childhood trauma. Reviewing the oscillatory behaviour (Figs 3.9 and 3.10) related to variations in the γ_2 parameter, low γ_2 values appear "healthy", providing a natural summing of all historic contextual prediction errors, and a good account of non-associative threat learning. However, the oscillatory behaviour caused by large γ_2 values may be representative of fear learning maladaptations that result in issues with affect regulation and emotional reactivity typical of cPTSD, leading to heightened emotional and physical responses to small stressors. If this oscillatory behaviour began at an early age due to ELS, this would naturally lead to persistent disturbances in many aspects of life (family, social, educational). The duration of momentum persistence would also impact the timescale of such symptoms.

Oscillatory behaviour of threat predictions may also explain failures in extinction learning, and thus provides explanation for the large percentage of treatment dropout and failure for exposure therapy. When behaviour is on a downwards "recovery" slope, where extinction learning is functioning correctly, exposure to small stressors may be successful in reducing the learned threat response and the patient may seem to be recovering. However, when an oscillation occurs causing an upward "relapse" slope, exposure to small stressors may amplify and result in sensitisation (a drastically increased threat level), causing a failure in treatment and a persistent sense of heightened fear even after therapy. Individuals may then also avoid future therapy due to sustained elevated fear.

Regarding the ELS scenarios in Figs. 3.11 and 3.12, these provide potential explanations for the symptom clusters of avoidance and alterations in arousal and reactivity. These figures show threat to plateau at high levels due to momentum if the original trauma context is not returned to. This persistent elevated sense of threat may influence avoidance behaviours, i.e. contexts and situations may be avoided due to them being reminiscent of trauma. The remaining momentum due to the original trauma context then cannot be reduced as individuals avoid the context itself as well as reminders, meaning that threat remains at a fixed (albeit different) level for all contexts. Repeated, prolonged trauma across various contexts would amplify this effect, where individuals may avoid reminders of all original trauma contexts and thus avoid any opportunity for the combined momentum to reduce. This could be particularly prevalent in victims of childhood trauma and cPTSD, due to these individuals experiencing more prolonged trauma. Indeed, for those who experienced abuse from a trusted person (e.g. a family member) where abuse frequently took place in a childhood home, avoidance of reminders of this place would be an example of how momentum is unable to decrease. Thus causing a higher sense of threat in contexts with little association to the original trauma context, prompting heightened emotional (mistrust) and physical (hyperarousal) responses to events that may not be related to the original trauma.

Finally, we saw in Figs. 3.13 and 3.14 how habituation can occur in the model due to intensely clustered ELS. We noted how threat in context B was maintained at a particularly low level (via momentum), slightly decreasing as time moved on even though trauma was still occurring repeatedly. Threat in non-trauma contexts remaining at a low level (even decreasing) indicates that even though events are still occurring which may be extremely traumatic, the individuals perception of such events is no longer classing them as threatening, and so they are not affecting threat predictions in other contexts as they would in a "healthy" individual. In other words, this unresponsive momentum term could be viewed as the individual becoming accustomed to such trauma, habitualising their responses and effectively numbing them to such traumatic events in all contexts, leading to the emotional numbing symptoms typical of cPTSD.

3.3.6.4 Additional Investigations - Conclusion

To summarise, the TD-Momentum model presented by Kaye et al. incorporates the associative and non-associative aspects of fear learning in PTSD, where momentum allows for threat in any context to influence threat prediction across all other contexts.

Various ELS scenarios were investigated, and we proposed that the TD-Momentum model can be representative of various dysfunctions in fear learning which may be reason for several symptom clusters in cPTSD. We also investigated the oscillatory behaviour caused by momentum with larger values of the γ_2 parameter, noting that we may expect "healthy" individuals to be associated with lower values, and more symptomatic, "unhealthy" individuals to be associated with larger values. A moderately influential, "healthy" momentum term would represent appropriate non-associative fear learning across various contexts. However, highly influential, "unhealthy" momentum biases fear learning, causing oscillatory behaviour which may be linked to issues with affect regulation in cPTSD. Additional research into how this parameter varies across control and disorder groups may provide further explanations. If differences in the γ_2 parameter (or any other parameter for that matter) are found, this could provide avenues for the application of this model as a tool for determining disorder risk and/or trajectory.

In terms of creating a human behavioural task to investigate this (where results are fit to the TD-Momentum model), it's important to note that Kaye et al. define a single value function that simply estimates threat and has no "choice" to make. This could be adapted however, the value function of threat prediction could be applied to choices made in a human behavioural experiment where a task may be to learn threatening images over the course of several days (e.g. threatening images mixed among non-threatening images, varying in vagueness/difficulty). TD-Momentum fits could then be computed and differences between subject groups (control vs PTSD) compared to provide potential distinctions between levels of disorder severity, or even PTSD/cPTSD. Following the success of previous studies [7][14][23], combining behavioural data with neuroimaging may help to elucidate links between the model and how the neuromodulatory implementation of processes is represented in the brain.

Kaye et al. note the binary expression of attack and threat estimation as a drawback (0 for no shock/no freezing, 1 for shock/freezing). Obviously for humans the perception of attacks and threat is not so simple; both can be based over a broad spectrum that may vary across individuals. The authors propose that more precise manipulations of threat prediction errors over time may improve model validation.

Chapter 4

Extending the TD-Momentum Model of Kaye et al. (2023)

In this chapter we propose extensions to the TD-Momentum model that add value and explainability to improve the understanding of the disorder. We take several different approaches, noting motivation and advantages/disadvantages for each.

4.1 Incorporating Associability in TD-Momentum Model

Here we investigate the properties of the associability term proposed by Brown et al. [7] (section 2.1.2), and incorporate this into the TD-Momentum model.

Both the TD and TD-Momentum model base learning on updates via prediction errors; agents learn when a prediction is different to the actual outcome. Brown et al. incorporate this prediction error based learning with a dynamic associability term, assigning trial-by-trial associability values to each cue that scale over time with the historic prediction errors associated with that cue. Cues with a history of being unreliable/surprising receive larger associability which increases the dynamic learning rate related to that cue, causing a larger response. Thus, learning in this model is gated by the trial-by-trial associability value of each cue.

Incorporating this into the TD-Momentum model for threat prediction, where specific cues were given associability values by Brown et al., we propose that separate contexts are given associability values. The associability value of each context represents its influence to learning, and scales with the history of prediction errors that have occurred within that context. Therefore, contexts in which more traumatic events occur result in larger prediction errors and receive larger associability. Here, associability can be viewed as a context-dependent momentum, that only (directly) influences learning within its related context.

Brown et al. define associability to represent the attention given to a cue over time, they found that learning in individuals with PTSD was modulated by this attention-based learning, as they exhibited an increased learning response (increased learning *rate*) to unexpected cues. We hypothesise that such individuals may have similar sensitised learning responses to high associability contexts (highly traumatic contexts).

4.1.1 Associability TD-Momentum Model Form

Combining Eq. (2.1) and Eq. (2.3), the proposed model takes the form below. Momentum is updated identically to the standard TD-Momentum model, and the trial-by-trial associability value of context c at time t is denoted $\kappa_{c,t}$ with a lower bound of 0.05.

$$T_{c,t} = T_{c,t-1} + \alpha\kappa_{c,t-1}(u_t - \gamma_1 T_{c,t-1}) + fm_t \tag{4.1}$$

$$m_t = m_{t-1} + \gamma_2 \sum_{c=\{A,B,\ldots\}} \alpha(u_t - T_{c,t-1}) \tag{4.2}$$

Prediction errors for each context are computed similar to the standard TD-Momentum model. Associability for each context is updated at each time step as follows,

$$\kappa_{c,t+1} = (1-\eta)\kappa_{c,t} + \eta|u_t - \gamma_1 T_{c,t-1}| \tag{4.3}$$

The associability weight parameter, $0 \leq \eta \leq 1$, controls how much historic prediction errors influence future associability values (this is general and not unique per context).

4.1.2 Implications of Associability TD-Momentum

As associability essentially creates unique, dynamic learning rates for each context, this model provides us with insights into how different contexts (traumatic/non-traumatic) influence learning dynamically over time. If threat learning were modulated by attention-based learning where traumatic contexts are more influential to learning, this model may explain why strong maladaptive cPTSD priors are so difficult to target via treatment.

Fig. 4.1 shows an example simulation with an input of 10 random attacks in context A (none in context B), associability for each context is initialised at 1 (similar to Brown et al. [7]). Associability of context A returns to the lower bound 0.05 rapidly after attacks, while associability of context B remains at the lower bound throughout. However, an example of the sensitised learning influence of traumatic context A is seen

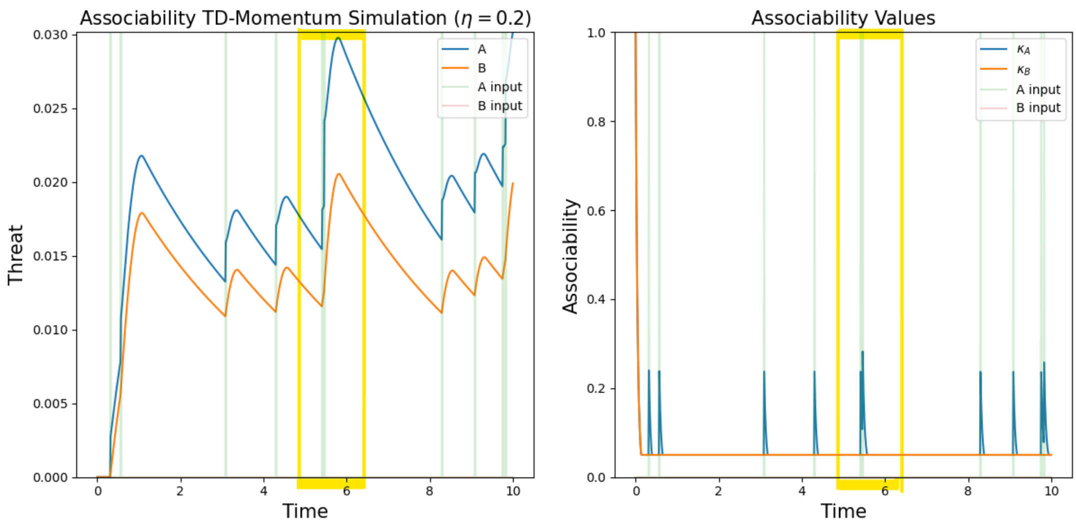

Figure 4.1: Identical parameters and attack sequence to TD-Momentum Simulation in 4.2. Associability TD-Momentum threat (left) with associability weight parameter $\eta = 0.2$ and related associability values (right). Associability model generates smoother threat predictions than TD-Momentum, with threat being much similar between both contexts.

in the highlighted box. Two attacks occur here in quick succession, threat in context A increases more due to the second attack than threat in context B (left plot), this can be explained by associability. Context A associability κ_A does not reach 0 before the second attack occurs (right plot), causing a larger spike in associability for context A when the second attack is experienced. This results in a larger dynamic learning rate at this time, meaning that context A threat increases by a larger amount compared to context B threat.

The associability of each context can *indirectly* effect learning in all other contexts through the momentum term (computed from the decayed sum of prediction errors across *all* contexts). As traumatic, unpredictable contexts have larger associability values (thus larger dynamic learning rates), the prediction errors generated in these contexts will be more influential to learning compared to those generated in less traumatic, predictable contexts. Meaning they will contribute more to momentum, i.e. trauma in unpredictable contexts will influence momentum more than the same trauma in predictable contexts.

This may cause threat estimations in more predictable contexts to be heavily influenced (via momentum) by unpredictable contexts, which could explain why threat predictions for both contexts in our simulation are so similar (Fig. 4.1). Prolonged childhood trauma would strengthen the large overpowered associability values for traumatic contexts, as this would ensure they maintain high associability values, amplifying this effect, perhaps even bringing threat predictions in all contexts to the same level as the original trauma context.

This model provides a method for investigating the increased effect of salient traumatic contexts, *why* individuals with PTSD can be so affected by reminders of the context of the trauma, and may explain extinction learning failures in treatment. For cPTSD, it offers a potential explanation for issues with affect regulation (e.g. heightened emotional reactivity to small stressors/contexts unrelated to original trauma). Associability creates a dynamic learning rate for each context that is the same for both positive and negative prediction errors within that context. Meaning, there is no difference between learning from positive or negative prediction errors in a given context. As learning in PTSD individuals may be modulated by unexpected experiences [14], incorporating separate learning rates for positive and negative prediction errors may lead to an improved representation of cPTSD learning. This is explored in the next section.

4.2 Incorporating Outcome-Sensitivity in TD-Momentum Model

Here we incorporate the "risk-sensitive" element of the RL model proposed by Ross et al. (section 2.1.3), although here we term the approach as "outcome-sensitive". The motive here being that PTSD individuals may learn differently based on whether an outcome is better or worse than their prediction (i.e. the sign of the prediction error signifying good news or bad news). As such, splitting the single learning rate of the TD-Momentum model into separate positive and negative learning rates may provide an improved representation of learning. This model explores how threat learning may be modulated by how one perceives outcomes. Including momentum here incorporates non-associative learning, allowing for prediction errors in any context to influence threat in all other contexts.

4.2.1 Outcome-Sensitive TD-Momentum Model Form

The form is identical to the original model, with the learning rate used depending on the sign of the prediction error $(u_t - \gamma_1 T_{c,t-1})$. Separate learning rates are reflected within the momentum term where we have inserted the γ_1 decay rate to ensure prediction errors are consistent between momentum calculation and the main update rule.

$$T_{c,t} = T_{c,t-1} + \alpha^{+/-}(u_t - \gamma_1 T_{c,t-1}) + f m_t \tag{4.4}$$

$$m_t = m_{t-1} + \gamma_2 \sum_{c=\{A,B,\ldots\}} \alpha^{+/-}(u_t - \gamma_1 T_{c,t-1}) \tag{4.5}$$

Learning rate is: α^+ when the prediction error is positive (larger than expected outcome), and α^- when the prediction error is negative (smaller than expected outcome). Thus, large, unexpected threat outcomes cause positive prediction errors here, so α^+ relates to learning from very traumatising experiences.

4.2.2 Implications of Outcome-Sensitive TD-Momentum

Hauser et al. [13] implemented a similar outcome-sensitive RL model investigating developmental aspects of cognitive flexibility (the ability to adapt to unplanned events) in adolescents, they compared adolescents and adults who performed a probabilistic reversal learning task. They found that adolescents had increased sensitivity to *negative* predictions errors compared to adults (where the outcome is less than expected, i.e. a punishment). These findings may also translate to the threat learning context. Although, we expect that in this scenario, where we have assigned a positive learning rate to larger than expected outcomes (large threat or trauma), that learning is more sensitive to *positive* prediction errors, as these represent more intense, unexpected trauma. Fig. 4.2 shows a simulation of this model. The larger α^+ is used following underestimated outcomes (attacks) and causes large increases in threat estimation in comparison to the original TD-Momentum model.

By introducing positive and negative learning rates, momentum updates will vary based on whether the outcome prediction is over-estimated or under-estimated, leading to differences in how non-associative threat is learned compared to the standard TD-momentum model. Fig. 4.2 shows how threat in context B is larger than in the standard simulation, in line with the larger α^+ used. Thus, threat transferred through non-associative learning is influenced by these outcome-sensitive learning rates, and could lead to drastic variations of threat in unrelated contexts. This may have links to the increased reactivity to minor stressors unrelated to original trauma, typical of PTSD.

Investigating differences in learning sensitivity (positive and negative learning rates) between PTSD/control individuals who experience various threatening outcomes may provide insight on how learning mechanisms behave when outcomes are perceived

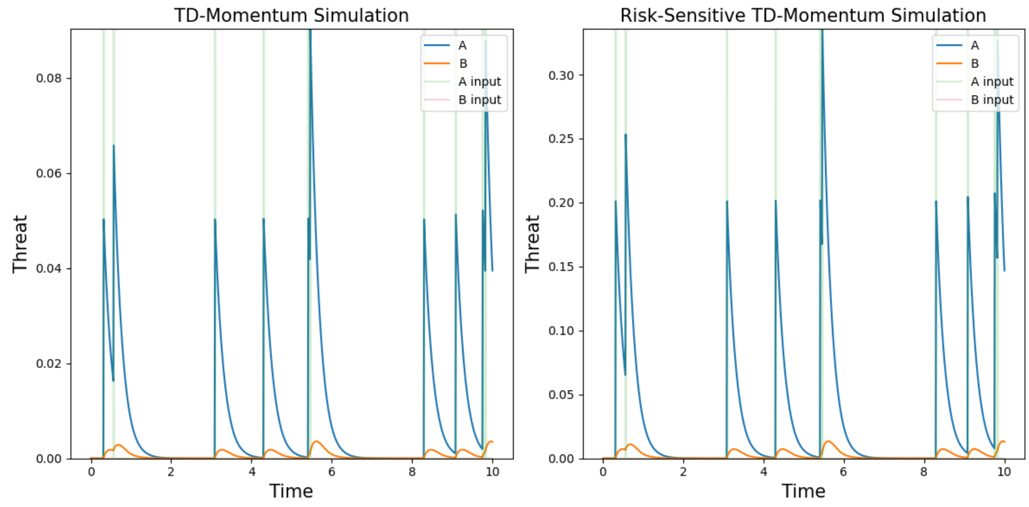

Figure 4.2: TD-Momentum plot (left) with parameters: $\alpha = 0.05, \gamma_1 = 0.9999, \gamma_2 = 0.05, f = 0.1$. Risk-Sensitive TD Momentum threat (right): $\alpha^- = 0.05, \alpha^+ = 0.2$. 10 random attacks across life in context A, none in B. Positive prediction errors use the larger α^+ causing larger overall threat and increases across both contexts (right).

differently by different groups. By reviewing any distinct group differences, this could provide the basis for a tool to determine PTSD risk and/or disorder trajectory.

A potential pitfall for this model (and the previous associability model) is the binary representation of inputs (0 for no attack, 1 for attack). This basic manipulation of threat sequences may result in lack of precision for fitted parameters. A model that can take a more continuous representation of threat as input allows for more complex threat sequences to be explored (e.g. between -1 and 1), and may provide more insight on fitting to behavioural data. A potential is explored in the final extension.

4.3 Incorporating Valence-Partitioning in TD-Momentum Model

Liebenow et al. [18] and Sands et al. [24] propose Valence Partitioning (VP) as a methodology for decoupling the algorithmic representation of rewards and punishments. They found that VP RL was effective at predicting dynamic changes in human choice behaviour and subjective experience. Consequently, they suggest that VP RL can be effective in deriving insights on mechanisms related to psychiatric disorders.

Standard TD learning treats punishment and reward as opposite ends of a single reward spectrum. Liebenow et al. propose this may not accurately reflect the true processes of how learning (and associated behaviour) operates. VP TD learning maintains the successful process of TD learning, but implements two independent, parallel positive and negative valence systems. Liebenow et al. [18] apply a continuous outcome scale between -1 and +1, where -1 represents maximum punishment and +1 represents maximum reward. A valence threshold is set at 0, i.e. rewards are positively valenced and punishments are negatively valenced. Positively valenced outcomes are processed via the positively valenced system, and negatively valenced outcomes are processed via the negatively valenced system. These parallel systems allow for asymmetric representations of learning from positively and negatively valenced outcomes. We use VP RL here to capture any asymmetries in learning due to PTSD, exploring if threat learning is modulated by *how* individuals valence different outcomes.

4.3.1 Valence-Partitioned TD-Momentum Model Form

Determining which valence system processes the outcome, how prediction errors are computed, and which learning rate is used, depends only on the valence of the outcome. Valenced prediction errors are generated as shown in Eqs. (4.6) and (4.7). Outcome values are within $-1 \leq u_t \leq 1$, where 1 represents the most safe, pleasant outcomes, and -1 represents the most threatening outcomes (0 represents a null outcome). If outcome valence is not within the receptive field of a system (where we have set a $threshold = 0$), the outcome is treated as a null outcome for that system. Prediction error in context c at time t for each valence system is computed as follows:

$$\delta^P_{c,t} = \begin{cases} u_t + \gamma^P V^P_{c,t+1} - V^P_{c,t} & if\ u_t > 0 \\ 0 + \gamma^P V^P_{c,t+1} - V^P_{c,t} & if\ u_t \leq 0 \end{cases} \tag{4.6}$$

$$\delta^N_{c,t} = \begin{cases} |u_t| + \gamma^N V^N_{c,t+1} - V^N_{c,t} & if\ u_t < 0 \\ 0 + \gamma^N V^N_{c,t+1} - V^N_{c,t} & if\ u_t \geq 0 \end{cases} \tag{4.7}$$

Two value functions are generated, one for appetitive values (Positive system - Eq. (4.8) processes pleasant outcomes) and one for aversive values (Negative system - Eq. (4.9) processes threatening outcomes).

$$V^P_{c,t+1} = V^P_{c,t} + \alpha^P \cdot \delta^P_{c,t} + fm^P_t \tag{4.8}$$

$$V^N_{c,t+1} = V^N_{c,t} + \alpha^N \cdot \delta^N_{c,t} + fm^N_t \tag{4.9}$$

Separate momentum terms are created for positively and negatively valenced outcomes across all contexts. Update form remains the same, only we include the corresponding positive- or negative-specific prediction errors ($\delta^P_{c,t} or \delta^N_{c,t}$) and learning rates ($\alpha^P$ or α^N) based on valence of the outcome.

$$m^P_t = m^P_{t-1} + \gamma_2 \sum_{c=\{A,B,...\}} \alpha^P \delta^P_{c,t} \tag{4.10}$$

$$m^N_t = m^N_{t-1} + \gamma_2 \sum_{c=\{A,B,...\}} \alpha^N \delta^N_{c,t} \tag{4.11}$$

Appetitive and aversive value functions are then combined to create the overall threat prediction in any context at any given time.

$$T_{c,t} = V^P_{c,t} - V^N_{c,t} \tag{4.12}$$

4.3.2 Implications of Valence-Partitioned TD-Momentum

Tracking the distinct valenced value functions and momentum terms will show us how behaviour of pleasantness/threat predictions across contexts varies between stimuli classed as "pleasant" and "threatening", i.e. positive and negative valence. Differences in group behaviour (PTSD/control) observed could provide explanations for how different stimuli can affect predictions over a lifetime and why symptom timescale and severity may differ between individuals (or groups). Regarding the distinct positively- and negatively-valenced momentums, we would expect the negatively valenced momentum to be larger and more influential to learning across all contexts as it represents the non-associative threat across all contexts from threatening outcomes. Comparing valenced momentum terms to the original single momentum will show, specifically, how much threatening outcomes can influence non-associative learning across all contexts, compared to positive outcomes.

A simulation of the VP TD-Momentum model with a larger negative learning rate is shown in Fig. 4.3, exposed to 10 positive (0.5, 1) and 10 negative outcomes (-1, -0.5), representing a broader spectrum of pleasant/threatening outcomes compared to previous extensions. This manipulation of outcomes leads to more diverse pleasant/threatening estimations in both contexts. Context B predictions (due to momentum) are small in value, oscillating around the neutral outcome of 0 due to the combination of positive and negative outcomes. The larger α^N parameter shows a disproportionate effect on learning from negatively valenced outcomes compared to positively valenced, shown by the downward spikes being more pronounced. The related VP value functions

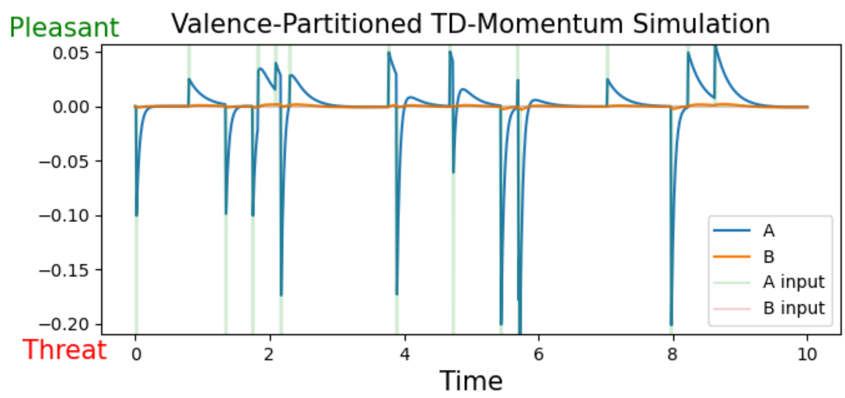

Figure 4.3: Valence-Partitioned TD-Momentum fit with identical parameters to TD Momentum Simulation in Fig. 4.2, and $\alpha^P = 0.05$, $\alpha^N = 0.2$. 10 positive (pleasant) outcomes and 10 negative (threat) stimuli. Larger negative learning rate α^- causes negative outcomes to influence learning more.

for both contexts, as well as the valenced momentum terms can be found in Figs. A.14, and A.15 of Appendix A. As expected, the value functions for context B are much lower in absolute value compared to context A; the negatively valenced value functions for both contexts are larger than the positively valenced, with the negatively valenced value function for trauma context A being disproportionately large (due to events occuring here) (Fig. A.14 left, orange). The negatively valenced momentum from threatening outcomes is larger and more influential to learning than the positively valenced momentum (Fig. A.15, orange) as expected and noted above. This is reflected in the overall fit on Fig. 4.3, negative valenced outcomes effect momentum more and thus are more influential to context B threat compared to positively valenced outcomes.

The *threshold* value of 0 may be subjective and investigations into fitting this to individuals/groups may provide insight on the heightened (or lessened, i.e. habituation) sensitivity to threats in PTSD. For example, this threshold value may be dysfunctionally large (e.g. 0.3) for highly symptomatic individuals, meaning they perceive small, pleasant outcomes as negative outcomes, processing them through the negatively valenced system, causing the negatively valenced system to influence learning more.

Although not applied by Liebenow et al. or Sands et al., we could include sensitivity parameters for positively and negatively valenced outcomes (β^P, β^N), similar to parameters proposed by Yanamori et al. (section 2.1.4). Overall pleasantness/threat prediction is then $T_{c,t} = \beta^P V^P_{c,t} - \beta^N V^N_{c,t}$. An index similar to the "reward-punishment sensitivity index" of Yanamori et al. could then be created for each individual, which we label the "valence sensitivity index" β^P/β^N, giving a unique value to each participant corresponding to where they lie on the positive vs negative valence sensitivity spectrum. Higher values correspond to individuals whose learning is more influenced by positively valenced outcomes, while lower values correspond to those more influenced by negatively valenced outcomes. Further investigations into this VP TD-Momentum model may provide more complex insights into the underlying mechanisms of cPTSD.

4.4 TD-Momentum Extensions Summary

We have presented three different approaches to extending the TD-Momentum model, incorporating concepts from various RL models of associability, outcome-sensitivity, and valence partitioning in order to offer further insight into the disorder. The momentum term is maintained in all extensions, allowing for non-associative threat learning.

Associability (brown et al. [7] and Homan et al. [14]) was integrated with the motive that trial-by-trial associability values, which are influenced by a contexts history of prediction errors, may modulate learning in each context. Considering the heightened learning resulting from high associability cues found by Brown et al., (also shown in our simulation) this approach suggests that similar attention-based modulation of learning may exist in threat learning across various contexts.

The outcome-sensitive (Ross et al. [23]) integration explores how learning rates may vary between individuals for positive and negative prediction errors, where PTSD/cPTSD may be linked with heightened sensitivity to one set of prediction errors.

The integration of valence-partitioning (Liebenow et al. [18] and Sands et al. [24]) creates separately valenced value functions positively- and negatively-valenced outcomes which are combined to create an overall pleasantness/threat prediction. This aims to capture how asymmetries in learning may be based on how individuals valence outcomes differently.

By applying results of relevant behavioural tasks performed by control and disorder-affected participants we can analyse variations of parameters and model behaviour between groups. Applying such behavioural data to these proposed extensions is thus key in indicating how much value they can add with regards to explaining the disorder, and will reveal how associability, outcome-sensitivity, and valence partitioning can influence threat prediction learning.

Chapter 5

Conclusion

The aim of this dissertation has been to re-implement the TD-Momentum model proposed by Kaye et al. [16], assess the capability of this model for explaining mechanisms of cPTSD, and provide potential model extensions that can add value and further insight.

We re-implemented the Bayesian baseline model proposed by Kaye et al. and found it predicted the probability of attack well, showing how a simulated ideal observer may perform in a Bayesian setting when predicting attack rate over a lifetime. Through ELS scenarios we showed how various types of trauma in early life can cause a disproportionate affect on threat prediction, with clustered attacks in early life causing larger mean estimates and variance in predictions.

TD and TD-Momentum RL models were then re-implemented and fit to data collected by an SEFL experiment. Mouse freezing data was fit and the model comparison showed that stressed mice favoured the TD-Momentum model while unstressed mice favoured the simpler TD model. We concluded similar to Kaye et al., that the stressed mice prefer the inclusion of the momentum term due to the incorporation of non-associative learning, i.e. momentum allows for threat predictions from the stressful day 1 context A to influence predictions across days 6 and 7 in context B.

Following this, we extended the findings of Kaye et al. by performing our own investigations into TD-Momentum model behaviour and how ELS scenarios affect predictions across contexts. We suggest that the oscillatory behaviour built into the momentum term due to certain parameter values may reflect differences between healthy and highly symptomatic individuals, where more oscillatory behaviour in momentum may have links to issues with affect regulation typically seen in cPTSD. The ELS scenarios showed that clustered attacks experienced in early life seem to have a larger disproportionate impact on threat learning within the original trauma context (similar to the Bayesian model), as well as in novel contexts via momentum. We found that clustered attacks administered in early life may also result in habituation to future attacks, which may be linked with the emotional numbing effects seen in cPTSD.

We also found similar results to Kaye et al. in relation to how extinction learning can be negatively affected by the momentum term, and may affect treatments like exposure therapy by generating a sensitised effect, rather than habituation. We suggest that a maintained high sense of threat in a recently exited trauma context, due to increased momentum, may cause threat in other contexts to be maintained at a higher rate than usual, resulting in symptom clusters such as increased reactivity/arousal and avoidance which may lead to difficulties in various aspects of life.

After exploring how much the TD-Momentum model can explain we offered three potential extensions which may add value and provide us with a tool for investigating symptoms, individual risk and implications to treatments. Proposed methodologies included: an associability term to act as a context-dependent momentum, the inclusion of outcome-sensitivity to assign different learning rates for positive/negative prediction errors, and the idea of partitioning positive (pleasant outcomes) and negative (threatening outcomes) learning based on valence of the outcome at each timestep.

These extensions aim to combat some of the pitfalls of the standard TD-Momentum model, such as: the heightened influence to learning that some (traumatic) contexts may have (associability model addresses this), the uniform view of how different outcomes may influence prediction more/less (constant learning rate for all outcomes, outcome-sensitive model addresses this), and the fact that learning from safe, pleasant outcomes and threatening outcomes may be processed by entirely different systems, based on how individuals interpret outcomes (valence partitioning addresses this).

Ultimately, further investigation into these extensions (and similar computational models) will improve the understanding of the underlying mechanisms of cPTSD, which is vital in defining the distinct differences and similarities between PTSD and other psychiatric disorders. These findings will be key in discovering the best methods for risk assessment, treatment planning and mapping trajectory of the disorder. Although computational psychiatry is still at a youthful stage, there is an increasing amount of research being published in this field. The findings from such research hope to someday (in the near future) be applied for clinical uses to improve the quality of life for all those affected by such mental health problems.

Bibliography

[1] Python Package Index - PyPI. Publisher: Python Software Foundation.

[2] Robin Achterhof, Rafaële J. C. Huntjens, Marie-Louise Meewisse, and Henk A. L. Kiers. Assessing the application of latent class and latent profile analysis for evaluating the construct validity of complex posttraumatic stress disorder: cautions and limitations. *European Journal of Psychotraumatology*, 10(1):1698223, December 2019.

[3] Laurence Aitchison and Máté Lengyel. With or without you: predictive coding and Bayesian inference in the brain. *Current Opinion in Neurobiology*, 46:219–227, October 2017.

[4] American Psychiatric Association. *Diagnostic and statistical manual of mental disorders (5th ed.)*. American Psychiatric Publishing, 2013.

[5] Jonathan I Bisson. Post-traumatic stress disorder. *BMJ*, 334(7597):789–793, April 2007.

[6] Kathleen T Brady, Tim Brewerton, Therese Killeen, and Sylvia Lucerini. Comorbidity of Psychiatric Disorders and Posttraumatic Stress Disorder. *J Clin Psychiatry*, April 2000.

[7] Vanessa M Brown, Lusha Zhu, John M Wang, B Christopher Frueh, Brooks King-Casas, and Pearl H Chiu. Associability-modulated loss learning is increased in posttraumatic stress disorder. *eLife*, 7:e30150, January 2018.

[8] Marylène Cloitre, Chris R. Brewin, Jonathan I. Bisson, Philip Hyland, Thanos Karatzias, Brigitte Lueger-Schuster, Andreas Maercker, Neil P. Roberts, and Mark Shevlin. Evidence for the coherence and integrity of the complex PTSD (CPTSD) diagnosis: response to Achterhof et al., (2019) and Ford (2020). *European Journal of Psychotraumatology*, 11(1):1739873, December 2020.

[9] Eran Eldar and Yael Niv. Interaction between emotional state and learning underlies mood instability. *Nature Communications*, 6(1):6149, January 2015.

[10] Eran Eldar, Robb B. Rutledge, Raymond J. Dolan, and Yael Niv. Mood as Representation of Momentum. *Trends in Cognitive Sciences*, 20(1):15–24, January 2016.

[11] Daniel Foreman-Mackey, David W. Hogg, Dustin Lang, and Jonathan Goodman. emcee : The MCMC Hammer. *Publications of the Astronomical Society of the Pacific*, 125(925):306–312, March 2013.

[12] Jonathan Goodman and Jonathan Weare. Ensemble samplers with affine invariance. *Communications in Applied Mathematics and Computational Science*, 5(1):65–80, January 2010.

[13] Tobias U. Hauser, Reto Iannaccone, Susanne Walitza, Daniel Brandeis, and Silvia Brem. Cognitive flexibility in adolescence: Neural and behavioral mechanisms of reward prediction error processing in adaptive decision making during development. *NeuroImage*, 104:347–354, January 2015.

[14] Philipp Homan, Ifat Levy, Eric Feltham, Charles Gordon, Jingchu Hu, Jian Li, Robert H. Pietrzak, Steven Southwick, John H. Krystal, Ilan Harpaz-Rotem, and Daniela Schiller. Neural computations of threat in the aftermath of combat trauma. *Nature Neuroscience*, 22(3):470–476, March 2019.

[15] Eric Jones, Travis Oliphant, Pearu Peterson, and others. SciPy: Open source scientific tools for Python, 2001.

[16] Alfred P. Kaye, Manasa G. Rao, Alex C. Kwan, Kerry J. Ressler, and John H. Krystal. A computational model for learning from repeated traumatic experiences under uncertainty. *Cognitive, Affective, & Behavioral Neuroscience*, May 2023.

[17] Ye Ji Kim, Sanne J.H. Rooij, Timothy D. Ely, Negar Fani, Kerry J. Ressler, Tanja Jovanovic, and Jennifer S. Stevens. Association between posttraumatic stress disorder severity and amygdala habituation to fearful stimuli. *Depression and Anxiety*, 36(7):647–658, July 2019.

[18] Brittany Liebenow, Rachel Jones, Emily DiMarco, Jonathan D. Trattner, Joseph Humphries, L. Paul Sands, Kasey P. Spry, Christina K. Johnson, Evelyn B. Farkas,

Angela Jiang, and Kenneth T. Kishida. Computational reinforcement learning, reward (and punishment), and dopamine in psychiatric disorders. *Frontiers in Psychiatry*, 13:886297, October 2022.

[19] Milen L. Radell, Catherine E. Myers, Jony Sheynin, and Ahmed A. Moustafa. Computational Models of Post-traumatic Stress Disorder (PTSD). In *Computational Models of Brain and Behavior*, pages 43–55. John Wiley & Sons, Ltd, 2017. Section: 4 _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781119159193.ch4.

[20] Vinuta Rau, Joseph P. DeCola, and Michael S. Fanselow. Stress-induced enhancement of fear learning: An animal model of posttraumatic stress disorder. *Neuroscience & Biobehavioral Reviews*, 29(8):1207–1223, 2005.

[21] Robert Rescorla. *A theory of Pavlovian conditioning : Variations in the effectiveness of reinforcement and nonreinforcement*. Appleton-Century-Crofts, New York, 1972.

[22] Kerry. J. Ressler, Sabina Berretta, Vadim Y. Bolshakov, Isabelle M. Rosso, Edward G. Meloni, Scott L. Rauch, and William A. Carlezon. Post-traumatic stress disorder: clinical and translational neuroscience from cells to circuits. *Nature Reviews Neurology*, 18(5):273–288, May 2022.

[23] Marisa C. Ross, Jennifer K. Lenow, Clinton D. Kilts, and Josh M. Cisler. Altered neural encoding of prediction errors in assault-related posttraumatic stress disorder. *Journal of Psychiatric Research*, 103:83–90, August 2018.

[24] L. Paul Sands, Angela Jiang, Rachel E. Jones, Jonathan D. Trattner, and Kenneth T. Kishida. Valence-partitioned learning signals drive choice behavior and phenomenal subjective experience in humans. preprint, Neuroscience, March 2023.

[25] Skipper Seabold and Josef Perktold. statsmodels: Econometric and statistical modeling with python. In *9th Python in Science Conference*, 2010.

[26] Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: an introduction*. Adaptive computation and machine learning series. The MIT Press, Cambridge, Massachusetts, second edition edition, 2018.

[27] Guido Van Rossum and Fred L. Drake. *Python 3 Reference Manual*. CreateSpace, Scotts Valley, CA, 2009.

[28] Christopher Watkins and Peter Dayan. Q-Learning. *Kluwer Academic Publishers*, pages 279–292, 1992.

[29] World Health Organization (WHO). *International Classification of Diseases, Eleventh Revision (ICD-11)*. (WHO), 2019.

[30] Sam Wilkinson, Guy Dodgson, and Kevin Meares. Predictive Processing and the Varieties of Psychological Trauma. *Frontiers in Psychology*, 8:1840, October 2017.

[31] Yumeya Yamamori, Oliver J Robinson, and Jonathan P Roiser. Approach-avoidance reinforcement learning as a translational and computational model of anxiety-related avoidance. preprint, elife, June 2023.

Appendix A

Additional Plots

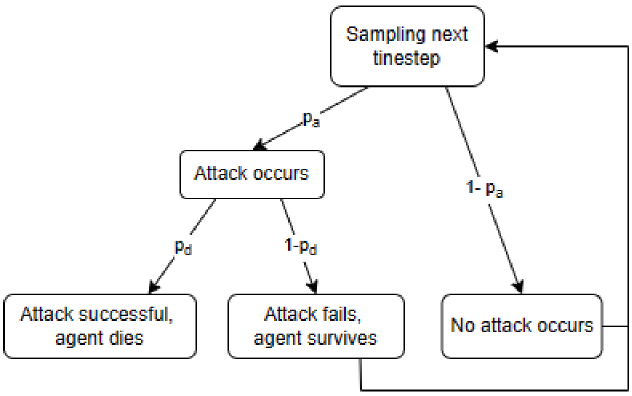

Figure A.1: Bayesian Probabilistic model of trauma attack estimation (re-created from Kaye et al. [16]). Attacks continue until agent death or end of timesteps.

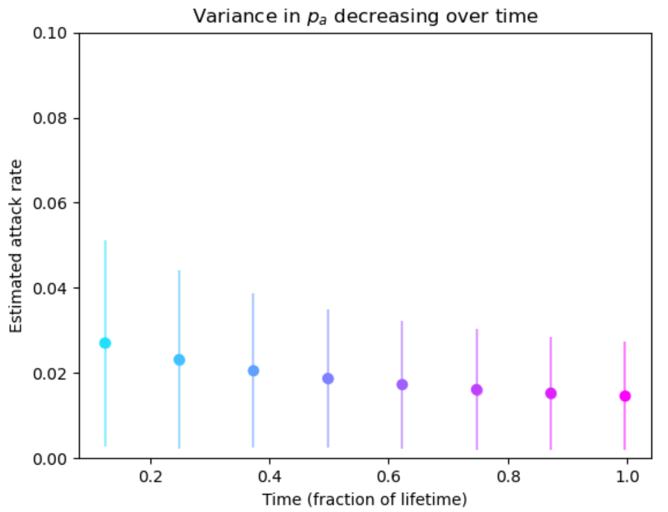

Figure A.2: Variance in p_a estimation decreases over the course of a lifetime, showing how the Bayesian model progressively improves and becomes more confident in its estimates. This plot shows the variance in p_a over a lifetime for a typical sequence, similar to that used in Fig. 3.1 (related code in Appendix B.4)

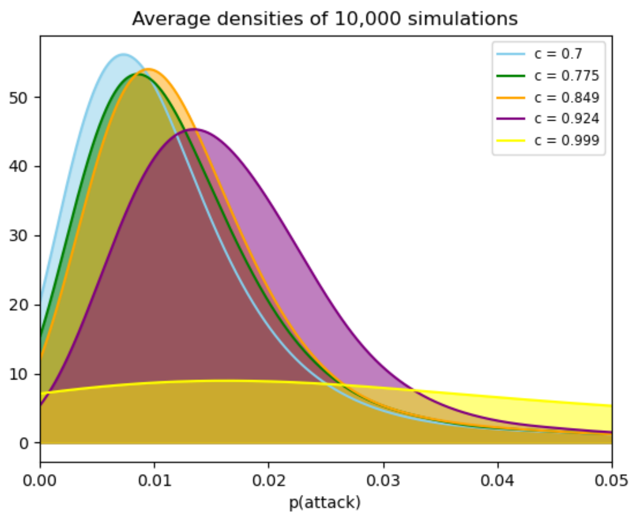

Figure A.3: Densities "averaged" over 10,000 simulations of AR time series for each correlation value. Larger AR correlation results in larger variance for threat predictions of p_a. Sharper, more precise predictions are given by $c = 0.7$, while flatter, less precise predictions given by $c = 0.999$.

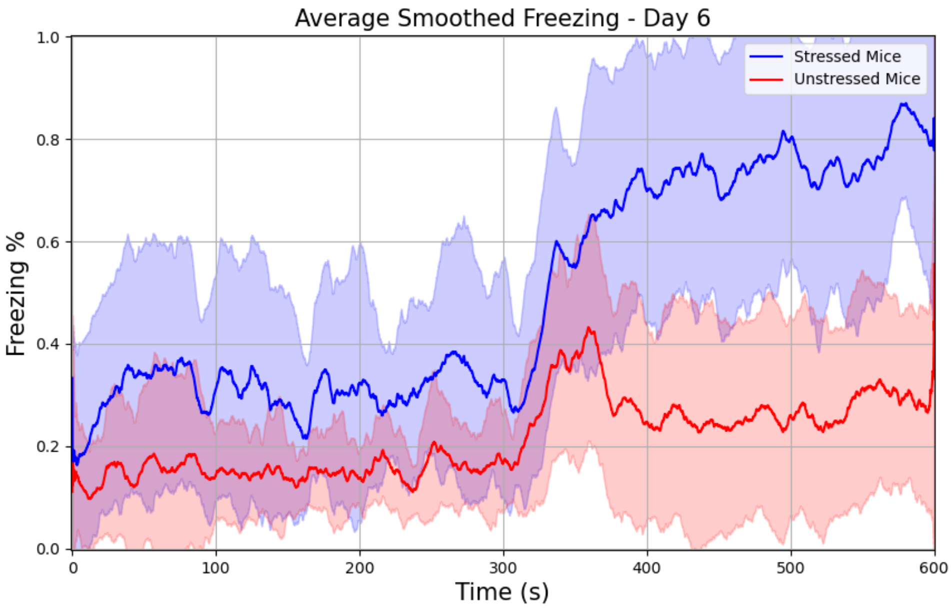

Figure A.4: Average freezing of 15 stressed and 18 unstressed mice across day 6 of the SEFL experiment.

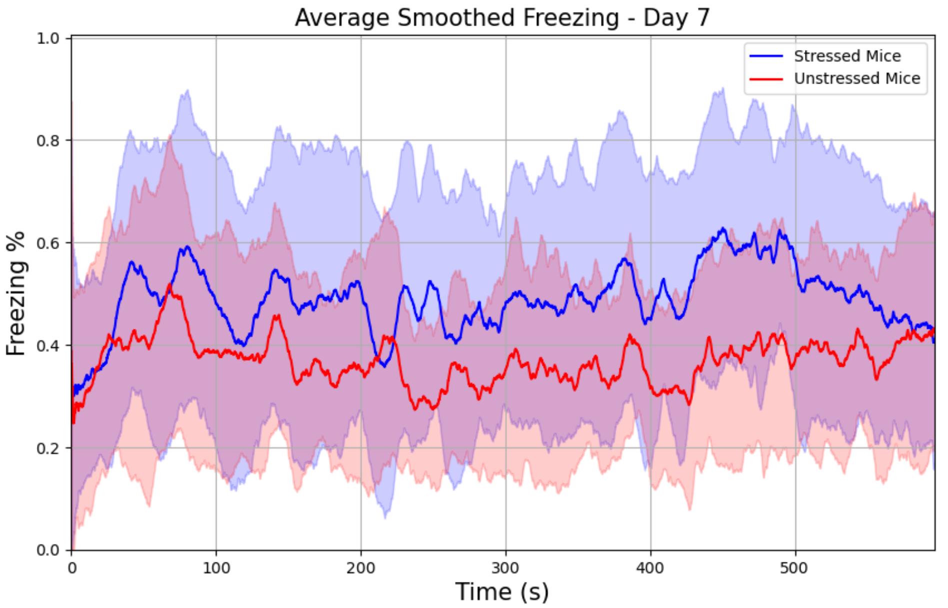

Figure A.5: Average freezing of 15 stressed and 18 unstressed mice across day 7 of the SEFL experiment.

TD Model - Multivariate Sampled Parameters - Recovery

α Coeff	α P-Val	γ_1 Coeff	γ_1 P-Val
0.9994	0.0000	0.8775	0.0000
0.9993	0.0000	0.9647	0.0000
1.0000	0.0000	0.9993	0.0000
1.0000	0.0000	0.9299	0.0000
0.9986	0.0000	0.9345	0.0000

Figure A.6: TD model parameter recovery Pearson correlation coefficients and p-values for multivariate sampled parameter values

TD Model - List Sampled Parameters - Recovery

α Coeff	α P-Val	γ_1 Coeff	γ_1 P-Val
1.0000	0.0000	0.9999	0.0000
1.0000	0.0000	1.0000	0.0000
1.0000	0.0000	1.0000	0.0000
1.0000	0.0000	0.9999	0.0000
1.0000	0.0000	0.9999	0.0000

Figure A.7: TD model parameter recovery Pearson correlation coefficients and p-values for list sampled parameter values

TD-Momentum Model - Multivariate Sampled Parameters - Recovery

α Coeff	α P-Val	γ_1 Coeff	γ_1 P-Val	γ_2 Coeff	γ_2 P-Val	f Coeff	f P-Val
0.9952	0.0000	0.6501	0.0000	0.4513	0.0010	0.6032	0.0000
0.9904	0.0000	0.8102	0.0000	0.5473	0.0000	0.4103	0.0031
0.9968	0.0000	0.5721	0.0000	0.4032	0.0037	0.3517	0.0123
0.9906	0.0000	0.8645	0.0000	0.4673	0.0006	0.2117	0.1400
0.9951	0.0000	0.7844	0.0000	0.4486	0.0011	0.3493	0.0129

Figure A.8: TD-Momentum model parameter recovery Pearson correlation coefficients and p-values for multivariate sampled parameter values

TD-Momentum Model - List Sampled Parameters - Recovery

α Coeff	α P-Val	γ_1 Coeff	γ_1 P-Val	γ_2 Coeff	γ_2 P-Val	f Coeff	f P-Val
0.9986	0.0000	0.8007	0.0000	0.3877	0.0054	0.4329	0.0017
0.9992	0.0000	0.8618	0.0000	0.5386	0.0001	0.2829	0.0465
0.9983	0.0000	0.6341	0.0000	0.6814	0.0000	0.5150	0.0001
0.9963	0.0000	0.6160	0.0000	0.6156	0.0000	0.4618	0.0007
0.9979	0.0000	0.6241	0.0000	0.4856	0.0004	0.4397	0.0014

Figure A.9: TD-Momentum model parameter recovery Pearson correlation coefficients and p-values for list sampled parameter values

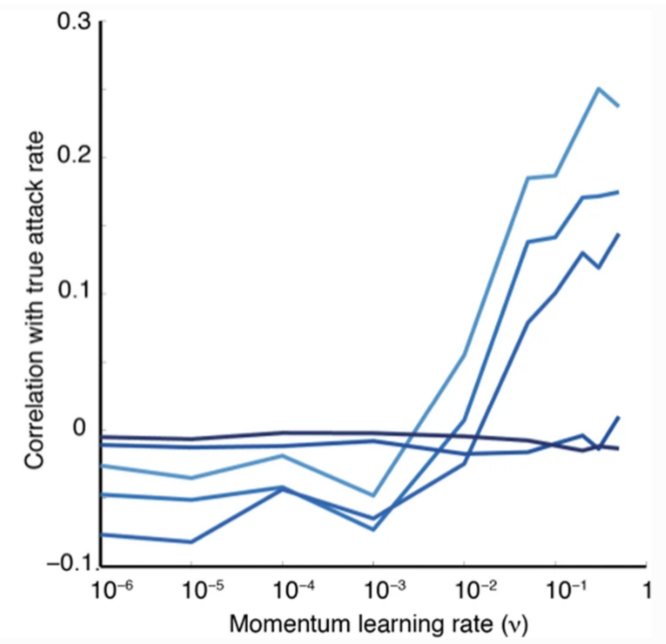

Figure A.10: Variations of auto-correlation in the AR process producing attack rates used to generate attack sequences shows how including momentum assists in extracting more information about the true attack rate (light blue indicates highest autoregression, dark blue indicates lowest autoregression) (figure by Kaye et al. [16])

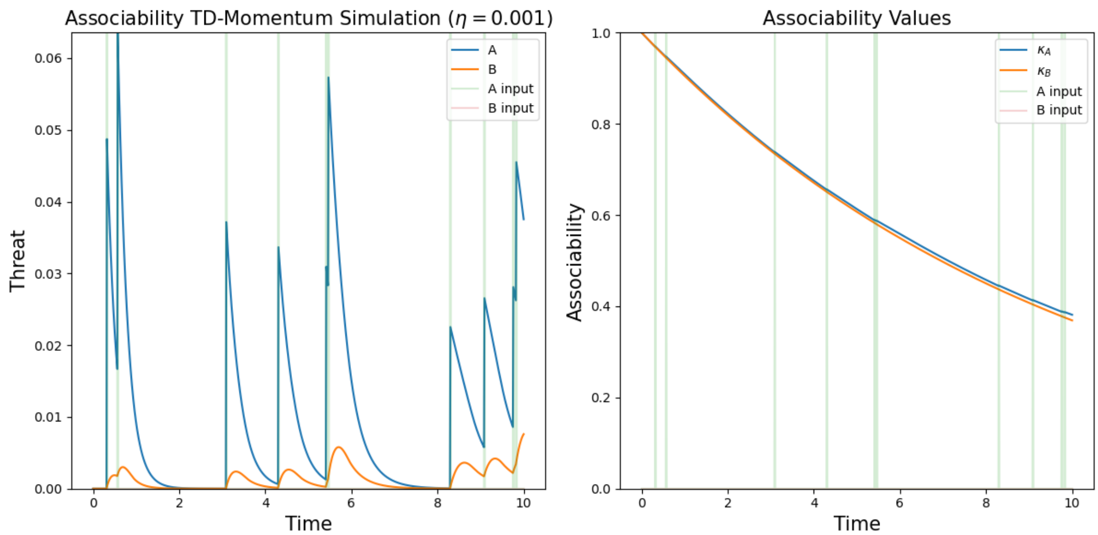

Figure A.11: Associability TD-Momentum model with $\eta = 0.001$. Very low values of η reduce this extended model to the original TD-Momentum model.

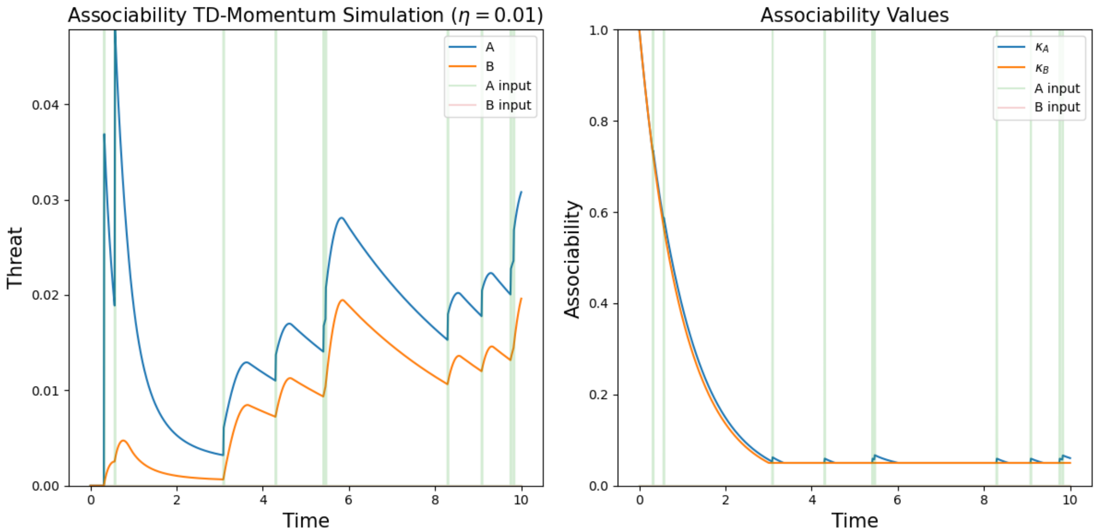

Figure A.12: Associability TD-Momentum model with $\eta = 0.01$. Increasing the value of η changes the shape of threat predictions acorss both contexts, reducing overall threat predictions and creating smoother predictions

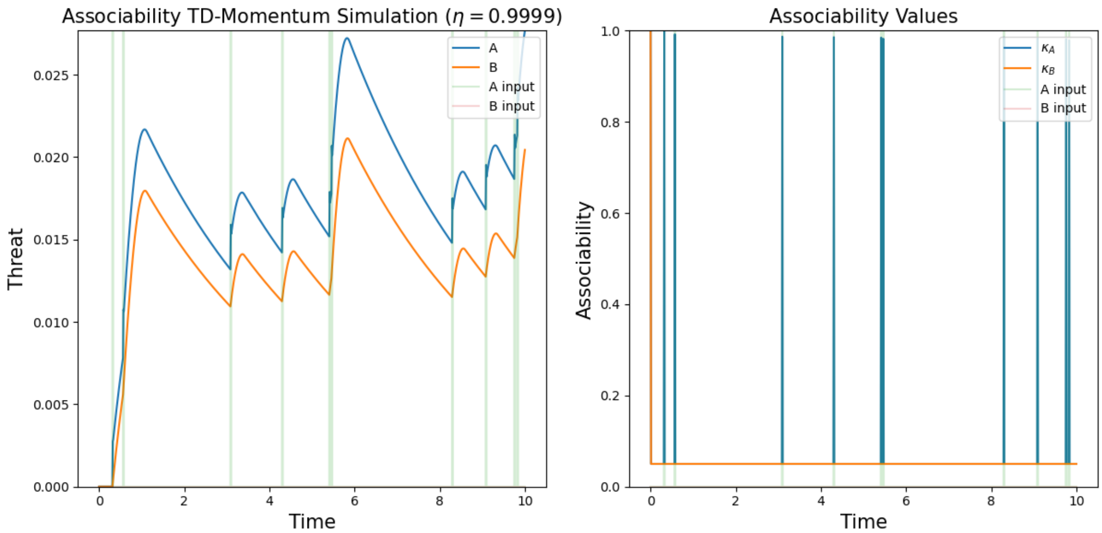

Figure A.13: Associability TD-Momentum model with $\eta = 0.9999$. Very high values of η generate threat predictions for both contexts that are very similar. Context B threat is only slightly below that of the trauma context A.

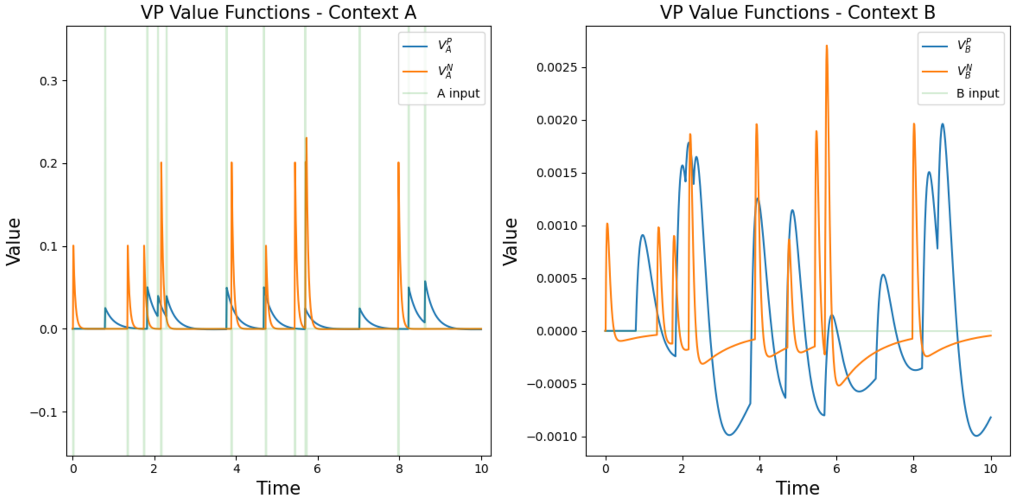

Figure A.14: Individual positive and negative VP value functions for each context (left - A, right - B). These are Eq. (4.8) and Eq. (4.9).

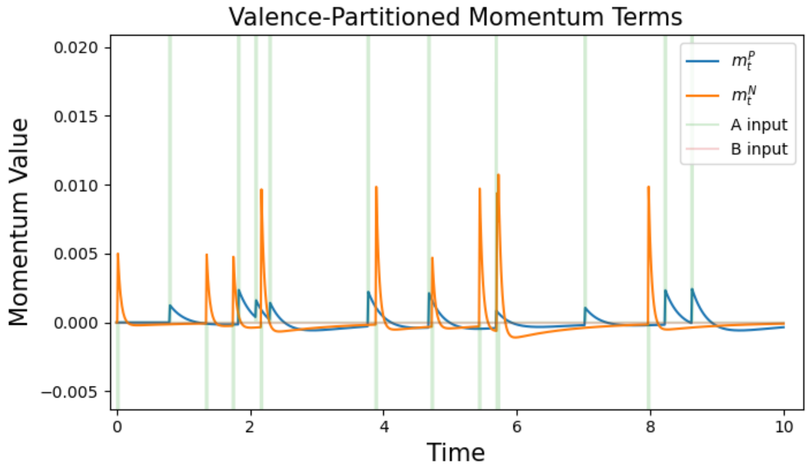

Figure A.15: Individual positively and negatively valenced momentum functions (blue - positive, orange - negative) relating to Eq. (4.10) and Eq. (4.11). Note how the negatively valenced momentum, containing the prediction errors of negative trauma outcomes, is larger and more influential than the positively valenced momentum of positive, non-traumatic outcomes.

Appendix B

Bayesian Model Python Code

B.1 Bayesian Model - Attack & Death Simulations

```python
import numpy as np
import pylab as pl
import scipy as sp
import scipy.stats as stats
import matplotlib.pyplot as plt
import matplotlib; matplotlib.use('Qt5Agg')
import emcee
from emcee import moves
import random
from scipy.ndimage import gaussian_filter
from matplotlib.colors import Normalize
from statsmodels.tsa.arima_process import ArmaProcess

def sim_sequences(pa, pd, num_time_steps):
    '''
    Simulates attack and death sequences based on binomial
    distribution, with constant pa and pd across timesteps.

    :param prob_attack: Probability of attack
    :param prob_death: Probability of death given attack happens
    :param num_time_steps: Total number of trials or timesteps where
     attack can occur

    :return: attacks, deaths which are arrays corresponding to
    sequences of attacks and deaths (0 is no attack/no death,
    1 is attack/death)
```

```
    '''
    # Simulating attack sequence where attacks are binomially
    distributed
    attacks = np.random.binomial(1, pa, num_time_steps)

    # Death sequence based on prob of death given attack
    deaths = np.random.binomial(1, pd, num_time_steps) * attacks

    return attacks, deaths
```

B.2 Bayesian Model - MCMC Sampler

```
### Bayesian model fitting procedure ###
# Now we move onto the Bayesian model fitting procedure.

# 'gwmcmc' function in matlab, here we use the emcee package in
    Python -
# Goodman and Weares Affine Invariant Markov chain Monte Carlo (MCMC
    ) Ensemble sampler

def ln_likelihood(params, attacks, deaths):
    '''
    Computes natural log of the joint likelihood

    :param params: Values for pa, pd
    :param attacks: Attack sequence
    :param deaths: Death sequence

    :return: Log-likelihood
    '''
    pa, pd = params

    # Compute number of attacks and deaths
    num_attacks = np.sum(attacks)
    num_deaths = np.sum(attacks * deaths)

    # Calculate log-likelihood
    ln_likelihood = num_attacks * np.log(pa * (1 - pd)) + (len(
    attacks) - num_attacks) * np.log(1 - pa) + num_deaths * np.log(pa
     * pd)

```

```python
    return ln_likelihood

def ln_prior(params):
    '''
    Computes natural log of prior

    :param params: Values for pa, pd

    :return: Log-prior
    '''
    pa, pd = params

    # Flat prior (uniform distribution), same for both pa and pd as both are probabilities
    p_min = 0   # Lower limit on range of values
    p_max = 1   # Upper limit on range of values

    # If pa, pd between correct range, 0-1, return prior = 1.0 (log-prior = 0), else 0 (log-prior = -inf)
    if p_min < pa < p_max and p_min < pd < p_max:
        return 0.0
    else:
        return -np.inf


def ln_posterior(params, attacks, deaths):
    '''
    Computes natural log of the joint posterior

    :param params: Values for pa, pd
    :param attacks: Attack sequence
    :param deaths: Death sequence

    :return: Log-posterior
    '''
    # Compute prior from function above
    ln_prior_val = ln_prior(params)

    # If function is NOT finite, return prob of 0 (log-prior = -inf)
    if not np.isfinite(ln_prior_val):
        return -np.inf

```

```python
    # Posterior = Prior * Likelihood (Log-Prior + Log-Likelihood)
    ln_posterior_val = ln_prior_val + ln_likelihood(params, attacks, deaths)

    return ln_posterior_val


def fit_bayesian_model(attacks, deaths, num_walkers, num_steps, burn_in, step_size):
    '''
    Fits Bayesian model using Affine Invariant MCMC Sampler from emcee package.

    :param attacks: Attack sequence
    :param deaths: Death sequence
    :param num_walkers: Number of walkers
    :param num_steps: Number of steps
    :param burn_in: Percentage of initial steps to remove as burn-in
    :param step_size: Step size to use in Strech_move parameter of sampler

    :return: Samples from chain created by sampler, i.e. progressive estimates for pa and pd
    '''
    # Define number of dims, i.e. pa and pd
    num_dimensions = 2

    # Initialise positions for the walkers
    initial_positions = np.random.rand(num_walkers, num_dimensions)

    # Set up the MCMC sampler with StretchMove - corresponds to "stepsize" = 2 in paper (matlab equivalent)
    stretch_move = moves.StretchMove(a=step_size)
    sampler = emcee.EnsembleSampler(num_walkers, num_dimensions, ln_posterior, args=(attacks, deaths),
                                    moves=[stretch_move])

    # Running burn-in phase, throwing these away so use _ as placeholder variables
    num_burn_in_steps = int(burn_in * num_steps)
    _, _, _ = sampler.run_mcmc(initial_positions, num_burn_in_steps, progress=True)
```

```python

    # Resetting the sampler
    sampler.reset()

    # Running production phase
    sampler.run_mcmc(None, num_steps, progress=True)

    # Retrieving MCMC samples from chain
    samples = sampler.get_chain(discard=num_burn_in_steps, flat=True)

    return samples
```

B.3 Bayesian Model - Posterior Plots

```python
def bayesian_plots(samples):
    '''
    Takes output of fit_bayesian_model and recreates plots for fig 2C in paper. Uses kernel density estimation (kde)
    to smoothen posterior distributions for pa and pd.

    :param samples: Output of fit_bayesian_model (estimations of pa and pd from MCMC Sampler)

    :return: Figure of 3 plots, p(attack), p(death|attack), p(attack) vs p(death|attack)
    '''
    # Create figure and axes
    fig, axes = plt.subplots(2, 2, sharex='col')

    # First plot - p(attack)
    kde1 = stats.gaussian_kde(samples[:, 0], bw_method=0.2)
    x1 = np.linspace(0, 0.025, 1000)
    y1 = kde1(x1)
    axes[0, 0].fill_between(x1, y1, color='skyblue', alpha=0.5)
    axes[0, 0].plot(x1, y1, label='p(attack) posterior', color='skyblue')
    axes[0, 0].set_xlim(0, 0.025)
    axes[0, 0].set_ylabel('p(attack)')
    axes[0, 0].legend(loc='upper right', fontsize='small')

```

```
    # Second plot - contour plot of p(attack) against p(death|attack)
    hist, xedges, yedges = np.histogram2d(samples[:, 0], samples[:, 1], bins=25)
    smooth_hist = gaussian_filter(hist.T, sigma=1)
    axes[1, 0].contourf(xedges[:-1], yedges[:-1], smooth_hist, cmap='Blues')
    #axes[1, 0].set_xlim(0, 0.025)
    # Determine the minimum and maximum values for the x-axis
    x_min = np.min(samples[:, 0])
    #x_max = np.max(samples[:, 0])

    # Set the x-axis limit based on the data range
    axes[1, 0].set_xlim(x_min, 0.025)

    axes[1, 0].set_ylabel('p(death|attack)')
    axes[1, 0].set_xlabel('p(attack)')

    # Third plot - p(death|attack)
    kde3 = stats.gaussian_kde(samples[:, 1], bw_method=0.5)
    x3 = np.linspace(0, 0.8, 1000) # changed from 1000 to 60
    y3 = kde3(x3)
    axes[1, 1].fill_between(x3, y3, color='skyblue', alpha=0.5)
    axes[1, 1].plot(x3, y3, label='p(death|attack) posterior', color='skyblue')
    axes[1, 1].set_xlabel('p(death|attack)')
    axes[1, 1].legend(loc='upper right', fontsize='small')

    # Remove top right empty plot
    fig.delaxes(axes[0, 1])

    # Adjust spacing between subplots
    plt.tight_layout()
    plt.show()


attacks, deaths = sim_sequences(0.01, 0.2, 700)
num_walkers = 30
num_steps = 60
burn_in = 0.3
step_size = 2

```

```
samples = fit_bayesian_model(attacks, deaths, num_walkers, num_steps
    , burn_in, step_size)
bayesian_plots(samples)
```

B.4 Bayesian Model - Variance in Threat Over Time

```
def variance_over_time(samples, colour, x_offset):
    '''
    Takes output of fit_bayesian_model, i.e. samples from MCMC
   sampler, and returns plot showing the variance in
    estimates over time. Groups use all available data up until the
   end of the respective group.

    :param samples: Output of fit_bayesian_model, samples from MCMC
   sampler
    :param colour: Colour scheme for plot
    :param x_offset: Use when comparing more than 1 sequence,
   separates plots for clarity


    :return: Plot showing variance in estimates over lifetime
    '''
    x = np.linspace(0, 1, np.size(samples[:, 0]))
    y = samples[:, 0]

    # Divide sample pa's into groups to plot with error bars
    num_groups = 8
    group_size = len(y) // num_groups

    # Empty sets to store group means and stds
    means = []
    stds = []

    # Compute mean and std dev for each group - each group will use
   info up until the last index of the group
    for i in range(1, num_groups + 1):
        end_idx = i * group_size
        group = y[:end_idx]
        means.append(np.mean(group))
        stds.append(np.std(group))

```

```python
    # Compute the x values for the last of each group
    group_last_x = x[group_size - 1::group_size]

    # Add the x-axis offset - only used for plotting two sequences together, e.g. the ELS and lifetime scenarios
    group_last_x += x_offset

    # Compute colors based on x-values
    cmap = plt.get_cmap(colour)
    norm = Normalize(vmin=np.min(x), vmax=np.max(x))
    colors = cmap(norm(group_last_x))

    # Plot the error bars with colormap
    for x_val, mean, std, color in zip(group_last_x, means, stds, colors):
        plt.errorbar(x_val, mean, yerr=std, fmt='o', color=color, alpha=0.5, label='95% CI')

    # Add colors to the points
    sc = plt.scatter(group_last_x, means, c=colors, cmap=cmap)

    # Add legend, labels, limits
    plt.ylim(0, 0.6)
    plt.xlabel('Time (fraction of lifetime)', fontsize=15)
    plt.ylabel('Estimated attack rate', fontsize=15)
    plt.title('Variance in $p_a$ decreasing over time', fontsize=15)
    plt.xticks(fontsize=14)
    plt.yticks(fontsize=14)
    plt.show()


samples = fit_bayesian_model(attacks, deaths, num_walkers, num_steps, burn_in, step_size)

variance_over_time(samples, colour = 'cool', x_offset=0)
```

B.5 Bayesian Model - AR Time Series

```python
def AR_PROCESS(c):
    '''
```

```
    Uses ArmaProcess function in tsa package to create a autocorrelated attack rate time series from an AR process.

    :param c: Chosen correlation coefficient to be used

    :return: Attack rate time series of 700 time steps.
    '''
    ar_coeff = np.array([1, -c])    # coeffs for AR process
    ma_coeff = np.array([1])        # equivalent to [1, 0], i.e. no moving average, only AR process
    AR_PROCESS = ArmaProcess(ar_coeff, ma_coeff).generate_sample(nsample=700, scale=0.01)    # scale param is noise std

    # Clip the AR process between 0 and 1
    clipped_AR_PROCESS = np.clip(AR_PROCESS, 0, 1)

    return clipped_AR_PROCESS
```

B.6 Bayesian Model - Posterior Comparisons for Various c Coefficients in AR Time Series

```
def average_density_many_ts(c, num_runs):
    '''
    Creates "average" density for many runs of time series. i.e. creates an "average" of the plots created in the
    bayesian_plots function over num_runs number of simulations of time series.

    :param c: Chosen correlation coefficient to be used
    :param num_runs: Number of runs to compute "average" density over

    :return: Outputs stacked densities over amount of numruns, plots "average" posterior in bayesian_plots
    '''
    # Create an empty array to store the results
    results = np.empty((num_runs, 1260, 2))     # (1260, 2) is shape of output from estimates_from_ts

    # Run the function multiple times
```

```
    for i in range(num_runs):
        results[i] = estimates_from_ts(c)

    # Calculate the average DENSITY of the results array
    # Correct way to take mean over densities, not the variables:
    stacked_values = np.column_stack((np.ravel(results[:, :, 0]), np
    .ravel(results[:, :, 1])))
    bayesian_plots(stacked_values)

    return stacked_values

average = average_density_many_ts(c = 0.7, num_runs = 10)

# Here, we turn to the supplementary material word doc, and recreate
     supp fig 2 and 3A

# Supp Fig 2C: Dispersion of estimated pa over time for varying
    correlation coefficients (c=0.7 - 0.99)
# Additional - Supp Fig 2C - Dispersion of estimated attack rate for
     varying correlation (c) values:

n = 10000
c_values = np.linspace(0.7, 0.999, 5)

variances = np.empty_like(c_values)
std_devs = np.empty_like(c_values)
densities = {}

for t, c in enumerate(c_values):
    dens = average_density_many_ts(c, n)[:,0]
    densities[t] = dens
    variances[t] = np.std(dens)
    std_devs[t] = np.var(dens)

# This provides us with a different view on sup fit 2C, here we've
    performed 10,000 sims per auto-correlation value
# We can see that the largest value c=0.999 does indeed provide us
    with the largest standard deviation and variance
# for estimated attack rate across all 10,000 runs. All other c vals
     are quite similar, indicating that c vals closer
# to 1 provide more variability in attack rate estimation.

```

```
# Average pa posteriors for 10,000 sims on each c value:

def posterior_comparison(c_vals, estimates):
    '''
    Takes output of average_density_many_ts function, i.e. 10,000 sims of various c values and creates pa posterior
    plot for comparison. Shows increased variability in estimation for larger c values.

    :param c_vals: List of associated c values
    :param estimates: Dictionary containing densities of each c value generated via average_density_many_ts

    :return: Plot comparing pa estimations
    '''
    colors = ['skyblue', 'green', 'orange', 'purple', 'yellow']  # List of colors for each plot
    labels = ['c = {}'.format(np.round(t, 3)) for t in c_vals]  # Labels for the legend

    for t in range(len(estimates)):
        # Posterior for p(attack)
        kde1 = stats.gaussian_kde(estimates[t], bw_method=0.2)
        x1 = np.linspace(0, 0.05, 1000)
        y1 = kde1(x1)
        plt.fill_between(x1, y1, color=colors[t], alpha=0.5)
        plt.plot(x1, y1, label=labels[t], color=colors[t])

    plt.xlim(0, 0.05)
    plt.xlabel('p(attack)')
    plt.title('Average densities of 10,000 simulations')
    plt.legend(loc='upper right', fontsize='small')

    plt.show()
posterior_comparison(c_values, densities)
```

B.7 Bayesian Model - ELS Example

```
def sim_scenarios(pd, num_time_steps, num_attacks):
    '''
```

```
    Simulates attack and death sequences for two scenarios,
    random_lifetime and random_els. Limiting the number
    of attacks to be equal for both scenarios = num_attacks.

    :param prob_attack: Probability of attack
    :param prob_death: Probability of death given attack happens
    :param num_time_steps: Total number of trials or timesteps where
     attack can occur

    :return: attacks, deaths which are arrays corresponding to
    sequences of attacks and deaths (0 is no attack/no death,
    1 is attack/death)
    '''
    # Lifetime Random Scenario:
    rand_life_attacks = np.zeros(num_time_steps)
    life_indices = np.random.choice(num_time_steps, num_attacks,
    replace=False)

    # Input attacks (1s) in random chosen timesteps
    rand_life_attacks[life_indices] = 1
    rand_life_deaths = np.random.binomial(1, pd, num_time_steps) *
    rand_life_attacks


    # ELS Random Scenario:
    rand_els_attacks = np.zeros(num_time_steps)
    half_num_time_steps = int((num_time_steps)/2)
    els_indices = np.random.choice(half_num_time_steps, num_attacks,
     replace=False)
    #els_indices = np.random.randint(0, half_num_time_steps, size=
    num_attacks)

    # Input attacks (1s) in random chosen timesteps
    rand_els_attacks[els_indices] = 1
    rand_els_deaths = np.random.binomial(1, pd, num_time_steps) *
    rand_els_attacks

    return rand_life_attacks, rand_life_deaths, rand_els_attacks,
    rand_els_deaths

# Now put this into fit_bayesian_model(attacks, deaths, num_walkers,
     num_steps, burn_in, step_size)
```

```python
scenarios = sim_scenarios(pd = 0.2, num_time_steps = 700,
    num_attacks = 5)

life_attack_seq, life_death_seq = scenarios[0], scenarios[1]
els_attack_seq, els_death_seq = scenarios[2], scenarios[3]

num_walkers = 30
num_steps = 60
burn_in = 0.3
step_size = 2

life_bayesian_fit = fit_bayesian_model(life_attack_seq,
    life_death_seq, num_walkers, num_steps, burn_in, step_size)
els_bayesian_fit = fit_bayesian_model(els_attack_seq, els_death_seq,
     num_walkers, num_steps, burn_in, step_size)

# Plug these into variance_over_time(samples) to get FIG 3B
life_variance = variance_over_time(life_bayesian_fit, colour = '
    Blues', x_offset = 0.015)
els_variance = variance_over_time(els_bayesian_fit, colour = 'Reds',
     x_offset = 0)
```

B.8 Bayesian Model - Clustered ELS Attacks

```python
def els_clustered(pd, num_time_steps, num_attacks, sims):
    '''
    Simulates attack and death sequences for two scenarios, attacks
    are clustered randomly or uniformly.
    These are then put into variance over time plot to show how
    threat prediction varies. Results are averaged across
    "sims" number of different runs (different lifetime and els
    attack sequences).

    :param prob_attack: Probability of attack
    :param prob_death: Probability of death given attack happens
    :param num_time_steps: Total number of trials or timesteps where
     attack can occur

    :return: attacks, deaths which are arrays corresponding to
    sequences of attacks and deaths (0 is no attack/no death,
    1 is attack/death)
```

```
'''
life_sims = np.zeros((sims, 1260, 2))   # burnin: 0.1->1620,
0.2->1440, 0.3->1260
els_sims = np.zeros((sims, 1260, 2))

for s in range(sims):
    # Lifetime Random Scenario:
    rand_life_attacks = np.zeros(num_time_steps)
    indices = np.random.choice(num_time_steps, num_attacks,
replace=False)
    print(indices)
    # Input attacks (1s) in random chosen timesteps
    rand_life_attacks[indices] = 1
    rand_life_deaths = np.random.binomial(1, pd, num_time_steps)
 * rand_life_attacks


    # ELS clustered Scenario: random clusters of 3 attacks
simultaneously (over 3 timesteps)
    clus_els_attacks = np.zeros(num_time_steps)
    half_num_time_steps = int(num_time_steps/2)
    #indices = np.random.choice(half_num_time_steps, num_attacks
, replace=False)  # RANDOM SPACING

    indices = np.linspace(0, half_num_time_steps, num_attacks,
dtype=int)   # EQUAL SPACING
    print(indices)

    # Input attacks (1s) in random chosen timesteps
    clus_els_attacks[indices] = 1
    next_ind = [x+1 for x in indices]
    print(next_ind)
    clus_els_attacks[next_ind] = 1
    next_ind = [x+1 for x in next_ind]
    print(next_ind)
    clus_els_attacks[next_ind] = 1

    #print(next_ind)

    rand_els_deaths = np.random.binomial(1, pd, num_time_steps)
* clus_els_attacks
    #print(np.sum(clus_els_attacks))
```

```python
        #print(np.sum(rand_life_attacks))

        num_walkers = 30
        num_steps = 60
        burn_in = 0.3
        step_size = 2

        life_bayesian_fit = fit_bayesian_model(rand_life_attacks,
    rand_life_deaths, num_walkers, num_steps, burn_in, step_size)
        els_bayesian_fit = fit_bayesian_model(clus_els_attacks,
    rand_els_deaths, num_walkers, num_steps, burn_in, step_size)

        life_sims[s,:,:] = life_bayesian_fit
        els_sims[s,:,:] = els_bayesian_fit

    # Average over sims for each step
    life_means = np.mean(life_sims, axis=0)
    els_means = np.mean(els_sims, axis=0)
    #print(np.shape(life_sims))
    #print(np.shape(els_sims))
    # Generating plots for avg variance over time
    life_variance = variance_over_time(life_means, colour='Blues',
    x_offset=0.015)
    els_variance = variance_over_time(els_means, colour='Reds',
    x_offset=0)

    return life_sims, els_sims

life_sims, els_sims = els_clustered(pd = 0.01, num_time_steps = 700,
     num_attacks = 5, sims=1)
```

Appendix C

TD Model Python Code

C.1 Preprocessing SEFL Data

```
import numpy as np
from sklearn.preprocessing import MinMaxScaler
from scipy.optimize import minimize
from scipy.optimize import differential_evolution
import pandas as pd
import scipy.stats


import scipy.io as sio
mat_contents = sio.loadmat('Supp_Mat')

### EXTRACTING ALL DETAILS ###

animals = mat_contents["sefl_behavior_day1"]["animal"].reshape(-1)
stress_type_index = mat_contents["sefl_behavior_day1"]["stress"].
    reshape(-1)
day1_freezing_ts = mat_contents["sefl_behavior_day1"]["
    freezing_time_series"].reshape(-1)
day1_smoothed_freezing = mat_contents["sefl_behavior_day1"]["
    smoothed_freezing"].reshape(-1)
day6_animal_index = mat_contents["sefl_behavior_day1"]["day6_index"
    ].reshape(-1)
day7_animal_index = mat_contents["sefl_behavior_day1"]["day7_index"
    ].reshape(-1)


```

```python
# Storing all details:
details_df = pd.DataFrame({
    "animal": animals,
    "stress": stress_type_index,
    "day1_freezing_time_series": day1_freezing_ts,
    "day1_smoothed_freezing": day1_smoothed_freezing,
    "day6_index": day6_animal_index,
    "day7_index": day7_animal_index
})

# Remove invalid rows:
# Mouse 3 removed - not in day7 file - G11
# Mouse 9 removed - no data - G17
# Mouse 25 removed - no data in day6 file - G33
# Mouse 28 removed - too many shocks - G36
# Mouse 35 removed - too many shocks - G9

indexes_to_remove = [2, 8, 24, 27, 34]
details_df = details_df.drop(indexes_to_remove)
# Rest row indexes are removing
details_df = details_df.reset_index(drop=True)

# Adjusting df to be nicer to work with:
for i in range(len(details_df["animal"])):
    details_df["animal"][i] = details_df["animal"][i][0]
    details_df["stress"][i] = details_df["stress"][i][0][0]
    details_df["day6_index"][i] = details_df["day6_index"][i][0][0]
    details_df["day7_index"][i] = details_df["day7_index"][i][0][0]
    details_df["day1_freezing_time_series"][i] = details_df["
    day1_freezing_time_series"][i].tolist()
    details_df["day1_smoothed_freezing"][i] = details_df["
    day1_smoothed_freezing"][i].tolist()

# Changing lower case g to upper case to match day6 and day7 animal
    names:
details_df["animal"] = details_df["animal"].str.replace(r'^g', 'G')

# Day 1 Shocks:
day1_shock_times = mat_contents["sefl_behavior_day1"]["shock_times"
    ].reshape(-1)
# Removing rows of relevant indexes:
day1_shock_times = np.delete(day1_shock_times, indexes_to_remove)
```

```


# Now creating the actual attack sequences (0s and 1s every timestep
    ) - 33 mice
day1_attack_sequences = np.zeros((33, 162000))

for m in range(33):
    times = day1_shock_times[m]
    # Adjust from 1-based indexing
    indexes = times-1

    # Add attack (1) at each index accordingly
    for i in indexes:
        day1_attack_sequences[m, int(i)] = 1

# Convert each row of day1_attack_sequences into a list
attack_sequences_list = [row.tolist() for row in
    day1_attack_sequences]

# Create the DataFrame with each list as a row
day1_attack_sequences = pd.DataFrame({"attack_sequence":
    attack_sequences_list})
# Sanity check: check for several mice that attacks are in right
    place:
#np.where(np.array(day1_attack_sequences["attack_sequence"][0]) ==
    1) # Mouse G1


# Now attach this to details_df
details_df.insert(2, "day1_attack_sequences", day1_attack_sequences)
#details_df["day1_attack_sequences"] = day1_attack_sequences



# Extracting day6 and day7 freezing time series and smoothed
    freezing for these mice:
day6_animal = mat_contents["sefl_behavior_day6"]["animal"].reshape
    (-1)
day6_freezing_ts = mat_contents["sefl_behavior_day6"]["
    freezing_time_series"].reshape(-1)
day6_smoothed_freezing = mat_contents["sefl_behavior_day6"]["
    smoothed_freezing"].reshape(-1)
```

```

day6_details_df = pd.DataFrame({
    "animal": day6_animal,
    "day6_freezing_time_series": day6_freezing_ts,
    "day6_smoothed_freezing": day6_smoothed_freezing
})

# Adjusting df to be nicer to work with:
for i in range(len(day6_details_df["animal"])):
    day6_details_df["animal"][i] = day6_details_df["animal"][i
    ][0][0][0]
    day6_details_df["day6_freezing_time_series"][i] =
    day6_details_df["day6_freezing_time_series"][i].tolist()
    day6_details_df["day6_smoothed_freezing"][i] = day6_details_df["
    day6_smoothed_freezing"][i].tolist()

# Day7
day7_animal = mat_contents["sefl_behavior_day7"]["animal"].reshape
    (-1)
day7_freezing_ts = mat_contents["sefl_behavior_day7"]["
    freezing_time_series"].reshape(-1)
day7_smoothed_freezing = mat_contents["sefl_behavior_day7"]["
    smoothed_freezing"].reshape(-1)

day7_details_df = pd.DataFrame({
    "animal": day7_animal,
    "day7_freezing_time_series": day7_freezing_ts,
    "day7_smoothed_freezing": day7_smoothed_freezing
})

# Adjusting df to be nicer to work with:
for i in range(len(day7_details_df["animal"])):
    day7_details_df["animal"][i] = day7_details_df["animal"][i
    ][0][0][0]
    day7_details_df["day7_freezing_time_series"][i] =
    day7_details_df["day7_freezing_time_series"][i].tolist()
    day7_details_df["day7_smoothed_freezing"][i] = day7_details_df["
    day7_smoothed_freezing"][i].tolist()


# Now joining day6 and day7 details into main details_df:
# Merging day6_details_df onto details_df based on "animal" column
```

```
details_df = pd.merge(details_df, day6_details_df[["animal", "day6_freezing_time_series"]], on="animal", how="left")
details_df = pd.merge(details_df, day6_details_df[["animal", "day6_smoothed_freezing"]], on="animal", how="left")


# Merging day7_details_df onto details_df based on "animal" column
details_df = pd.merge(details_df, day7_details_df[["animal", "day7_freezing_time_series"]], on="animal", how="left")
details_df = pd.merge(details_df, day7_details_df[["animal", "day7_smoothed_freezing"]], on="animal", how="left")

# Now details_df has all details I require, I can fit data for all days for the correct mice:
stressed_mice_details = details_df[details_df["stress"] == 1].copy()
stressed_mice_details = stressed_mice_details.reset_index(drop=True)

unstressed_mice_details = details_df[details_df["stress"] == 0].copy()
unstressed_mice_details = unstressed_mice_details.reset_index(drop=True)
```

C.2 TD Model - Smoothed Freezing & Fitting SEFL Data

```
def TD_model_init(init, u, alpha, gamma1):
    '''
    Evaluates TD model threat at each timestep

    :param u: Input (sequence of unconditioned stimuli, i.e. sequences of footshocks, 0s and 1s), for different
    contexts, i.e A and B (day 1 is context A, day 6 and 7 is context B).
    :param alpha: Learning rate
    :param gamma1: Decay rate for threat

    :return: TD model threat estimation levels over all timesteps
    '''
    T = np.zeros_like(u)
    T[0] = init
    for t in range(1, len(T)):
        T[t] = T[t-1] + alpha * (u[t] - gamma1 * T[t-1])
```

```python

    # Scaling between 0.1 and 0.9
    scaler = MinMaxScaler(feature_range=(0.1, 0.9))
    T = scaler.fit_transform(T.reshape(-1, 1)).flatten()

    return T

def NLL_td(params, init, stimuli, threats):
    '''
    Calculates NLL for one set of parameters given a sequence of attacks and threat predictions for the TD model.
    The likelihood calculation uses the PMF of a Bernoulli distribution to calculate the log probability of observing
    the given threat probability "threat" based on the current value of T.

    :param params: Array of parameters alpha and gamma1
    :param init: initialisation for threat sequence (eg end of day 1 context A for day 6 context A
    :param stimuli: Sequence of shocks (attacks)
    :param threats: Sequence of threat predictions

    :return: Negative Log-Likelihood of one set of parameters.
    '''
    alpha, gamma1 = params
    log_likelihood = 0.0

    T = TD_model_init(init = init, u = stimuli, alpha = alpha, gamma1 = gamma1)

    eps = 1e-10
    T = np.clip(T, 0+eps, 1-eps)

    for t in range(len(stimuli)):
        # Assuming threat distribution is Bernoulli - this appears to work as needed
        # Uses probability mass function (PMF) of a Bernoulli distribution: considers threats as probabilities, i.e.
        # each 'threat' represents the prob of observing a shock.
        log_likelihood += threats[t] * np.log(T[t]) + (1 - threats[t]) * np.log(1 - T[t])

```

```
    nll = -log_likelihood

    return nll

def smoothing(raw_time_series, window_size):
    smoothed_time_series = []
    half_window = window_size // 2

    # Pad the time series
    padded_time_series = np.pad(raw_time_series, (half_window,
   half_window), mode='edge')

    for i in range(len(raw_time_series)):
        window_values = padded_time_series[i : i + window_size]
        smoothed_value = np.mean(window_values)  # Average of window
        smoothed_time_series.append(smoothed_value)

    return np.array(smoothed_time_series)





### FITTING CONTEXT A DAY 1 ### STRESSED

# Storing MANUAL smoothed freezing data for each mouse
day1_stressed_smoothed_freezing = np.zeros((len(
   stressed_mice_details), len(stressed_mice_details["
   day1_freezing_time_series"][0])))

# Applying smoothing function to each mouse freezing time series
for i in range(len(stressed_mice_details)):
    raw_time_series = stressed_mice_details["
   day1_freezing_time_series"][i]
    smoothed_time_series = smoothing(raw_time_series, window_size =
   15)
    day1_stressed_smoothed_freezing[i, :] = smoothed_time_series

day1_stressed_freezing = day1_stressed_smoothed_freezing
       # (15, 162000) use as prediction input
day1_stressed_attacks = stressed_mice_details["day1_attack_sequences
   "]
```

```
day1_stressed_attacks = np.array(day1_stressed_attacks.to_list())       # (15, 162000) use as stimuli input
initial = 0

initial_vals = np.array((0.5, 0.95))
bounds = [(0.05,0.9), (0.9,1)]

# Minimization for MICE
for m in range(0, len(stressed_mice_details)):
    # Compute minimization for each participant
    result = differential_evolution(NLL_td, x0=initial_vals, bounds=bounds,
                                args=(initial, day1_stressed_attacks[m:m + 1].reshape(-1), day1_stressed_freezing[m:m + 1].reshape(-1)),
                                    polish=True)

    print(f'n_iter: {result.nit} - success: {result.success} - nll {result.fun}')

    # Store in results dataframe:
    TD_stressed_results["Day 1 Context A NLL"][m] = result.fun
    alpha, gamma1 = result.x
    TD_stressed_results["Day 1 Context A Params"][m] = [alpha, gamma1]



### FITTING CONTEXT B DAY 1 ### STRESSED
day1_stressed_attacks_B = np.zeros_like(day1_stressed_attacks)     # (15, 162000) use as stimuli input
initial = 0

# Minimization for MICE
for m in range(0, len(stressed_mice_details)):
    # Compute TD threat fit from parameters for this day, compute NLL
    a, g = TD_stressed_results["Day 1 Context A Params"][m]
    day1_stressed_freezing_B = TD_model_init(init=initial, u=day1_stressed_attacks_B[m,:], alpha= a, gamma1=g)  # (15, 162000) use as threat input

    params = [a, g]
```

```
    nll = NLL_td(params=params, init=initial, stimuli=
    day1_stressed_attacks_B[m,:], threats=day1_stressed_freezing_B)
    # Store in results dataframe:
    TD_stressed_results["Day 1 Context B NLL"][m] = nll





### FITTING CONTEXT B DAY 6 ### STRESSED
# Storing MANUAL smoothed freezing data for each mouse
day6_stressed_smoothed_freezing = np.zeros((len(
    stressed_mice_details), len(stressed_mice_details["
    day6_freezing_time_series"][0])))

# Applying smoothing function to each mouse freezing time series
for i in range(len(stressed_mice_details)):
    raw_time_series = stressed_mice_details["
    day6_freezing_time_series"][i]
    smoothed_time_series = smoothing(raw_time_series, window_size =
    15)
    day6_stressed_smoothed_freezing[i, :] = smoothed_time_series

day6_stressed_freezing = day6_stressed_smoothed_freezing
        # (15, 18000) use as prediction input
day6_stressed_attacks_B = np.zeros_like(day6_stressed_freezing)
      # (15, 18000) use as stimuli input

for m in range(len(day6_stressed_attacks_B)):
    day6_stressed_attacks_B[m,8999] = 1
initial = 0

initial_vals = np.array((0.5, 0.95))
bounds = [(0.05,0.9), (0.9,1)]
# Minimization for MICE
for m in range(0, len(stressed_mice_details)):
    # Compute minimization for each participant
    result = differential_evolution(NLL_td, x0=initial_vals, bounds=
    bounds,
                                    args=(initial, day6_stressed_attacks_B[m:
    m + 1].reshape(-1), day6_stressed_freezing[m:m + 1].reshape(-1)),
                                                  polish=True)
```

```

    print(f'n_iter: {result.nit} - success: {result.success} - nll {
    result.fun}')

    # Store in results dataframe:
    TD_stressed_results["Day 6 Context B NLL"][m] = result.fun
    alpha, gamma1 = result.x
    TD_stressed_results["Day 6 Context B Params"][m] = [alpha,
    gamma1]




### FITTING CONTEXT A DAY 6 ### STRESSED
day6_stressed_attacks_A = np.zeros_like(day6_stressed_attacks_B)
      # (15, 162000) use as stimuli input

# Minimization for MICE
for m in range(0, len(stressed_mice_details)):
    # Finding initial for each mouse
    a_prev, g_prev = TD_stressed_results["Day 1 Context A Params"][m
    ]
    initial = TD_model_init(init=0, u=day1_stressed_attacks[m,:],
    alpha=a_prev, gamma1=g_prev)[-1]

    # Compute TD threat fit from parameters for this day, compute
    NLL
    a, g = TD_stressed_results["Day 6 Context B Params"][m]
    day6_stressed_freezing_A = TD_model_init(init=initial, u=
    day6_stressed_attacks_A[m,:], alpha= a, gamma1=g)  # (15, 162000)
     use as threat input

    params = [a, g]
    nll = NLL_td(params=params, init=initial, stimuli=
    day6_stressed_attacks_A[m,:], threats=day6_stressed_freezing_A)
    # Store in results dataframe:
    TD_stressed_results["Day 6 Context A NLL"][m] = nll
```

```python

### FITTING CONTEXT B DAY 7 ### STRESSED
# Storing MANUAL smoothed freezing data for each mouse
day7_stressed_smoothed_freezing = np.zeros((len(stressed_mice_details), len(stressed_mice_details["day7_freezing_time_series"][0])))

# Applying smoothing function to each mouse freezing time series
for i in range(len(stressed_mice_details)):
    raw_time_series = stressed_mice_details["day7_freezing_time_series"][i]
    smoothed_time_series = smoothing(raw_time_series, window_size = 15)
    day7_stressed_smoothed_freezing[i, :] = smoothed_time_series


day7_stressed_freezing = day7_stressed_smoothed_freezing       # (15, 18000) use as prediction input
day7_stressed_attacks_B = np.zeros_like(day7_stressed_freezing)          # (15, 18000) use as stimuli input

initial_vals = np.array((0.5, 0.95))
bounds = [(0.05,0.9), (0.9,1)]
# Minimization for MICE
for m in range(0, len(stressed_mice_details)):
    # Getting initial for each mouse:
    a_prev_prev, g_prev_prev = TD_stressed_results["Day 1 Context A Params"][m]
    initial_prev = TD_model_init(init=0, u=day1_stressed_attacks_B[m,:], alpha=a_prev_prev, gamma1=g_prev_prev)[-1]

    a, g = TD_stressed_results["Day 6 Context B Params"][m]
    initial = TD_model_init(init=initial_prev, u=day6_stressed_attacks_B[m,:], alpha=a, gamma1=g)[-1]

    # Compute minimization for each participant
    result = differential_evolution(NLL_td, x0=initial_vals, bounds=bounds,
                                    args=(initial, day7_stressed_attacks_B[m:m + 1].reshape(-1), day7_stressed_freezing[m:m + 1].reshape(-1)),
                                    polish=True)

```

```
    print(f'n_iter: {result.nit} - success: {result.success} - nll {result.fun}')

    # Store in results dataframe:
    TD_stressed_results["Day 7 Context B NLL"][m] = result.fun
    alpha, gamma1 = result.x
    TD_stressed_results["Day 7 Context A Params"][m] = [alpha, gamma1]




### FITTING CONTEXT A DAY 7 ### STRESSED
day7_stressed_attacks_A = np.zeros_like(day7_stressed_attacks_B)        # (15, 18000) use as stimuli input

# Minimization for MICE
for m in range(0, len(stressed_mice_details)):
    # Finding initial for each mouse
    a_prev_prev, g_prev_prev = TD_stressed_results["Day 1 Context A Params"][m]
    initial_prev = TD_model_init(init=0, u=day1_stressed_attacks[m,:], alpha=a_prev_prev, gamma1=g_prev_prev)[-1]

    a_prev, g_prev = TD_stressed_results["Day 6 Context B Params"][m]
    initial = TD_model_init(init=initial_prev, u=day6_stressed_attacks_A[m,:], alpha=a_prev, gamma1=g_prev)[-1]

    # Compute TD threat fit from parameters for this day, compute NLL
    a, g = TD_stressed_results["Day 7 Context A Params"][m]
    day7_stressed_freezing_A = TD_model_init(init=initial, u=day7_stressed_attacks_A[m,:], alpha= a, gamma1=g)  # (15, 162000) use as threat input

    params = [a, g]
    nll = NLL_td(params=params, init=initial, stimuli=day7_stressed_attacks_A[m,:], threats=day7_stressed_freezing_A)
    # Store in results dataframe:
    TD_stressed_results["Day 7 Context A NLL"][m] = nll

```

```
### NOW FOR UNSTRESSED ###

### FITTING CONTEXT A DAY 1 ### UNSTRESSED

# Storing MANUAL smoothed freezing data for each mouse
day1_unstressed_smoothed_freezing = np.zeros((len(
    unstressed_mice_details), len(unstressed_mice_details["
    day1_freezing_time_series"][0])))

# Applying smoothing function to each mouse freezing time series
for i in range(len(unstressed_mice_details)):
    raw_time_series = unstressed_mice_details["
    day1_freezing_time_series"][i]
    smoothed_time_series = smoothing(raw_time_series, window_size =
    15)
    day1_unstressed_smoothed_freezing[i, :] = smoothed_time_series

day1_unstressed_freezing = day1_unstressed_smoothed_freezing
            # (15, 162000) use as prediction input
day1_unstressed_attacks = np.zeros_like(day1_unstressed_freezing)
initial = 0

initial_vals = np.array((0.5, 0.95))
bounds = [(0.05,0.9), (0.9,1)]

# Minimization for MICE
for m in range(0, len(unstressed_mice_details)):
    # Compute minimization for each participant
    result = differential_evolution(NLL_td, x0=initial_vals, bounds=
    bounds,
                                    args=(initial, day1_unstressed_attacks[m:
    m + 1].reshape(-1), day1_unstressed_freezing[m:m + 1].reshape(-1)
```

```
    ),
                                               polish=True)

    print(f'n_iter: {result.nit} - success: {result.success} - nll {
    result.fun}')

    # Store in results dataframe:
    TD_unstressed_results["Day 1 Context A NLL"][m] = result.fun
    alpha, gamma1 = result.x
    TD_unstressed_results["Day 1 Context A Params"][m] = [alpha,
    gamma1]



### FITTING CONTEXT B DAY 1 ### UNSTRESSED
day1_unstressed_attacks_B = np.zeros_like(day1_unstressed_attacks)
         # (15, 162000) use as stimuli input
initial = 0

# Minimization for MICE
for m in range(0, len(unstressed_mice_details)):
    # Compute TD threat fit from parameters for this day, compute
    NLL
    a, g = TD_unstressed_results["Day 1 Context A Params"][m]
    day1_unstressed_freezing_B = TD_model_init(init=initial, u=
    day1_unstressed_attacks_B[m,:], alpha= a, gamma1=g)  # (15,
    162000) use as threat input

    params = [a, g]
    nll = NLL_td(params=params, init=initial, stimuli=
    day1_unstressed_attacks_B[m,:], threats=
    day1_unstressed_freezing_B)
    # Store in results dataframe:
    TD_unstressed_results["Day 1 Context B NLL"][m] = nll





### FITTING CONTEXT B DAY 6 ### UNSTRESSED
# Storing MANUAL smoothed freezing data for each mouse
day6_unstressed_smoothed_freezing = np.zeros((len(
```

```
    unstressed_mice_details), len(unstressed_mice_details["
    day6_freezing_time_series"][0])))

# Applying smoothing function to each mouse freezing time series
for i in range(len(unstressed_mice_details)):
    raw_time_series = unstressed_mice_details["
    day6_freezing_time_series"][i]
    smoothed_time_series = smoothing(raw_time_series, window_size =
    15)
    day6_unstressed_smoothed_freezing[i, :] = smoothed_time_series

day6_unstressed_freezing = day6_unstressed_smoothed_freezing
            # (15, 18000) use as prediction input
day6_unstressed_attacks_B = np.zeros_like(day6_unstressed_freezing)
          # (15, 18000) use as stimuli input

for m in range(len(day6_unstressed_attacks_B)):
    day6_unstressed_attacks_B[m,8999] = 1
initial = 0

initial_vals = np.array((0.5, 0.95))
bounds = [(0.05,0.9), (0.9,1)]
# Minimization for MICE
for m in range(0, len(unstressed_mice_details)):
    # Compute minimization for each participant
    result = differential_evolution(NLL_td, x0=initial_vals, bounds=
    bounds,
                                    args=(initial, day6_unstressed_attacks_B[
    m:m + 1].reshape(-1), day6_unstressed_freezing[m:m + 1].reshape
    (-1)),
                                             polish=True)

    print(f'n_iter: {result.nit} - success: {result.success} - nll {
    result.fun}')

    # Store in results dataframe:
    TD_unstressed_results["Day 6 Context B NLL"][m] = result.fun
    alpha, gamma1 = result.x
    TD_unstressed_results["Day 6 Context B Params"][m] = [alpha,
    gamma1]


```

```python


### FITTING CONTEXT A DAY 6 ### UNSTRESSED
day6_unstressed_attacks_A = np.zeros_like(day6_unstressed_attacks_B)          # (15, 162000) use as stimuli input

# Minimization for MICE
for m in range(0, len(unstressed_mice_details)):
    # Finding initial for each mouse
    a_prev, g_prev = TD_unstressed_results["Day 1 Context A Params"][m]
    initial = TD_model_init(init=0, u=day1_unstressed_attacks[m,:], alpha=a_prev, gamma1=g_prev)[-1]

    # Compute TD threat fit from parameters for this day, compute NLL
    a, g = TD_unstressed_results["Day 6 Context B Params"][m]
    day6_unstressed_freezing_A = TD_model_init(init=initial, u=day6_unstressed_attacks_A[m,:], alpha= a, gamma1=g)  # (15, 162000) use as threat input

    params = [a, g]
    nll = NLL_td(params=params, init=initial, stimuli=day6_unstressed_attacks_A[m,:], threats=day6_unstressed_freezing_A)
    # Store in results dataframe:
    TD_unstressed_results["Day 6 Context A NLL"][m] = nll






### FITTING CONTEXT B DAY 7 ### UNSTRESSED
# Storing MANUAL smoothed freezing data for each mouse
day7_unstressed_smoothed_freezing = np.zeros((len(unstressed_mice_details), len(unstressed_mice_details["day7_freezing_time_series"][0])))

# Applying smoothing function to each mouse freezing time series
for i in range(len(unstressed_mice_details)):
    raw_time_series = unstressed_mice_details["
```

```
day7_freezing_time_series"][i]
    smoothed_time_series = smoothing(raw_time_series, window_size = 15)
    day7_unstressed_smoothed_freezing[i, :] = smoothed_time_series


day7_unstressed_freezing = day7_unstressed_smoothed_freezing            # (15, 18000) use as prediction input
day7_unstressed_attacks_B = np.zeros_like(day7_unstressed_freezing)                # (15, 18000) use as stimuli input

initial_vals = np.array((0.5, 0.95))
bounds = [(0.05,0.9), (0.9,1)]
# Minimization for MICE
for m in range(0, len(unstressed_mice_details)):
    # Getting initial for each mouse:
    a_prev_prev, g_prev_prev = TD_unstressed_results["Day 1 Context A Params"][m]
    initial_prev = TD_model_init(init=0, u=day1_unstressed_attacks_B[m,:], alpha=a_prev_prev, gamma1=g_prev_prev)[-1]

    a, g = TD_unstressed_results["Day 6 Context B Params"][m]
    initial = TD_model_init(init=initial_prev, u=day6_unstressed_attacks_B[m,:], alpha=a, gamma1=g)[-1]

    # Compute minimization for each participant
    result = differential_evolution(NLL_td, x0=initial_vals, bounds=bounds,
                                    args=(initial, day7_unstressed_attacks_B[m:m + 1].reshape(-1), day7_unstressed_freezing[m:m + 1].reshape(-1)),
                                                 polish=True)

    print(f'n_iter: {result.nit} - success: {result.success} - nll {result.fun}')

    # Store in results dataframe:
    TD_unstressed_results["Day 7 Context B NLL"][m] = result.fun
    alpha, gamma1 = result.x
    TD_unstressed_results["Day 7 Context A Params"][m] = [alpha, gamma1]

```

```python



### FITTING CONTEXT A DAY 7 ### UNSTRESSED
day7_unstressed_attacks_A = np.zeros_like(day7_unstressed_attacks_B)
          # (15, 18000) use as stimuli input

# Minimization for MICE
for m in range(0, len(unstressed_mice_details)):
    # Finding initial for each mouse
    a_prev_prev, g_prev_prev = TD_unstressed_results["Day 1 Context
    A Params"][m]
    initial_prev = TD_model_init(init=0, u=day1_unstressed_attacks[m
    ,:], alpha=a_prev_prev, gamma1=g_prev_prev)[-1]

    a_prev, g_prev = TD_unstressed_results["Day 6 Context B Params"
    ][m]
    initial = TD_model_init(init=initial_prev, u=
    day6_unstressed_attacks_A[m,:], alpha=a_prev, gamma1=g_prev)[-1]

    # Compute TD threat fit from parameters for this day, compute
    NLL
    a, g = TD_unstressed_results["Day 7 Context A Params"][m]
    day7_unstressed_freezing_A = TD_model_init(init=initial, u=
    day7_unstressed_attacks_A[m,:], alpha= a, gamma1=g)  # (15,
    162000) use as threat input

    params = [a, g]
    nll = NLL_td(params=params, init=initial, stimuli=
    day7_unstressed_attacks_A[m,:], threats=
    day7_unstressed_freezing_A)
    # Store in results dataframe:
    TD_unstressed_results["Day 7 Context A NLL"][m] = nll
```

C.3 TD Model - Computing BIC Scores

```python
### BIC SCORES FOR TD MODEL ###

# BIC = 2 * NLL + p*log(n)

```

```
# where p = number of params, n = number of observations (162000 day1 and 18000 for day 6 and 7)
p = 2
# Stressed BIC
TD_stressed_results["BIC Score"] = None
for m in range(len(TD_stressed_results)):
    Day1_NLL = (TD_stressed_results["Day 1 Context A NLL"][m] +
                 TD_stressed_results["Day 1 Context B NLL"][m])

    Day6_NLL = (TD_stressed_results["Day 6 Context A NLL"][m] +
                 TD_stressed_results["Day 6 Context B NLL"][m])

    Day7_NLL = (TD_stressed_results["Day 7 Context A NLL"][m] +
                 TD_stressed_results["Day 7 Context B NLL"][m])

    Day1_BIC = 2 * Day1_NLL + p * np.log(162000)

    Day6_BIC = 2 * Day6_NLL + p * np.log(18000)

    Day7_BIC = 2 * Day7_NLL + p * np.log(18000)

    TD_stressed_results["BIC Score"][m] = Day1_BIC + Day6_BIC + Day7_BIC

# Unstressed BIC
TD_unstressed_results["BIC Score"] = None
for m in range(len(TD_unstressed_results)):
    Day1_NLL = (TD_unstressed_results["Day 1 Context A NLL"][m] +
                 TD_unstressed_results["Day 1 Context B NLL"][m])

    Day6_NLL = (TD_unstressed_results["Day 6 Context A NLL"][m] +
                 TD_unstressed_results["Day 6 Context B NLL"][m])

    Day7_NLL = (TD_unstressed_results["Day 7 Context A NLL"][m] +
                 TD_unstressed_results["Day 7 Context B NLL"][m])

    Day1_BIC = 2 * Day1_NLL + p * np.log(162000)

    Day6_BIC = 2 * Day6_NLL + p * np.log(18000)

    Day7_BIC = 2 * Day7_NLL + p * np.log(18000)

```

```
    TD_unstressed_results["BIC Score"][m] = Day1_BIC + Day6_BIC +
   Day7_BIC
```

C.4 TD Model - Multivariate Sampled Parameter Recovery

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.optimize import minimize
import scipy
import pandas as pd
from scipy.optimize import differential_evolution


# Model Param lists (taken from paper supplementary material)
# Loading matlab file from paper - creates dict "mat_contents" which
     stores all variables
import scipy.io as sio
mat_contents = sio.loadmat('Supp_Mat')


def day_1(shocks):
    '''
    Evaluates day1 of SEFL experiment.

    :return: Outputs sequence of 15 shocks randomly over 90 mins
   (5400 secs)
    '''
    min_shock_int = 4 * 1800
    max_shock_int = 6 * 1800
    # 90 mins = 5400 secs
    day_1 = np.zeros(162000)

    all_shock_intervals = []
    while np.sum(day_1) < shocks:
        # Generate random interval between 4 and 8 mins
        shock_interval = np.random.randint(min_shock_int,
   max_shock_int)
        # Add this to the list of all interval times
        all_shock_intervals.append(shock_interval)
```

```
        # Calculate the shock index by summing cumulative intervals
        shock_index = np.sum(all_shock_intervals)
        # Add shock to day_1 array
        day_1[shock_index] = 1
    return day_1

def NLL_td(params, stimuli, threats):
    '''
    Calculates NLL for one set of parameters given a sequence of attacks and threat predictions for the TD model.
    The likelihood calculation uses the PMF of a Bernoulli distribution to calculate the log probability of observing
    the given threat probability "threat" based on the current value of T.

    :param params: Array of parameters alpha and gamma1
    :param stimuli: Sequence of shocks (attacks)
    :param threats: Sequence of threat predictions

    :return: Negative Log-Likelihood of one set of parameters.
    '''
    alpha, gamma1 = params
    T_prev = 0.0
    log_likelihood = 0.0

    for u, threat in zip(stimuli, threats):

        T = T_prev + alpha * (u - gamma1 * T_prev)

        # Getting some invalid log values, divisions by 0 so adding small value to stop this
        eps = 1e-10
        T = np.clip(T, 0+eps, 1-eps)

        # Assuming threat distribution is Bernoulli - this appears to work as needed
        # Uses probability mass function (PMF) of a Bernoulli distribution: considers threats as probabilities, i.e.
        # each 'threat' represents the prob of observing a shock.
        log_likelihood += np.log((T ** threat) * ((1 - T) ** (1 - threat)))

```

```
        T_prev = T

    nll = -log_likelihood

    return nll


def TD_model(u, alpha, gamma1):
    '''
    Evaluates TD model threat at each timestep

    :param u: Input (sequence of unconditioned stimuli, i.e.
   sequences of footshocks, 0s and 1s), for different
    contexts, i.e A and B (day 1 is context A, day 6 and 7 is
   context B).
    :param alpha: Learning rate
    :param gamma1: Decay rate for threat

    :return: TD model threat estimation levels over all timesteps
    '''
    T = np.zeros_like(u)
    for t in range(1, len(T)):
        T[t] = T[t-1] + alpha * (u[t] - gamma1 * T[t-1])

    return T

# Use this now to generate 100 simulations and fit params
# Draw 100 sets of param values from suitable dist, e.g.
   multivariate normal distribution
# Set covariance to 0, choose small numbers for variance.
# Check values before using. Clip to 0 and 1.

alpha_mean = 0.5
gamma1_mean = 0.95
means = [alpha_mean, gamma1_mean]

alpha_var = 0.03
gamma1_var = 0.001
covs = np.zeros((2,2))
covs[0,0] = alpha_var
covs[1,1] = gamma1_var

```

```
sample_params = np.random.multivariate_normal(mean=means, cov=covs,
    size=100)
sample_params[:,0] = np.clip(sample_params[:,0], 0, 1)
sample_params[:,1] = np.clip(sample_params[:,1], 0.9, 1)


'''
# Plot showing parameter values for reference
fig, ax = plt.subplots(2)
fig.suptitle("Sampled Parameter Values")
# add a big axes, hide frame
fig.add_subplot(111, frameon=False)
# hide tick and tick label of the big axes
plt.tick_params(labelcolor='none', top=False, bottom=False, left=
    False, right=False)
plt.grid(False)
x = np.linspace(0, 100, 100)
ax[0].plot(x, sample_params[:,0], "r+", label="Sampled $alpha$")
ax[1].plot(x, sample_params[:,1], "bo", label="Sampled $gamma_1$")
plt.xlabel("Simulation (Individual Mouse)")
plt.ylabel("Sampled Parameter Value")
ax[1].legend(fontsize="8")
ax[0].legend(fontsize="8")
plt.show()
'''

# Now create simulations of attacks and corresponding threat
    predictions for each pair of params:
sampled_attacks = np.zeros((100, 162000))
sampled_predictions = np.zeros((100, 162000))

for index, (a, g1) in enumerate(zip(sample_params[:,0],
    sample_params[:,1])):
    # Generate day 1 sequence of attacks for each
    attacks = day_1(15)
    # Generate threat predictions from this simulated sequence of
    attacks
    predictions = TD_model(u = attacks, alpha = a, gamma1 = g1)

    # Add to sampled attacks/preds arrays
    sampled_attacks[index, :] = attacks
    sampled_predictions[index, :] = predictions
```

```python

np.sum(sampled_predictions, axis=1)

# Fitting parameter values to these simulations (see if we can recover params)
initial_vals = np.array((0.5, 0.95))
bounds = [(0.05,0.9), (0.9,1)]

opt_sampled_data = np.zeros((len(sampled_predictions[:,0]), 2))

# Minimization for simulations (Mice)
for m in range(0, len(sampled_predictions[:,0])):
    # Compute minimization for each participant
    result = differential_evolution(NLL_td, x0=initial_vals,
                                    bounds=bounds,
                                    args=(sampled_attacks[m:m + 1].reshape(-1), sampled_predictions[m:m + 1].reshape(-1)),
                                    strategy='best1bin', polish=True)

    print(f'n_iter: {result.nit} - success: {result.success} - nll {result.fun}')

    alpha, gamma1 = result.x
    # Add results to opt_params storing opt param values for each participant
    opt_sampled_data[m, 0] = alpha
    opt_sampled_data[m, 1] = gamma1

# Now, to see if these match up fairly well (as they should) to the real simulated params used
#Alpha params
multivariate_sim_alpha = sample_params[:,0]
fitted_simulated_alpha = opt_sampled_data[:,0]
alpha_pearson = scipy.stats.pearsonr(multivariate_sim_alpha, fitted_simulated_alpha)

# gamma1 params
multivariate_sim_gamma1 = sample_params[:,1]
fitted_simulated_gamma1 = opt_sampled_data[:,1]
gamma1_pearson = scipy.stats.pearsonr(multivariate_sim_gamma1, fitted_simulated_gamma1)
```

C.5 TD Model - List Sampled Parameter Recovery

```
# Extracting lists form supplementary information
alpha_list = mat_contents["learning_rate_list"][0].tolist()
gamma1_list = mat_contents["decay_list"][0].tolist()

sample_params = np.zeros((100, 2))

for s in range(100):
    sample_params[s, 0] = np.random.choice(alpha_list)
    sample_params[s, 1] = np.random.choice(gamma1_list)


# Now create simulations of attacks and corresponding threat predictions for each set of params

sampled_attacks = np.zeros((100, 162000))
sampled_predictions = np.zeros((100, 162000))

for index, (a, g1) in enumerate(zip(sample_params[:,0], sample_params[:,1])):
    # Generate day 1 sequence of attacks for each
    attacks = day_1(15)
    # Generate threat predictions from this simulated sequence of attacks
    predictions = TD_model(u = attacks, alpha = a, gamma1 = g1)

    # Add to sampled attacks/preds arrays
    sampled_attacks[index, :] = attacks
    sampled_predictions[index, :] = predictions

# Sanity check
np.sum(sampled_predictions, axis=1)

# Fitting parameter values to these simulations (see if we can recover params)
initial_vals = np.array((0.5, 0.95))

bounds = [(0.05, 0.9), (0.9, 1)]

opt_sampled_data = np.zeros((len(sampled_predictions[:,0]), 2))

```

```
# Minimization for simulations (Mice)
for m in range(0, len(sampled_predictions[:,0])):
    # Compute minimization for each participant
    result = differential_evolution(NLL_td, x0=initial_vals,
                                    bounds=bounds,
                                    args=(sampled_attacks[m:m + 1].reshape(-1), sampled_predictions[m:m + 1].reshape(-1)),
                                    strategy='best1bin', polish=True)

    alpha, gamma1 = result.x
    # Add results to opt_params storing opt param values for each participant
    opt_sampled_data[m, 0] = alpha
    opt_sampled_data[m, 1] = gamma1


# This results in poor recovery, many pairs don't change from initial values!
#Alpha params
selected_sim_alpha = sample_params[:,0]
fitted_simulated_alpha = opt_sampled_data[:,0]
alpha_pearson = scipy.stats.pearsonr(selected_sim_alpha, fitted_simulated_alpha)

# gamma1 params
selected_sim_gamma1 = sample_params[:,1]
fitted_simulated_gamma1 = opt_sampled_data[:,1]
gamma1_pearson = scipy.stats.pearsonr(selected_sim_gamma1, fitted_simulated_gamma1)
```

Appendix D

TD-Momentum Model Code

D.1 TD-Momentum Model - Example of SEFL Model Fit & Related Freezing (Fig 3.7)

```
import numpy as np
from scipy.optimize import minimize
import scipy.io as sio
mat_contents = sio.loadmat('Supp_Mat')
import matplotlib.pyplot as plt


### PRE-PROCESSING START ###

# DAY1: 162000 steps, 38 mice
# Day 1 Freezing
# This is the first mice freezing! [:,1] for next etc, 38 total.
    Shape to be (38, 162000), 162000 timesteps

day1_freezing = np.zeros((38, 162000))

for m in range(38):
    day1_freezing[m,:] = mat_contents["sefl_behavior_day1"]["
    smoothed_freezing"][:,m][0].reshape(-1)

# Remove rows for removed mice
day1_freezing = np.delete(day1_freezing, 8, axis=0)
day1_freezing = np.delete(day1_freezing, 33, axis=0)
day1_freezing = np.delete(day1_freezing, 26, axis=0)
```

```


# Day 1 Shocks:
# 38 total
# Mouse 9 removed - no data
# Mouse 28 removed - too many shocks
# Mouse 35 removed - too many shocks
day1_shock_times = mat_contents["sefl_behavior_day1"]["shock_times"]
# Remove indexes from day1_shock_times
day1_shock_times_modified = np.delete(day1_shock_times, 8, axis=1)
day1_shock_times_modified = np.delete(day1_shock_times_modified, 33,
     axis=1)
day1_shock_times_modified = np.delete(day1_shock_times_modified, 26,
     axis=1)

day1_shock_times = np.zeros((35, 15))

for m in range(35):
    day1_shock_times[m,:] = day1_shock_times_modified[:,m][0].
    reshape(-1)

# Now creating the actual attack sequences (0s and 1s every timestep
    )
day1_attack_sequences = np.zeros((35, 162000))

for m in range(35):
    times = day1_shock_times[m,:]
    indexes = times-1

    for i in indexes:
        day1_attack_sequences[m, int(i)] = 1


# Array of stress type for each mouse, 0 is unstressed, 1 is
    stressed (unstressed are controls = no shocks on day1)
day1_stress_type = np.zeros(38)

for m in range(38):
    day1_stress_type[m] = mat_contents["sefl_behavior_day1"]["stress
    "][0][m][0][0]

# Removing rows for removed mice
```

```
day1_stress_type = np.delete(day1_stress_type, 8, axis=0)
day1_stress_type = np.delete(day1_stress_type, 33, axis=0)
day1_stress_type = np.delete(day1_stress_type, 26, axis=0)

# Final arrays for stressed and unstressed mice indexes
stressed_indexes = np.where(day1_stress_type == 1)[0].tolist()
unstressed_indexes = np.where(day1_stress_type == 0)[0].tolist()


### PRE-PROCESSING END ###



### EXTRACTING EXAMPLE MOUSE FOOTSHOCKS TO FEED INTO NEW SCALED TD MOM MODEL TO REPROUCE FIG5A ###


def TD_momentum_model_days1_6_7(init, contexts, alpha, gamma1, gamma2, f):
    '''
    Evaluates TD Momentum model threat at each timestep in each context only, no changing contexts, assumes the same
    agent in one contex for the whole time, creates threat in other context as momentum term only.

    :param init: Provides initialisation points for threat readings in both contexts (days 6 and 7)
    :param contexts: Sequences of unconditioned stimuli across all contexts. e.g. sefl(sims = 2)[1] for 2nd mouse sim
    :param u: Key of chosen context to compute TD momentum model for. (day1, day6, day7)
    :param alpha: Learning rate
    :param gamma1: Decay rate for threat
    :param gamma2: Decay rate for momentum across all contexts
    :param f: Scaling Constant for momentum term

    :return: TD Momentum model threat estimation levels over all timesteps
    '''
    # Initialise momentum array
    m = np.zeros_like(contexts[:,0])

```

```
    T_A = np.zeros_like(contexts[:,0])
    T_B = np.zeros_like(contexts[:,1])

    # Set initialisation
    T_A[0] = init[0]
    T_B[0] = init[1]

    for t in range(1, len(contexts[:,0])):
        # Set PE to 0 at every time step before computing PE for
    current step
        PE = 0
        PE += alpha * (contexts[:, 0][t] - T_A[t - 1])
        PE += alpha * (contexts[:, 1][t] - T_B[t - 1])

        m[t] = m[t-1] + gamma2 * PE
        m[t] = np.clip(m[t], 0, 1)    # NEWLY ADDED 25/07/23

        T_A[t] = T_A[t - 1] + alpha * (contexts[:, 0][t] - gamma1 *
    T_A[t - 1]) + f * m[t]
        T_B[t] = T_B[t - 1] + alpha * (contexts[:, 1][t] - gamma1 *
    T_B[t - 1]) + f * m[t]

        T_A[t] = np.clip(T_A[t], 0, 1)
        T_B[t] = np.clip(T_B[t], 0, 1)

    return T_A, T_B

# Freezing and attacks for ALL mice (35)
day1_stressed_freezing = day1_freezing[stressed_indexes]
day1_stressed_attacks = day1_attack_sequences[stressed_indexes]


### DAY 1 ###
# Taking first stressed mouse as example, extracting shock sequence
    for day 1: 0
mouse_1_day1_context_A = day1_stressed_attacks[10]
mouse_1_day1_context_B = np.zeros_like(mouse_1_day1_context_A)

# Day 1 threat
day1_contexts_stacked = np.column_stack((mouse_1_day1_context_A,
    mouse_1_day1_context_B))
x = np.linspace(0, 90, len(mouse_1_day1_context_A))
```

```

init_day1 = [0, 0]
A_threat_day1, B_threat_day1 = TD_momentum_model_days1_6_7(init =
    init_day1, contexts = day1_contexts_stacked, alpha =0.000015,
                                                        gamma1 =
    0.9999, gamma2 = 0.01, f = 0.1)


### Day 6 ###
# in both contexts for 10 mins, single shock in context B halfway
    through

mouse_1_day6_context_A = np.zeros(18000)                          #
    18000 timesteps on day 6 and day 7 (10 mins exposure)
mouse_1_day6_context_B = np.zeros_like(mouse_1_day6_context_A)
mouse_1_day6_context_B[8999] = 1                                  #
    Single attack at halfway point

# day 6 threat:
day6_contexts_stacked = np.column_stack((mouse_1_day6_context_A,
    mouse_1_day6_context_B))
x = np.linspace(0, 10, len(mouse_1_day6_context_A))

init_day6 = [A_threat_day1[-1], B_threat_day1[-1]]
A_threat_day6, B_threat_day6 = TD_momentum_model_days1_6_7(init =
    init_day6, contexts = day6_contexts_stacked, alpha =0.000015,
                                                        gamma1 =
    0.9999, gamma2 = 0.01, f = 0.1)


### DAY 7 ###
mouse_1_day7_context_A = np.zeros(18000)                          #
    18000 timesteps on day 6 and day 7 (10 mins exposure)
mouse_1_day7_context_B = np.zeros_like(mouse_1_day7_context_A)

day7_contexts_stacked = np.column_stack((mouse_1_day7_context_A,
    mouse_1_day7_context_B))
x = np.linspace(0, 10, len(mouse_1_day7_context_A))

init_day7 = [A_threat_day6[-1], B_threat_day6[-1]]
A_threat_day7, B_threat_day7 = TD_momentum_model_days1_6_7(init =
    init_day7, contexts = day7_contexts_stacked, alpha =0.000015,
```

```
                                                            gamma1 =
    0.9999, gamma2 = 0.01, f = 0.1)



# Scale function for 6 arrays: Contexts A and B for days 1, 6 and 7
def combined_rescale_example(arrs, new_min, new_max):
    combined_min = min(np.min(arr) for arr in arrs)
    combined_max = max(np.max(arr) for arr in arrs)
    rescaled_arrs = [(arr - combined_min) * (new_max - new_min) / (
    combined_max - combined_min) + new_min for arr in arrs]
    return rescaled_arrs

# Scaling all days/contexts together:
scale_arrays = [A_threat_day1, B_threat_day1, A_threat_day6,
    B_threat_day6, A_threat_day7, B_threat_day7]

A_threat_day1, B_threat_day1, A_threat_day6, B_threat_day6,
    A_threat_day7, B_threat_day7 = combined_rescale_example(
    scale_arrays, 0.1, 0.9)


### G34 FREEZING DATA ###
### DAY 1 ###

mouse_1_day1_freezing = mat_contents["sefl_behavior_day1"]["
    smoothed_freezing"][:,25][0].reshape(-1)
mouse_1_day1_freezing = mouse_1_day1_freezing * 100
x = np.linspace(0, 90, len(mouse_1_day1_freezing))

### DAY 6 ###
mouse_1_day6_freezing = mat_contents["sefl_behavior_day6"]["
    smoothed_freezing"][:,25][0].reshape(-1)
mouse_1_day6_freezing = mouse_1_day6_freezing * 100
x = np.linspace(0, 10, len(mouse_1_day6_freezing))

### DAY 7 ###
mouse_1_day7_freezing = mat_contents["sefl_behavior_day7"]["
    smoothed_freezing"][:,24][0].reshape(-1)
mouse_1_day7_freezing = mouse_1_day7_freezing * 100
x = np.linspace(0, 10, len(mouse_1_day7_freezing))

```

```python




# Plotting RL Momentum Threat example and actual smoothed freezing
fig, axes = plt.subplots(2, 3, figsize=(15, 10))

# Plotting TD-Momentum Threat data for each day
days = [1, 6, 7]
for i, day in enumerate(days):
    if day == 1:
        x = np.linspace(0, 90, len(mouse_1_day1_context_A))
    else:
        x = np.linspace(0, 10, len(mouse_1_day6_context_A)) if day
   == 6 else np.linspace(0, 10, len(mouse_1_day7_context_A))

    # Plot the threat data and context data on the top row axes
    axes[1, i].plot(x, A_threat_day1 if day == 1 else A_threat_day6
   if day == 6 else A_threat_day7, label="A")
    axes[1, i].plot(x, B_threat_day1 if day == 1 else B_threat_day6
   if day == 6 else B_threat_day7, label="B")
    axes[1, i].plot(x, mouse_1_day1_context_A if day == 1 else
   mouse_1_day6_context_A if day == 6 else mouse_1_day7_context_A,
   label="A input", alpha=0.20)
    axes[1, i].plot(x, mouse_1_day1_context_B if day == 1 else
   mouse_1_day6_context_B if day == 6 else mouse_1_day7_context_B,
   label="B input", alpha=0.20)
    axes[1, i].set_xlabel("Time (mins)", fontsize=15)
    axes[1, i].set_ylabel("TD-Momentum Threat", fontsize=15)
    axes[1, i].set_ylim(0, 1)
    axes[1, i].legend()

    if i != 0:
        axes[1, i].set_yticklabels([])
        axes[1, i].set_ylabel("")

# Hide the x-axis labels for the top row
plt.setp(axes[0, :], xticks=[])

# Plotting Freezing data for each day
for i, day in enumerate(days):
    if day == 1:
```

```
        x = np.linspace(0, 90, len(mouse_1_day1_freezing))
        freezing_data = mouse_1_day1_freezing
    else:
        x = np.linspace(0, 10, len(mouse_1_day6_freezing)) if day ==
     6 else np.linspace(0, 10, len(mouse_1_day7_freezing))
        freezing_data = mouse_1_day6_freezing if day == 6 else
    mouse_1_day7_freezing

    # Plot the freezing data on the bottom row axes
    axes[0, i].plot(x, freezing_data, label="Freezing", color="r",
    alpha=0.6)
    axes[0, i].set_title(f"Day {day} - Context A", fontsize=15)
    axes[0, i].set_ylabel("Freezing (%)", fontsize=15)
    axes[0, i].set_ylim(0, 100)
    axes[0, i].legend()

    if i != 0:
        axes[0, i].set_yticklabels([])
        axes[0, i].set_ylabel("")

    if i == 1:
        axes[0, i].set_title(f"Day {day} - Context B", fontsize=15)

    if i == 2:
        axes[0, i].set_title(f"Day {day} - Context B", fontsize=15)

plt.tight_layout()
plt.show()
```

D.2 TD-Momentum Model - Fitting SEFL Data

```
import numpy as np
from sklearn.preprocessing import MinMaxScaler
from scipy.optimize import minimize
from scipy.optimize import differential_evolution
import pandas as pd
import scipy.stats
import matplotlib.pyplot as plt

import scipy.io as sio
mat_contents = sio.loadmat('Supp_Mat')
```

```

# Copying versions of TD for use of TD Mom model and to stored
    results:

TD_Mom_stressed_results = pd.DataFrame.copy(TD_stressed_results)
TD_Mom_unstressed_results = pd.DataFrame.copy(TD_unstressed_results)

TD_Mom_stressed_results[TD_Mom_stressed_results.columns[1:]] = None
TD_Mom_unstressed_results[TD_Mom_stressed_results.columns[1:]] =
    None


### Defining functions required ###
def combined_rescale(arr1, arr2, new_min, new_max):
    combined_min = min(np.min(arr1), np.min(arr2))
    combined_max = max(np.max(arr1), np.max(arr2))
    rescaled_arr1 = (arr1 - combined_min) * (new_max - new_min) / (
    combined_max - combined_min + 1e-10) + new_min
    rescaled_arr2 = (arr2 - combined_min) * (new_max - new_min) / (
    combined_max - combined_min + 1e-10) + new_min
    return rescaled_arr1, rescaled_arr2

def TD_momentum_model_init(init, contexts, alpha, gamma1, gamma2, f)
    :
    '''
    Evaluates TD Momentum model threat at each timestep in each
    context only, no changing contexts, assumes the same
    agent in one contex for the whole time, creates threat in other
    context as momentum term only.

    :param init: Provides initialisation points for threat readings
    in both contexts (days 6 and 7)
    :param contexts: Sequences of unconditioned stimuli across all
    contexts. e.g. sefl(sims = 2)[1] for 2nd mouse sim
    :param u: Key of chosen context to compute TD momentum model for
    . (day1, day6, day7)
    :param alpha: Learning rate
    :param gamma1: Decay rate for threat
    :param gamma2: Decay rate for momentum across all contexts
    :param f: Scaling Constant for momentum term

    :return: TD Momentum model threat estimation levels over all
```

```
    timesteps
    '''
    # Initialise momentum array
    m = np.zeros_like(contexts[:,0])

    T_A = np.zeros_like(contexts[:,0])
    T_B = np.zeros_like(contexts[:,1])

    # Set initialisation
    T_A[0] = init[0]
    T_B[0] = init[1]

    for t in range(1, len(contexts[:,0])):
        # Set PE to 0 at every time step before computing PE for current step
        PE = 0
        PE += alpha * (contexts[:, 0][t] - T_A[t - 1])
        PE += alpha * (contexts[:, 1][t] - T_B[t - 1])

        m[t] = m[t-1] + gamma2 * PE
        m[t] = np.clip(m[t], 0, 1)

        T_A[t] = T_A[t - 1] + alpha * (contexts[:, 0][t] - gamma1 * T_A[t - 1]) + f * m[t]
        T_B[t] = T_B[t - 1] + alpha * (contexts[:, 1][t] - gamma1 * T_B[t - 1]) + f * m[t]

        T_A[t] = np.clip(T_A[t], 0, 1)
        T_B[t] = np.clip(T_B[t], 0, 1)

    T_A, T_B = combined_rescale(T_A, T_B, 0.1, 0.9)

    return T_A, T_B

def NLL_td_mom(params, init, stimuli, threats, u):
    alpha, gamma1, gamma2, f = params
    log_likelihood = 0.0

    T_A, T_B = TD_momentum_model_init(init = init, contexts = stimuli, alpha = alpha, gamma1 = gamma1, gamma2 = gamma2, f = f)

    eps = 1e-10
```

```
    T_A = np.clip(T_A, 0+eps, 1-eps)
    T_B = np.clip(T_B, 0+eps, 1-eps)

    if u == 0:
        Tu = T_A

    if u == 1:
        Tu = T_B

    for t in range(len(stimuli[:,0])):
        log_likelihood += threats[t, u] * np.log(Tu[t]) + (1 -
    threats[t, u]) * np.log(1 - Tu[t])

    nll = -log_likelihood

    return nll

def smoothing(raw_time_series, window_size):
    smoothed_time_series = []
    half_window = window_size // 2

    # Pad the time series
    padded_time_series = np.pad(raw_time_series, (half_window,
    half_window), mode='edge')

    for i in range(len(raw_time_series)):
        window_values = padded_time_series[i : i + window_size]
        smoothed_value = np.mean(window_values)  # Average of window
        smoothed_time_series.append(smoothed_value)

    return np.array(smoothed_time_series)






### FITTING CONTEXT A DAY 1 ### STRESSED

initial = [0,0]
initial_vals = np.array((0.5, 0.95, 0.4, 1.5))
bounds = [(0.05, 1), (0.9, 1), (0, 0.8), (0, 3)]
```

```

# Minimization for MICE
for m in range(0, len(stressed_mice_details)):
    # Compute minimization for each participant
    B_shocks = np.zeros_like(day1_stressed_attacks[m:m + 1].reshape(-1))
    contexts_stacked = np.column_stack((day1_stressed_attacks[m:m + 1].reshape(-1), B_shocks))

    result_A = differential_evolution(NLL_td_mom, x0=initial_vals, bounds=bounds,
                                      args=(initial, contexts_stacked, day1_stressed_freezing[m:m + 1].reshape((162000, 1)), 0),
                                      polish=True)

    print(f'A: n_iter: {result_A.nit} - success: {result_A.success} - nll {result_A.fun}')

    # Store in results dataframe:
    TD_Mom_stressed_results["Day 1 Context A NLL"][m] = result_A.fun
    alpha, gamma1, gamma2, f = result_A.x
    TD_Mom_stressed_results["Day 1 Context A Params"][m] = [alpha, gamma1, gamma2, f]




### FITTING CONTEXT B DAY 1 ### STRESSED
day1_stressed_attacks_B = np.zeros_like(day1_stressed_attacks)     # (15, 162000) use as stimuli input
initial = [0,0]

# Minimization for MICE
for m in range(0, len(stressed_mice_details)):
    # Compute minimization for each participant
    B_shocks = np.zeros_like(day1_stressed_attacks[m:m + 1].reshape(-1))
    contexts_stacked = np.column_stack((day1_stressed_attacks[m:m + 1].reshape(-1), B_shocks))

    # Compute TD threat fit from parameters for this day, compute NLL
```

```
    a, g1, g2, f = TD_Mom_stressed_results["Day 1 Context A Params"][m]
    day1_stressed_freezing_B = TD_momentum_model_init(init = initial, contexts = contexts_stacked,
                                                                        alpha=a, gamma1=g1, gamma2=g2, f=f)[1]

    params = [a, g1, g2, f]
    threats_stacked = np.column_stack((day1_stressed_freezing[m], day1_stressed_freezing_B))
    nll = NLL_td_mom(params=params, init=initial, stimuli=contexts_stacked, threats=threats_stacked, u=1)
    # Store in results dataframe:
    TD_Mom_stressed_results["Day 1 Context B NLL"][m] = nll






# Adjust NLL for day 6 - remove u param
def NLL_td_mom(params, init, stimuli, threats):
    alpha, gamma1, gamma2, f = params
    log_likelihood = 0.0

    T_A, T_B = TD_momentum_model_init(init = init, contexts = stimuli, alpha = alpha, gamma1 = gamma1, gamma2 = gamma2, f = f)

    eps = 1e-10
    T_A = np.clip(T_A, 0+eps, 1-eps)
    T_B = np.clip(T_B, 0+eps, 1-eps)


    Tu = T_B

    for t in range(len(stimuli[:,0])):
        log_likelihood += threats[t, 0] * np.log(Tu[t]) + (1 - threats[t, 0]) * np.log(1 - Tu[t])

    nll = -log_likelihood

    return nll
```

```
### FITTING CONTEXT B DAY 6 ### STRESSED

initial_vals = np.array((0.5, 0.95, 0.4, 1.5))
bounds = [(0.05, 1), (0.9, 1), (0, 0.8), (0, 3)]

# Minimization for MICE
for m in range(0, len(stressed_mice_details)):
    # Setting up initial starting point for threat
    a_prev, g1_prev, g1_prev, f_prev = TD_Mom_stressed_results["Day
   1 Context A Params"][m]
    initial_prev = [0,0]
    B_shocks_prev = np.zeros_like(day1_stressed_attacks[m:m + 1].
   reshape(-1))
    contexts_stacked_prev = np.column_stack((day1_stressed_attacks[m
   :m + 1].reshape(-1), B_shocks_prev))

    initial_B = TD_momentum_model_init(init = initial_prev, contexts
    = contexts_stacked_prev,
                                                                  alpha=a,
   gamma1=g1, gamma2=g2, f=f)[1][-1]

    initial_A = TD_momentum_model_init(init = initial_prev, contexts
    = contexts_stacked_prev,
                                                                  alpha=a,
   gamma1=g1, gamma2=g2, f=f)[0][-1]

    # Compute minimization for each participant
    A_shocks = np.zeros_like(day6_stressed_attacks_B[m:m + 1].
   reshape(-1))
    contexts_stacked = np.column_stack((A_shocks,
   day6_stressed_attacks_B[m:m + 1].reshape(-1)))
    initial = [initial_A, initial_B]

    result_B = differential_evolution(NLL_td_mom, x0=initial_vals,
   bounds=bounds,
                              args=(initial, contexts_stacked,
   day6_stressed_freezing[m:m + 1].reshape((18000, 1))),
```

```
                                                  polish=True)

    print(f'A: n_iter: {result_B.nit} - success: {result_B.success} - nll {result_B.fun}')

    # Store in results dataframe:
    TD_Mom_stressed_results["Day 6 Context B NLL"][m] = result_B.fun
    alpha, gamma1, gamma2, f = result_B.x
    TD_Mom_stressed_results["Day 6 Context B Params"][m] = [alpha, gamma1, gamma2, f]


# Back to original NLL for day 6 context A
def NLL_td_mom(params, init, stimuli, threats, u):
    alpha, gamma1, gamma2, f = params
    log_likelihood = 0.0

    T_A, T_B = TD_momentum_model_init(init = init, contexts = stimuli, alpha = alpha, gamma1 = gamma1, gamma2 = gamma2, f = f)

    eps = 1e-10
    T_A = np.clip(T_A, 0+eps, 1-eps)
    T_B = np.clip(T_B, 0+eps, 1-eps)

    if u == 0:
        Tu = T_A

    if u == 1:
        Tu = T_B

    for t in range(len(stimuli[:,0])):
        log_likelihood += threats[t, u] * np.log(Tu[t]) + (1 - threats[t, u]) * np.log(1 - Tu[t])

    nll = -log_likelihood

    return nll

### FITTING CONTEXT A DAY 6 ### STRESSED

# Minimization for MICE
for m in range(0, len(stressed_mice_details)):
```

```
    # Setting up initial starting point for threat
    a_prev, g1_prev, g1_prev, f_prev = TD_Mom_stressed_results["Day 1 Context A Params"][m]
    initial_prev = [0,0]
    B_shocks_prev = np.zeros_like(day1_stressed_attacks[m:m + 1].reshape(-1))
    contexts_stacked_prev = np.column_stack((day1_stressed_attacks[m:m + 1].reshape(-1), B_shocks_prev))

    initial_A = TD_momentum_model_init(init = initial_prev, contexts = contexts_stacked_prev,
                                                         alpha=a, gamma1=g1, gamma2=g2, f=f)[0][-1]

    initial_B = TD_momentum_model_init(init = initial_prev, contexts = contexts_stacked_prev,
                                                         alpha=a, gamma1=g1, gamma2=g2, f=f)[1][-1]

    # Compute TD Mom threat fit from parameters for this day, compute NLL
    A_shocks = np.zeros_like(day6_stressed_attacks_B[m:m + 1].reshape(-1))
    contexts_stacked = np.column_stack((A_shocks, day6_stressed_attacks_B[m:m + 1].reshape(-1)))
    initial = [initial_A, initial_B]

    a, g1, g2, f = TD_Mom_stressed_results["Day 6 Context B Params"][m]
    day6_stressed_freezing_A = TD_momentum_model_init(init = initial, contexts = contexts_stacked,
                                                         alpha=a, gamma1=g1, gamma2=g2, f=f)[0]

    params = [a, g1, g2, f]
    threats_stacked = np.column_stack((day6_stressed_freezing_A, day6_stressed_freezing[m]))
    nll = NLL_td_mom(params=params, init=initial, stimuli=contexts_stacked, threats=threats_stacked, u=0)
    # Store in results dataframe:
    TD_Mom_stressed_results["Day 6 Context A NLL"][m] = nll

```

```
# Adjust NLL for day 7 - remove u param
def NLL_td_mom(params, init, stimuli, threats):
    alpha, gamma1, gamma2, f = params
    log_likelihood = 0.0

    T_A, T_B = TD_momentum_model_init(init = init, contexts = stimuli, alpha = alpha, gamma1 = gamma1, gamma2 = gamma2, f = f)

    eps = 1e-10
    T_A = np.clip(T_A, 0+eps, 1-eps)
    T_B = np.clip(T_B, 0+eps, 1-eps)


    Tu = T_B

    for t in range(len(stimuli[:,0])):
        log_likelihood += threats[t, 0] * np.log(Tu[t]) + (1 - threats[t, 0]) * np.log(1 - Tu[t])

    nll = -log_likelihood

    return nll


### FITTING CONTEXT B DAY 7 ### STRESSED


initial_vals = np.array((0.5, 0.95, 0.4, 1.5))
bounds = [(0.05, 1), (0.9, 1), (0, 0.8), (0, 3)]

# Minimization for MICE
for m in range(0, len(stressed_mice_details)):
    # Setting up initial starting point for threat
    a_prev_prev, g1_prev_prev, g2_prev_prev, f_prev_prev = TD_Mom_stressed_results["Day 1 Context A Params"][m]
    initial_prev_prev = [0,0]
    B_shocks_prev_prev = np.zeros_like(day1_stressed_attacks[m:m + 1].reshape(-1))
```

```
    contexts_stacked_prev_prev = np.column_stack((
   day1_stressed_attacks[m:m + 1].reshape(-1), B_shocks_prev_prev))

    initial_prev_A = TD_momentum_model_init(init = initial_prev_prev
   , contexts = contexts_stacked_prev_prev,
                                                   alpha=
   a_prev_prev, gamma1=g1_prev_prev,
                                  gamma2=g2_prev_prev, f=
   f_prev_prev)[0][-1]

    initial_prev_B = TD_momentum_model_init(init = initial_prev_prev
   , contexts = contexts_stacked_prev_prev,
                                                   alpha=
   a_prev_prev, gamma1=g1_prev_prev,
                                  gamma2=g2_prev_prev, f=
   f_prev_prev)[1][-1]


    initial_prev = [initial_prev_A, initial_prev_B]
    a_prev, g1_prev, g2_prev, f_prev = TD_Mom_stressed_results["Day
   6 Context B Params"][m]
    A_shocks_prev = np.zeros_like(day6_stressed_attacks_B[m:m + 1].
   reshape(-1))
    contexts_stacked_prev = np.column_stack((A_shocks_prev,
   day6_stressed_attacks_B[m:m + 1].reshape(-1)))

    initial_A = TD_momentum_model_init(init=initial_prev, contexts=
   contexts_stacked_prev,
                                       alpha=a_prev, gamma1=
   g1_prev,
                                       gamma2=g2_prev, f=f_prev)
   [0][-1]

    initial_B = TD_momentum_model_init(init=initial_prev, contexts=
   contexts_stacked_prev,
                                       alpha=a_prev, gamma1=
   g1_prev,
                                       gamma2=g2_prev, f=f_prev)
   [1][-1]

    # Compute minimization for each participant
    initial = [initial_A, initial_B]
```

```
    A_shocks = np.zeros_like(day7_stressed_attacks_A[m:m + 1].reshape(-1))    # doesn't matter if A or B, both filled with zeros
    contexts_stacked = np.column_stack((A_shocks, day7_stressed_attacks_B[m:m + 1].reshape(-1)))

    result_B = differential_evolution(NLL_td_mom, x0=initial_vals, bounds=bounds,
                           args=(initial, contexts_stacked, day7_stressed_freezing[m:m + 1].reshape((18000, 1))),
                                    polish=True)

    print(f'B: n_iter: {result_B.nit} - success: {result_B.success} - nll {result_B.fun}')

    # Store in results dataframe:
    TD_Mom_stressed_results["Day 7 Context B NLL"][m] = result_B.fun
    alpha, gamma1, gamma2, f = result_B.x
    TD_Mom_stressed_results["Day 7 Context A Params"][m] = [alpha, gamma1, gamma2, f]





# Back to original NLL for day 7 context A
def NLL_td_mom(params, init, stimuli, threats, u):
    alpha, gamma1, gamma2, f = params
    log_likelihood = 0.0

    T_A, T_B = TD_momentum_model_init(init = init, contexts = stimuli, alpha = alpha, gamma1 = gamma1, gamma2 = gamma2, f = f)

    eps = 1e-10
    T_A = np.clip(T_A, 0+eps, 1-eps)
    T_B = np.clip(T_B, 0+eps, 1-eps)

    if u == 0:
        Tu = T_A

    if u == 1:
        Tu = T_B
```

```

    for t in range(len(stimuli[:,0])):
        log_likelihood += threats[t, u] * np.log(Tu[t]) + (1 -
    threats[t, u]) * np.log(1 - Tu[t])

    nll = -log_likelihood

    return nll
### FITTING CONTEXT A DAY 7 ### STRESSED

for m in range(0, len(stressed_mice_details)):
    # Setting up initial starting point for threat
    a_prev_prev, g1_prev_prev, g2_prev_prev, f_prev_prev =
    TD_Mom_stressed_results["Day 1 Context A Params"][m]
    initial_prev_prev = [0, 0]
    B_shocks_prev_prev = np.zeros_like(day1_stressed_attacks[m:m +
    1].reshape(-1))
    contexts_stacked_prev_prev = np.column_stack((
    day1_stressed_attacks[m:m + 1].reshape(-1), B_shocks_prev_prev))

    initial_prev_A = TD_momentum_model_init(init=initial_prev_prev,
    contexts=contexts_stacked_prev_prev,
                                            alpha=a_prev_prev, gamma1=
    g1_prev_prev,
                                            gamma2=g2_prev_prev, f=
    f_prev_prev)[0][-1]

    initial_prev_B = TD_momentum_model_init(init=initial_prev_prev,
    contexts=contexts_stacked_prev_prev,
                                            alpha=a_prev_prev, gamma1=
    g1_prev_prev,
                                            gamma2=g2_prev_prev, f=
    f_prev_prev)[1][-1]

    a_prev, g1_prev, g2_prev, f_prev = TD_Mom_stressed_results["Day
    6 Context B Params"][m]
    A_shocks_prev = np.zeros_like(day6_stressed_attacks_B[m:m + 1].
    reshape(-1))
    contexts_stacked_prev = np.column_stack((A_shocks_prev,
    day6_stressed_attacks_B[m:m + 1].reshape(-1)))
    initial_prev = [initial_prev_A, initial_prev_B]

```

```
        initial_A = TD_momentum_model_init(init=initial_prev, contexts=contexts_stacked_prev,
                                           alpha=a_prev, gamma1=g1_prev,
                                           gamma2=g2_prev, f=f_prev)[0][-1]

        initial_B = TD_momentum_model_init(init=initial_prev, contexts=contexts_stacked_prev,
                                           alpha=a_prev, gamma1=g1_prev,
                                           gamma2=g2_prev, f=f_prev)[1][-1]

        # Compute minimization for each participant
        A_shocks = np.zeros_like(day7_stressed_attacks_A[m:m + 1].reshape(-1))
        contexts_stacked = np.column_stack((A_shocks, day7_stressed_attacks_B[m:m + 1].reshape(-1)))
        a, g1, g2, f = TD_Mom_stressed_results["Day 7 Context A Params"][m]

        initial = [initial_A, initial_B]

        day7_stressed_freezing_A = TD_momentum_model_init(init=initial, contexts=contexts_stacked, alpha=a, gamma1=g1,
                                                          gamma2=g2, f=f)[0]

        params = [a, g1, g2, f]
        threats_stacked = np.column_stack((day7_stressed_freezing_A, day7_stressed_freezing[m]))
        nll = NLL_td_mom(params=params, init=initial, stimuli=contexts_stacked, threats=threats_stacked, u=0)
        # Store in results dataframe:
        TD_Mom_stressed_results["Day 7 Context A NLL"][m] = nll




### NOW FOR UNSTRESSED ###



```

```
### FITTING CONTEXT A DAY 1 ### UNSTRESSED

initial = [0,0]
initial_vals = np.array((0.5, 0.95, 0.4, 1.5))
bounds = [(0.05, 1), (0.9, 1), (0, 0.8), (0, 3)]

# Minimization for MICE
for m in range(0, len(unstressed_mice_details)):
    # Compute minimization for each participant
    B_shocks = np.zeros_like(day1_unstressed_attacks[m:m + 1].reshape(-1))
    contexts_stacked = np.column_stack((day1_unstressed_attacks[m:m + 1].reshape(-1), B_shocks))

    result_A = differential_evolution(NLL_td_mom, x0=initial_vals, bounds=bounds,
                               args=(initial, contexts_stacked, day1_unstressed_freezing[m:m + 1].reshape((162000, 1)), 0),
                                        polish=True)

    print(f'A: n_iter: {result_A.nit} - success: {result_A.success} - nll {result_A.fun}')

    # Store in results dataframe:
    TD_Mom_unstressed_results["Day 1 Context A NLL"][m] = result_A.fun
    alpha, gamma1, gamma2, f = result_A.x
    TD_Mom_unstressed_results["Day 1 Context A Params"][m] = [alpha, gamma1, gamma2, f]




### FITTING CONTEXT B DAY 1 ### UNSTRESSED
day1_unstressed_attacks_B = np.zeros_like(day1_unstressed_attacks)        # (15, 162000) use as stimuli input
initial = [0,0]

# Minimization for MICE
for m in range(0, len(unstressed_mice_details)):
    # Compute minimization for each participant
    B_shocks = np.zeros_like(day1_unstressed_attacks[m:m + 1].
```

```
    reshape(-1))
    contexts_stacked = np.column_stack((day1_unstressed_attacks[m:m
    + 1].reshape(-1), B_shocks))

    # Compute TD threat fit from parameters for this day, compute
    NLL
    a, g1, g2, f = TD_Mom_unstressed_results["Day 1 Context A Params
    "][m]
    day1_unstressed_freezing_B = TD_momentum_model_init(init =
    initial, contexts = contexts_stacked,
                                                              alpha=a,
    gamma1=g1, gamma2=g2, f=f)[1]

    params = [a, g1, g2, f]
    threats_stacked = np.column_stack((day1_unstressed_freezing[m],
    day1_unstressed_freezing_B))
    nll = NLL_td_mom(params=params, init=initial, stimuli=
    contexts_stacked, threats=threats_stacked, u=1)
    # Store in results dataframe:
    TD_Mom_unstressed_results["Day 1 Context B NLL"][m] = nll






# Adjust NLL for day 6 - remove u param
def NLL_td_mom(params, init, stimuli, threats):
    alpha, gamma1, gamma2, f = params
    log_likelihood = 0.0

    T_A, T_B = TD_momentum_model_init(init = init, contexts =
    stimuli, alpha = alpha, gamma1 = gamma1, gamma2 = gamma2, f = f)

    eps = 1e-10
    T_A = np.clip(T_A, 0+eps, 1-eps)
    T_B = np.clip(T_B, 0+eps, 1-eps)


    Tu = T_B

    for t in range(len(stimuli[:,0])):
```

```
        log_likelihood += threats[t, 0] * np.log(Tu[t]) + (1 - threats[t, 0]) * np.log(1 - Tu[t])

    nll = -log_likelihood

    return nll





### FITTING CONTEXT B DAY 6 ### UNSTRESSED

initial_vals = np.array((0.5, 0.95, 0.4, 1.5))
bounds = [(0.05, 1), (0.9, 1), (0, 0.8), (0, 3)]

# Minimization for MICE
for m in range(0, len(unstressed_mice_details)):
    # Setting up initial starting point for threat
    a_prev, g1_prev, g1_prev, f_prev = TD_Mom_unstressed_results["Day 1 Context A Params"][m]
    initial_prev = [0,0]
    B_shocks_prev = np.zeros_like(day1_unstressed_attacks[m:m + 1].reshape(-1))
    contexts_stacked_prev = np.column_stack((day1_unstressed_attacks[m:m + 1].reshape(-1), B_shocks_prev))

    initial_B = TD_momentum_model_init(init = initial_prev, contexts = contexts_stacked_prev,
                                                                   alpha=a, gamma1=g1, gamma2=g2, f=f)[1][-1]

    initial_A = TD_momentum_model_init(init = initial_prev, contexts = contexts_stacked_prev,
                                                                   alpha=a, gamma1=g1, gamma2=g2, f=f)[0][-1]

    # Compute minimization for each participant
    A_shocks = np.zeros_like(day6_unstressed_attacks_B[m:m + 1].reshape(-1))
    contexts_stacked = np.column_stack((A_shocks, day6_unstressed_attacks_B[m:m + 1].reshape(-1)))
```

```
    initial = [initial_A, initial_B]

    result_B = differential_evolution(NLL_td_mom, x0=initial_vals, bounds=bounds,
                                args=(initial, contexts_stacked, day6_unstressed_freezing[m:m + 1].reshape((18000, 1))),
                                          polish=True)

    print(f'A: n_iter: {result_B.nit} - success: {result_B.success} - nll {result_B.fun}')

    # Store in results dataframe:
    TD_Mom_unstressed_results["Day 6 Context B NLL"][m] = result_B.fun
    alpha, gamma1, gamma2, f = result_B.x
    TD_Mom_unstressed_results["Day 6 Context B Params"][m] = [alpha, gamma1, gamma2, f]


# Back to original NLL for day 6 context A
def NLL_td_mom(params, init, stimuli, threats, u):
    alpha, gamma1, gamma2, f = params
    log_likelihood = 0.0

    T_A, T_B = TD_momentum_model_init(init = init, contexts = stimuli, alpha = alpha, gamma1 = gamma1, gamma2 = gamma2, f = f)

    eps = 1e-10
    T_A = np.clip(T_A, 0+eps, 1-eps)
    T_B = np.clip(T_B, 0+eps, 1-eps)

    if u == 0:
        Tu = T_A

    if u == 1:
        Tu = T_B

    for t in range(len(stimuli[:,0])):
        log_likelihood += threats[t, u] * np.log(Tu[t]) + (1 - threats[t, u]) * np.log(1 - Tu[t])

    nll = -log_likelihood
```

```python

    return nll

### FITTING CONTEXT A DAY 6 ### UNSTRESSED

# Minimization for MICE
for m in range(0, len(unstressed_mice_details)):
    # Setting up initial starting point for threat
    a_prev, g1_prev, g1_prev, f_prev = TD_Mom_unstressed_results["Day 1 Context A Params"][m]
    initial_prev = [0,0]
    B_shocks_prev = np.zeros_like(day1_unstressed_attacks[m:m + 1].reshape(-1))
    contexts_stacked_prev = np.column_stack((day1_unstressed_attacks[m:m + 1].reshape(-1), B_shocks_prev))

    initial_A = TD_momentum_model_init(init = initial_prev, contexts = contexts_stacked_prev,
                                                                    alpha=a, gamma1=g1, gamma2=g2, f=f)[0][-1]

    initial_B = TD_momentum_model_init(init = initial_prev, contexts = contexts_stacked_prev,
                                                                    alpha=a, gamma1=g1, gamma2=g2, f=f)[1][-1]

    # Compute TD Mom threat fit from parameters for this day, compute NLL
    A_shocks = np.zeros_like(day6_unstressed_attacks_B[m:m + 1].reshape(-1))
    contexts_stacked = np.column_stack((A_shocks, day6_unstressed_attacks_B[m:m + 1].reshape(-1)))
    initial = [initial_A, initial_B]

    a, g1, g2, f = TD_Mom_unstressed_results["Day 6 Context B Params"][m]
    day6_unstressed_freezing_A = TD_momentum_model_init(init = initial, contexts = contexts_stacked,
                                                                    alpha=a, gamma1=g1, gamma2=g2, f=f)[0]

    params = [a, g1, g2, f]
```

```
    threats_stacked = np.column_stack((day6_unstressed_freezing_A,
   day6_unstressed_freezing[m]))
    nll = NLL_td_mom(params=params, init=initial, stimuli=
   contexts_stacked, threats=threats_stacked, u=0)
    # Store in results dataframe:
    TD_Mom_unstressed_results["Day 6 Context A NLL"][m] = nll










# Adjust NLL for day 7 - remove u param
def NLL_td_mom(params, init, stimuli, threats):
    alpha, gamma1, gamma2, f = params
    log_likelihood = 0.0

    T_A, T_B = TD_momentum_model_init(init = init, contexts =
   stimuli, alpha = alpha, gamma1 = gamma1, gamma2 = gamma2, f = f)

    eps = 1e-10
    T_A = np.clip(T_A, 0+eps, 1-eps)
    T_B = np.clip(T_B, 0+eps, 1-eps)


    Tu = T_B

    for t in range(len(stimuli[:,0])):
        log_likelihood += threats[t, 0] * np.log(Tu[t]) + (1 -
   threats[t, 0]) * np.log(1 - Tu[t])

    nll = -log_likelihood

    return nll

### FITTING CONTEXT B DAY 7 ### UNSTRESSED


```

```
initial_vals = np.array((0.5, 0.95, 0.4, 1.5))
bounds = [(0.05, 1), (0.9, 1), (0, 0.8), (0, 3)]

# Minimization for MICE
for m in range(0, len(unstressed_mice_details)):
    # Setting up initial starting point for threat
    a_prev_prev, g1_prev_prev, g2_prev_prev, f_prev_prev = 
    TD_Mom_unstressed_results["Day 1 Context A Params"][m]
    initial_prev_prev = [0,0]
    B_shocks_prev_prev = np.zeros_like(day1_unstressed_attacks[m:m + 
    1].reshape(-1))
    contexts_stacked_prev_prev = np.column_stack((
    day1_unstressed_attacks[m:m + 1].reshape(-1), B_shocks_prev_prev)
    )

    initial_prev_A = TD_momentum_model_init(init = initial_prev_prev
    , contexts = contexts_stacked_prev_prev,
                                                            alpha=
    a_prev_prev, gamma1=g1_prev_prev,
                                            gamma2=g2_prev_prev, f=
    f_prev_prev)[0][-1]

    initial_prev_B = TD_momentum_model_init(init = initial_prev_prev
    , contexts = contexts_stacked_prev_prev,
                                                            alpha=
    a_prev_prev, gamma1=g1_prev_prev,
                                            gamma2=g2_prev_prev, f=
    f_prev_prev)[1][-1]


    initial_prev = [initial_prev_A, initial_prev_B]
    a_prev, g1_prev, g2_prev, f_prev = TD_Mom_unstressed_results["
    Day 6 Context B Params"][m]
    A_shocks_prev = np.zeros_like(day6_unstressed_attacks_B[m:m + 
    1].reshape(-1))
    contexts_stacked_prev = np.column_stack((A_shocks_prev, 
    day6_unstressed_attacks_B[m:m + 1].reshape(-1)))

    initial_A = TD_momentum_model_init(init=initial_prev, contexts=
    contexts_stacked_prev,
                                                alpha=a_prev, gamma1=
    g1_prev,
```

```
                                                       gamma2=g2_prev, f=f_prev)
[0][-1]

    initial_B = TD_momentum_model_init(init=initial_prev, contexts=
contexts_stacked_prev,
                                                       alpha=a_prev, gamma1=
g1_prev,
                                                       gamma2=g2_prev, f=f_prev)
[1][-1]

    # Compute minimization for each participant
    initial = [initial_A, initial_B]
    A_shocks = np.zeros_like(day7_unstressed_attacks_A[m:m + 1].
reshape(-1))     # doesn't matter if A or B, both filled with
zeros
    contexts_stacked = np.column_stack((A_shocks,
day7_unstressed_attacks_B[m:m + 1].reshape(-1)))

    result_B = differential_evolution(NLL_td_mom, x0=initial_vals,
bounds=bounds,
                                  args=(initial, contexts_stacked,
day7_unstressed_freezing[m:m + 1].reshape((18000, 1))),
                                            polish=True)

    print(f'B: n_iter: {result_B.nit} - success: {result_B.success}
- nll {result_B.fun}')

    # Store in results dataframe:
    TD_Mom_unstressed_results["Day 7 Context B NLL"][m] = result_B.
fun
    alpha, gamma1, gamma2, f = result_B.x
    TD_Mom_unstressed_results["Day 7 Context A Params"][m] = [alpha,
 gamma1, gamma2, f]






# Back to original NLL for day 7 context A
def NLL_td_mom(params, init, stimuli, threats, u):
    alpha, gamma1, gamma2, f = params
    log_likelihood = 0.0
```

```python

    T_A, T_B = TD_momentum_model_init(init = init, contexts = stimuli, alpha = alpha, gamma1 = gamma1, gamma2 = gamma2, f = f)

    eps = 1e-10
    T_A = np.clip(T_A, 0+eps, 1-eps)
    T_B = np.clip(T_B, 0+eps, 1-eps)

    if u == 0:
        Tu = T_A

    if u == 1:
        Tu = T_B

    for t in range(len(stimuli[:,0])):
        log_likelihood += threats[t, u] * np.log(Tu[t]) + (1 - threats[t, u]) * np.log(1 - Tu[t])

    nll = -log_likelihood

    return nll
### FITTING CONTEXT A DAY 7 ### UNSTRESSED

for m in range(0, len(unstressed_mice_details)):
    # Setting up initial starting point for threat
    a_prev_prev, g1_prev_prev, g2_prev_prev, f_prev_prev = TD_Mom_unstressed_results["Day 1 Context A Params"][m]
    initial_prev_prev = [0, 0]
    B_shocks_prev_prev = np.zeros_like(day1_unstressed_attacks[m:m + 1].reshape(-1))
    contexts_stacked_prev_prev = np.column_stack((day1_unstressed_attacks[m:m + 1].reshape(-1), B_shocks_prev_prev))

    initial_prev_A = TD_momentum_model_init(init=initial_prev_prev, contexts=contexts_stacked_prev_prev,
                                            alpha=a_prev_prev, gamma1=g1_prev_prev,
                                            gamma2=g2_prev_prev, f=f_prev_prev)[0][-1]

    initial_prev_B = TD_momentum_model_init(init=initial_prev_prev,
```

```
contexts=contexts_stacked_prev_prev,
                                                   alpha=a_prev_prev, gamma1=
g1_prev_prev,
                                                   gamma2=g2_prev_prev, f=
f_prev_prev)[1][-1]

 a_prev, g1_prev, g2_prev, f_prev = TD_Mom_unstressed_results["
Day 6 Context B Params"][m]
 A_shocks_prev = np.zeros_like(day6_unstressed_attacks_B[m:m +
1].reshape(-1))
 contexts_stacked_prev = np.column_stack((A_shocks_prev,
day6_unstressed_attacks_B[m:m + 1].reshape(-1)))
 initial_prev = [initial_prev_A, initial_prev_B]

 initial_A = TD_momentum_model_init(init=initial_prev, contexts=
contexts_stacked_prev,
                                            alpha=a_prev, gamma1=g1_prev,
                                            gamma2=g2_prev, f=f_prev)
[0][-1]

 initial_B = TD_momentum_model_init(init=initial_prev, contexts=
contexts_stacked_prev,
                                            alpha=a_prev, gamma1=g1_prev,
                                            gamma2=g2_prev, f=f_prev)
[1][-1]

 # Compute minimization for each participant
 A_shocks = np.zeros_like(day7_unstressed_attacks_A[m:m + 1].
reshape(-1))
 contexts_stacked = np.column_stack((A_shocks,
day7_unstressed_attacks_B[m:m + 1].reshape(-1)))
 a, g1, g2, f = TD_Mom_unstressed_results["Day 7 Context A Params
"][m]

 initial = [initial_A, initial_B]

 day7_stressed_freezing_A = TD_momentum_model_init(init=initial,
contexts=contexts_stacked, alpha=a, gamma1=g1,
                                                         gamma2=g2, f=f
)[0]

 params = [a, g1, g2, f]
```

```
    threats_stacked = np.column_stack((day7_unstressed_freezing_A,
   day7_unstressed_freezing[m]))
    nll = NLL_td_mom(params=params, init=initial, stimuli=
   contexts_stacked, threats=threats_stacked, u=0)
    # Store in results dataframe:
    TD_Mom_unstressed_results["Day 7 Context A NLL"][m] = nll
```

D.3 TD-Momentum Model - Computing BIC Scores

```
### BIC SCORES FOR TD Momentum MODEL ###

# BIC = 2 * NLL + p*log(n)

# where p = number of params, n = number of observations (162000
   day1 and 18000 for day 6 and 7)
p = 4

# Stressed BIC
for m in range(len(TD_Mom_stressed_results)):
    Day1_NLL = (TD_Mom_stressed_results["Day 1 Context A NLL"][m] +
                TD_Mom_stressed_results["Day 1 Context B NLL"][m])

    Day6_NLL = (TD_Mom_stressed_results["Day 6 Context A NLL"][m] +
                TD_Mom_stressed_results["Day 6 Context B NLL"][m])

    Day7_NLL = (TD_Mom_stressed_results["Day 7 Context A NLL"][m] +
                TD_Mom_stressed_results["Day 7 Context B NLL"][m])

    Day1_BIC = 2 * Day1_NLL + p * np.log(162000)

    Day6_BIC = 2 * Day6_NLL + p * np.log(18000)

    Day7_BIC = 2 * Day7_NLL + p * np.log(18000)

    TD_Mom_stressed_results["BIC Score"][m] = Day1_BIC + Day6_BIC +
   Day7_BIC

# Unstressed BIC
for m in range(len(TD_Mom_unstressed_results)):
    Day1_NLL = (TD_Mom_unstressed_results["Day 1 Context A NLL"][m]
   +
```

```
                  TD_Mom_unstressed_results["Day 1 Context B NLL"][m])

    Day6_NLL = (TD_Mom_unstressed_results["Day 6 Context A NLL"][m]
    +
                  TD_Mom_unstressed_results["Day 6 Context B NLL"][m])

    Day7_NLL = (TD_Mom_unstressed_results["Day 7 Context A NLL"][m]
    +
                  TD_Mom_unstressed_results["Day 7 Context B NLL"][m])

    Day1_BIC = 2 * Day1_NLL + p * np.log(162000)

    Day6_BIC = 2 * Day6_NLL + p * np.log(18000)

    Day7_BIC = 2 * Day7_NLL + p * np.log(18000)

    TD_Mom_unstressed_results["BIC Score"][m] = Day1_BIC + Day6_BIC
    + Day7_BIC
```

D.4 TD-Momentum Model - Multivariate Sampled Parameter Recovery

```
import matplotlib.pyplot as plt
import numpy as np
from scipy.optimize import minimize
import scipy
import pandas as pd
from scipy.optimize import differential_evolution


# Model Param lists (taken from paper supplementary material)
# Loading matlab file from paper - creates dict "mat_contents" which
     stores all variables
import scipy.io as sio
mat_contents = sio.loadmat('Supp_Mat')

def day_1(shocks):
    '''
    Evaluates day1 of SEFL experiment.

```

```
    :return: Outputs sequence of 15 shocks randomly over 90 mins (5400 secs)
    '''
    min_shock_int = 4 * 1800
    max_shock_int = 6 * 1800
    # 90 mins = 5400 secs
    day_1 = np.zeros(162000)

    all_shock_intervals = []
    while np.sum(day_1) < shocks:
        # Generate random interval between 4 and 8 mins
        shock_interval = np.random.randint(min_shock_int, max_shock_int)
        # Add this to the list of all interval times
        all_shock_intervals.append(shock_interval)
        # Calculate the shock index by summing cumulative intervals
        shock_index = np.sum(all_shock_intervals)
        # Add shock to day_1 array
        day_1[shock_index] = 1
    return day_1

# Combined scaling function for context A and B
def combined_rescale(arr1, arr2, new_min, new_max):
    combined_min = min(np.min(arr1), np.min(arr2))
    combined_max = max(np.max(arr1), np.max(arr2))
    rescaled_arr1 = (arr1 - combined_min) * (new_max - new_min) / (combined_max - combined_min + 1e-10) + new_min
    rescaled_arr2 = (arr2 - combined_min) * (new_max - new_min) / (combined_max - combined_min + 1e-10) + new_min
    return rescaled_arr1, rescaled_arr2

# Creating NLL function and performing parameter recovery for simulated data
def TD_momentum_model(contexts, alpha, gamma1, gamma2, f):
    '''
    Evaluates TD Momentum model threat at each timestep in each context only, no changing contexts, assumes the same
    agent in one contex for the whole time, creates threat in other context as momentum term only.

    :param contexts: Sequences of unconditioned stimuli across all contexts. e.g. sefl(sims = 2)[1] for 2nd mouse sim
```

```
    :param u: Key of chosen context to compute TD momentum model for. (day1, day6, day7)
    :param alpha: Learning rate
    :param gamma1: Decay rate for threat
    :param gamma2: Decay rate for momentum across all contexts
    :param f: Scaling Constant for momentum term

    :return: TD Momentum model threat estimation levels over all timesteps
    '''
    # Initialise momentum array
    m = np.zeros_like(contexts[:,0])

    T_A = np.zeros_like(contexts[:,0])
    T_B = np.zeros_like(contexts[:,1])


    for t in range(1, len(contexts[:,0])):
        # Set PE to 0 at every time step before computing PE for current step
        PE = 0
        PE += alpha * (contexts[:, 0][t] - T_A[t - 1])
        PE += alpha * (contexts[:, 1][t] - T_B[t - 1])

        m[t] = m[t-1] + gamma2 * PE
        m[t] = np.clip(m[t], 0, 1)

        T_A[t] = T_A[t - 1] + alpha * (contexts[:, 0][t] - gamma1 * T_A[t - 1]) + f * m[t]
        T_B[t] = T_B[t - 1] + alpha * (contexts[:, 1][t] - gamma1 * T_B[t - 1]) + f * m[t]

        T_A[t] = np.clip(T_A[t], 0, 1)
        T_B[t] = np.clip(T_B[t], 0, 1)

    #T_A, T_B = combined_rescale(T_A, T_B, 0.1, 0.9)

    return T_A, T_B


def NLL_td_mom(params, stimuli, threats, u):
    alpha, gamma1, gamma2, f = params
```

```
    log_likelihood = 0.0

    T_A, T_B = TD_momentum_model(contexts = stimuli, alpha = alpha, gamma1 = gamma1, gamma2 = gamma2, f = f)

    eps = 1e-10
    T_A = np.clip(T_A, 0+eps, 1-eps)
    T_B = np.clip(T_B, 0+eps, 1-eps)

    if u == 0:
        Tu = T_A

    if u == 1:
        Tu = T_B

    for t in range(len(stimuli[:,0])):
        log_likelihood += threats[t, u] * np.log(Tu[t]) + (1 - threats[t, u]) * np.log(1 - Tu[t])

    nll = -log_likelihood

    return nll


### PARAMETER RECOVERY FOR 100 MULTIVARIATE NORMAL SAMPLES PARAMETER SETS ### CONTEXT A ###
bounds = [(0.05,1), (0.9,1), (0,0.8), (0,3)]

alpha_mean = 0.5
gamma1_mean = 0.95
gamma2_mean = 0.45
f_mean = 1.5
means = [alpha_mean, gamma1_mean, gamma2_mean, f_mean]

alpha_var = 0.05
gamma1_var = 0.001
gamma2_var = 0.05
f_var = 0.5

covs = np.zeros((4,4))
covs[0,0] = alpha_var
covs[1,1] = gamma1_var
```

```
covs[2,2] = gamma2_var
covs[3,3] = f_var

sample_params = np.random.multivariate_normal(mean=means, cov=covs, size=100)
sample_params[:,0] = np.clip(sample_params[:,0], 0.05, 0.9)
sample_params[:,1] = np.clip(sample_params[:,1], 0.9, 1)
sample_params[:,2] = np.clip(sample_params[:,2], 0, 0.8)
sample_params[:,3] = np.clip(sample_params[:,3], 0, 3)

'''
# Plot of param values
fig, ax = plt.subplots(4)
fig.suptitle("Sampled Parameter Values")
# add a big axes, hide frame
fig.add_subplot(111, frameon=False)
# hide tick and tick label of the big axes
plt.tick_params(labelcolor='none', top=False, bottom=False, left=False, right=False)
plt.grid(False)
x = np.linspace(0, 100, 100)
ax[0].plot(x, sample_params[:,0], "r+", label="Sampled $alpha$")
ax[1].plot(x, sample_params[:,1], "bo", label="Sampled $gamma_1$")
ax[2].plot(x, sample_params[:,2], "g*", label="Sampled $gamma_2$")
ax[3].plot(x, sample_params[:,3], "+", label="Sampled $f$")
plt.xlabel("Simulation (Individual Mouse)")
plt.ylabel("Sampled Parameter Value")
ax[0].legend(fontsize="8")
ax[1].legend(fontsize="8")
ax[2].legend(fontsize="8")
ax[2].legend(fontsize="8")
plt.show()
'''

# Calculate the correlation matrix - check for correlation between values
sampled_corr_matrix = np.corrcoef(sample_params, rowvar=False)
print(sampled_corr_matrix)

# Now create simulations of attacks and corresponding threat predictions for each set of params
sampled_attacks = np.zeros((100, 5400))
```

```
sampled_predictions = np.zeros((100, 5400))

for index, (a, g1, g2, f) in enumerate(zip(sample_params[:,0], sample_params[:,1], sample_params[:,2], sample_params[:,3])):
    # Generate day 1 sequence of attacks for each
    A_shocks = day_1(15)
    B_shocks = np.zeros_like(A_shocks)
    # Stack for input to TD-Mom Model
    contexts_stacked = np.column_stack((A_shocks, B_shocks))

    # Generate threat predictions from this simulated sequence of attacks
    predictions = TD_momentum_model(contexts = contexts_stacked, alpha = a, gamma1 = g1, gamma2 = g2, f = f)[0]

    # Add to sampled attacks/preds arrays
    sampled_attacks[index, :] = A_shocks
    sampled_predictions[index, :] = predictions

np.sum(sampled_predictions, axis=1)
#np.where(np.sum(sampled_predictions, axis=1)==0)


# Fitting parameter values to these simulations (see if we can recover params)
initial_vals = np.array((0.5, 0.95, 0.4, 1.5))

bounds = [(0.05, 1), (0.9, 1), (0, 0.8), (0, 3)]

opt_sampled_data = np.zeros((len(sampled_predictions[:,0]), 4))

# Minimization for simulations (Mice)
for m in range(0, len(sampled_predictions[:,0])):
    # Compute minimization for each participant
    B_shocks = np.zeros_like(sampled_attacks[m:m + 1].reshape(-1))
    contexts_stacked = np.column_stack((sampled_attacks[m:m + 1].reshape(-1), B_shocks))

    result = differential_evolution(NLL_td_mom, x0=initial_vals, bounds=bounds,
                                    args=(contexts_stacked, sampled_predictions[m:m + 1].reshape((5400, 1)), 0),
```

```
                                    strategy='best1bin', polish=True
    )

     print(f'n_iter: {result.nit} - success: {result.success} - nll {
    result.fun}')

     alpha, gamma1, gamma2, f= result.x
     # Add results to opt_params storing opt param values for each
    participant
     opt_sampled_data[m, 0] = alpha
     opt_sampled_data[m, 1] = gamma1
     opt_sampled_data[m, 2] = gamma2
     opt_sampled_data[m, 3] = f

#Alpha params
multivariate_sim_alpha = sample_params[:,0]
fitted_simulated_alpha = opt_sampled_data[:,0]
alpha_pearson = scipy.stats.pearsonr(multivariate_sim_alpha,
    fitted_simulated_alpha)

# gamma1 params
multivariate_sim_gamma1 = sample_params[:,1]
fitted_simulated_gamma1 = opt_sampled_data[:,1]
gamma1_pearson = scipy.stats.pearsonr(multivariate_sim_gamma1,
    fitted_simulated_gamma1)

# gamma2 params
multivariate_sim_gamma2 = sample_params[:,2]
fitted_simulated_gamma2 = opt_sampled_data[:,2]
gamma2_pearson = scipy.stats.pearsonr(multivariate_sim_gamma2,
    fitted_simulated_gamma2)

# f params
multivariate_sim_f = sample_params[:,3]
fitted_simulated_f = opt_sampled_data[:,3]
f_pearson = scipy.stats.pearsonr(multivariate_sim_f,
    fitted_simulated_f)
```

D.5 TD-Momentum Model - List Sampled Parameter Recovery

```
### DIFFERENTIAL EVOLUTION - LIST PARAMS ###
alpha_list = mat_contents["learning_rate_list"][0]
gamma1_list = mat_contents["decay_list"][0]
gamma2_list = mat_contents["momentum_rate_list"][0]
f_list = -mat_contents["scaling_list"][0]

sample_params = np.zeros((100, 4))

for s in range(100):
    sample_params[s, 0] = np.random.choice(alpha_list)
    sample_params[s, 1] = np.random.choice(gamma1_list)
    sample_params[s, 2] = np.random.choice(gamma2_list)
    sample_params[s, 3] = np.random.choice(f_list)

corr_matrix = np.corrcoef(sample_params, rowvar=False)
print(corr_matrix)

# Now create simulations of attacks and corresponding threat
    predictions for each set of params

sampled_attacks = np.zeros((100, 5400))
sampled_predictions = np.zeros((100, 5400))

for index, (a, g1, g2, f) in enumerate(zip(sample_params[:,0],
    sample_params[:,1], sample_params[:,2], sample_params[:,3])):
    # Generate day 1 sequence of attacks for each
    A_shocks = day_1(15)
    B_shocks = np.zeros_like(A_shocks)
    # Stack for input to TD-Mom Model
    contexts_stacked = np.column_stack((A_shocks, B_shocks))

    # Generate threat predictions from this simulated sequence of
    attacks
    predictions = TD_momentum_model(contexts = contexts_stacked,
    alpha = a, gamma1 = g1, gamma2 = g2, f = f)[0]

    # Add to sampled attacks/preds arrays
    sampled_attacks[index, :] = A_shocks
```

```
    sampled_predictions[index, :] = predictions

np.sum(sampled_predictions, axis=1)
# Low alpha and high f give large threat estimates

# Fitting parameter values to these simulations (see if we can
   recover params)

from scipy.optimize import differential_evolution
initial_vals = np.array((0.5, 0.95, 0.4, 1.5))

bounds = [(0.05, 1), (0.9, 1), (0, 0.8), (0, 3)]

opt_sampled_data = np.zeros((len(sampled_predictions[:,0]), 4))
# Minimization for simulations (Mice)
for m in range(0, len(sampled_predictions[:,0])):
    # Compute minimization for each participant
    B_shocks = np.zeros_like(sampled_attacks[m:m + 1].reshape(-1))
    contexts_stacked = np.column_stack((sampled_attacks[m:m + 1].
   reshape(-1), B_shocks))

    result = differential_evolution(NLL_td_mom, x0=initial_vals,
   bounds=bounds,
                                    args=(contexts_stacked,
   sampled_predictions[m:m + 1].reshape((5400, 1)), 0),
                                    strategy='best1bin', polish=True
   )

    print(f'n_iter: {result.nit} - success: {result.success} - nll {
   result.fun}')
    alpha, gamma1, gamma2, f= result.x
    # Add results to opt_params storing opt param values for each
   participant
    opt_sampled_data[m, 0] = alpha
    opt_sampled_data[m, 1] = gamma1
    opt_sampled_data[m, 2] = gamma2
    opt_sampled_data[m, 3] = f

#Alpha params
selected_sim_alpha = sample_params[:,0]
fitted_simulated_alpha = opt_sampled_data[:,0]
alpha_pearson = scipy.stats.pearsonr(selected_sim_alpha,
```

```
    fitted_simulated_alpha)

# gamma1 params
selected_sim_gamma1 = sample_params[:,1]
fitted_simulated_gamma1 = opt_sampled_data[:,1]
gamma1_pearson = scipy.stats.pearsonr(selected_sim_gamma1,
    fitted_simulated_gamma1)

# gamma2 params
selected_sim_gamma2 = sample_params[:,2]
fitted_simulated_gamma2 = opt_sampled_data[:,2]
gamma2_pearson = scipy.stats.pearsonr(selected_sim_gamma2,
    fitted_simulated_gamma2)

# f params
selected_sim_f = sample_params[:,3]
fitted_simulated_f = opt_sampled_data[:,3]
f_pearson = scipy.stats.pearsonr(selected_sim_f, fitted_simulated_f)
```

D.6 BIC Model Comparison

```
# Unstressed model comparison: BIC_TD - BIC_TD-MOM

unstressed_comparison = pd.DataFrame({
    "BIC_TD - BIC_TD-MOM": [None] * len(TD_Mom_unstressed_results)})

unstressed_comparison["BIC_TD - BIC_TD-MOM"] = TD_unstressed_results
    ["BIC Score"] - TD_Mom_unstressed_results["BIC Score"]


# Stressed model comparison: BIC_TD - BIC_TD-MOM
stressed_comparison = pd.DataFrame({
    "BIC_TD - BIC_TD-MOM": [None] * len(TD_Mom_stressed_results)})

stressed_comparison["BIC_TD - BIC_TD-MOM"] = TD_stressed_results["
    BIC Score"] - TD_Mom_stressed_results["BIC Score"]

stressed_comparison["BIC_TD - BIC_TD-MOM"] = stressed_comparison["
    BIC_TD - BIC_TD-MOM"]*(-1)


```

```
### Re-creating fig 5D: Model Comparison Plot
# Create a scatter plot
plt.figure(figsize=(8, 6))

# Plot unstressed_comparison on the top
plt.scatter(unstressed_comparison, range(len(unstressed_comparison))
    , color='blue', label='Unstressed')

# Plot stressed_comparison on the bottom
plt.scatter(stressed_comparison, range(len(stressed_comparison)),
    color='red', label='Stressed')

plt.axvline(x=0, color='black', linestyle='-')
plt.axhline(y=-1, color='black', linestyle='-')
plt.xlabel('$BIC_{TD} - BIC_{TD\ Momentum}$', fontsize=15)
plt.yticks([])
plt.title('Stressed vs. Unstressed Mice Model Comparison', fontsize
    =15)
plt.legend()

# Add arrows with text annotations
plt.annotate('Favours TD Momentum', xy=(87, -1.7), xytext=(1, -2),
             arrowprops=dict(facecolor='black', arrowstyle='simple')
    , fontsize=15)

plt.annotate('Favours TD', xy=(-53, -1.7), xytext=(-35, -2),
             arrowprops=dict(facecolor='black', arrowstyle='simple')
    , fontsize=15)
plt.tight_layout()
plt.show()
```

Appendix E

Extension Simulations Code

E.1 Associability TD-Momentum Model

```
import numpy as np
import matplotlib.pyplot as plt

def ASSOCIABILITY_TD_MOM(init, contexts, alpha, gamma1, gamma2, f,
    eta):
     '''
     Evaluates TD Momentum model threat at each timestep in each
    context only, no changing contexts, assumes the same
     agent in one contex for the whole time, creates threat in other
    context as momentum term only.

     :param init: Provides initialisation points for threat readings
    in both contexts (days 6 and 7)
     :param contexts: Sequences of unconditioned stimuli across all
    contexts. e.g. sefl(sims = 2)[1] for 2nd mouse sim
     :param u: Key of chosen context to compute TD momentum model for
    . (day1, day6, day7)
     :param alpha: Learning rate
     :param gamma1: Decay rate for threat
     :param gamma2: Decay rate for momentum across all contexts
     :param f: Scaling Constant for momentum term
     :param eta: Associability weight parameter (between 0 and 1)

     :return: TD Momentum model threat estimation levels over all
    timesteps
     '''
```

```
    # Initialise momentum array
    m = np.zeros_like(contexts[:,0])

    T_A = np.zeros_like(contexts[:,0])
    T_B = np.zeros_like(contexts[:,1])
    Kappa_A = np.zeros_like(T_B)
    Kappa_B = np.zeros_like(T_B)

    # Set initialisation
    T_A[0] = init[0]
    T_B[0] = init[1]
    Kappa_A[0] = 1
    Kappa_B[0] = 1


    for t in range(1, len(contexts[:,0])):
        # Set PE to 0 at every time step before computing PE for
current step
        PE = 0
        PE += alpha * (contexts[:, 0][t] - T_A[t - 1])
        PE += alpha * (contexts[:, 1][t] - T_B[t - 1])

        m[t] = m[t-1] + gamma2 * PE
        m[t] = np.clip(m[t], 0, 1)

        Kappa_A[t] = (1 - eta) * Kappa_A[t-1] + eta * (np.abs(
contexts[:, 0][t] - gamma1 * T_A[t - 1]))
        Kappa_B[t] = (1 - eta) * Kappa_B[t-1] + eta * (np.abs(
contexts[:, 1][t] - gamma1 * T_B[t - 1]))

        # Set lower bound constraint for associability values
        if Kappa_A[t] < 0.05:
            Kappa_A[t] = 0.05

        if Kappa_B[t] < 0.05:
            Kappa_B[t] = 0.05

        T_A[t] = T_A[t - 1] + alpha * Kappa_A[t-1] * (contexts[:,
0][t] - gamma1 * T_A[t - 1]) + f * m[t]
        T_B[t] = T_B[t - 1] + alpha * Kappa_B[t-1] * (contexts[:,
1][t] - gamma1 * T_B[t - 1]) + f * m[t]

```

```
        T_A[t] = np.clip(T_A[t], 0, 1)
        T_B[t] = np.clip(T_B[t], 0, 1)

    #T_A, T_B = combined_rescale(T_A, T_B, 0, 1)
    #Kappa_A, Kappa_B = combined_rescale(Kappa_A, Kappa_B, 0, 1)

    return T_A, T_B, Kappa_A, Kappa_B



# Compare to this !
def TD_momentum_model(contexts, alpha, gamma1, gamma2, f):
    '''
    Evaluates TD Momentum model threat at each timestep in each
    context only, no changing contexts, assumes the same
    agent in one contex for the whole time, creates threat in other
    context as momentum term only.

    :param contexts: Sequences of unconditioned stimuli across all
    contexts. e.g. sefl(sims = 2)[1] for 2nd mouse sim
    :param u: Key of chosen context to compute TD momentum model for
    . (day1, day6, day7)
    :param alpha: Learning rate
    :param gamma1: Decay rate for threat
    :param gamma2: Decay rate for momentum across all contexts
    :param f: Scaling Constant for momentum term

    :return: TD Momentum model threat estimation levels over all
    timesteps
    '''
    # Initialise momentum array
    m = np.zeros_like(contexts[:,0])

    T_A = np.zeros_like(contexts[:,0])
    T_B = np.zeros_like(contexts[:,1])


    for t in range(1, len(contexts[:,0])):
        # Set PE to 0 at every time step before computing PE for
    current step
        PE = 0
        PE += alpha * (contexts[:, 0][t] - T_A[t - 1])
```

```
        PE += alpha * (contexts[:, 1][t] - T_B[t - 1])

        m[t] = m[t-1] + gamma2 * PE
        m[t] = np.clip(m[t], 0, 1)    # NEWLY ADDED 25/07/23

        T_A[t] = T_A[t - 1] + alpha * (contexts[:, 0][t] - gamma1 * T_A[t - 1]) + f * m[t]
        T_B[t] = T_B[t - 1] + alpha * (contexts[:, 1][t] - gamma1 * T_B[t - 1]) + f * m[t]

        T_A[t] = np.clip(T_A[t], 0, 1)
        T_B[t] = np.clip(T_B[t], 0, 1)

    #T_A, T_B = combined_rescale(T_A, T_B, 0, 1)    # NEWLY ADDED 25/07/23

    return T_A, T_B



# Create contexts attack sequences
#A = day_1(15)

A = np.zeros(1000)
# Insert 25 random 1s at random positions in the array
random_indices = np.random.choice(1000, 10, replace=False)
A[random_indices] = 1

B = np.zeros_like(A)
# Stack for input to TD-Mom Model
contexts_stacked = np.column_stack((A, B))
initial = [0, 0]

#x = np.linspace(0, 90, 162000)

x = np.linspace(0, 10, 1000)

y1, y2, y3, y4 = ASSOCIABILITY_TD_MOM(init = initial, contexts = contexts_stacked, alpha =0.05, gamma1 = 0.9999,
                            gamma2 = 0.05, f = 0.1, eta = 0.01)


```

```
y5, y6 = TD_momentum_model(contexts = contexts_stacked, alpha =0.05,
    gamma1 = 0.9999,
                        gamma2 = 0.05, f = 0.1)


fig, axes = plt.subplots(1, 3, figsize=(18, 6))

# Plot for the first graph
axes[1].plot(x, y1, label="A")
axes[1].plot(x, y2, label="B")
axes[1].plot(x, A, label="A input", alpha=0.20)
axes[1].plot(x, B, label="B input", alpha=0.20)
axes[1].set_title("Associability TD-Momentum Simulation ($\eta =
    0.01$)", fontsize=15)
axes[1].set_xlabel("Time", fontsize=15)
axes[1].set_ylabel("Threat", fontsize=15)
axes[1].set_ylim(0, np.max(y1))
axes[1].legend(loc="upper right")

# Plot for the second graph
axes[2].plot(x, y3, label="$\kappa_A$")
axes[2].plot(x, y4, label="$\kappa_B$")
axes[2].plot(x, A, label="A input", alpha=0.20)
axes[2].plot(x, B, label="B input", alpha=0.20)
axes[2].set_title("Associability Values", fontsize=15)
axes[2].set_xlabel("Time", fontsize=15)
axes[2].set_ylabel("Associability", fontsize=15)
axes[2].set_ylim(0, np.max(y3))
axes[2].legend(loc="upper right")

# Plot for the third graph
axes[0].plot(x, y5, label="A")
axes[0].plot(x, y6, label="B")
axes[0].plot(x, A, label="A input", alpha=0.20)
axes[0].plot(x, B, label="B input", alpha=0.20)
axes[0].set_xlabel("Time", fontsize=15)
axes[0].set_ylabel("Threat", fontsize=15)
axes[0].set_title("TD-Momentum Simulation", fontsize=15)
axes[0].set_ylim(0, np.max(y5))
axes[0].legend(loc="upper right")

plt.tight_layout()
```

```
plt.show()
```

E.2 Risk-Sensitive TD-Momentum Model

```
import numpy as np
import matplotlib.pyplot as plt

def RISK_SENSITIVE_TD_MOM(init, contexts, alpha_pos, alpha_neg,
    gamma1, gamma2, f):
    '''
    Evaluates TD Momentum model threat at each timestep in each
    context only, no changing contexts, assumes the same
    agent in one contex for the whole time, creates threat in other
    context as momentum term only.

    :param init: Provides initialisation points for threat readings
    in both contexts (days 6 and 7)
    :param contexts: Sequences of unconditioned stimuli across all
    contexts. e.g. sefl(sims = 2)[1] for 2nd mouse sim
    :param alpha_pos: Learning rate for positive prediction error
    :param alpha_neg: Learning rate for negative prediction error
    :param gamma1: Decay rate for threat
    :param gamma2: Decay rate for momentum across all contexts
    :param f: Scaling Constant for momentum term:

    :return: TD Momentum model threat estimation levels over all
    timesteps
    '''
    # Initialise momentum array
    m = np.zeros_like(contexts[:,0])

    T_A = np.zeros_like(contexts[:,0])
    T_B = np.zeros_like(contexts[:,1])

    # Set initialisation
    T_A[0] = init[0]
    T_B[0] = init[1]

    for t in range(1, len(contexts[:,0])):
        # Selecting learning rate for time step:
        # For context A:
```

```
        if (contexts[:, 0][t] - gamma1 * T_A[t - 1]) < 0:
            alpha_A = alpha_neg

        if (contexts[:, 0][t] - gamma1 * T_A[t - 1]) >= 0:
            alpha_A = alpha_pos

        # For context B:
        if (contexts[:, 1][t] - gamma1 * T_B[t - 1]) < 0:
            alpha_B = alpha_neg

        if (contexts[:, 1][t] - gamma1 * T_B[t - 1]) >= 0:
            alpha_B = alpha_pos


        # Set PE to 0 at every time step before computing PE for current step
        PE = 0
        PE += alpha_A * (contexts[:, 0][t] - gamma1 * T_A[t - 1])
        PE += alpha_B * (contexts[:, 1][t] - gamma1 * T_B[t - 1])

        m[t] = m[t-1] + gamma2 * PE
        m[t] = np.clip(m[t], 0, 1)


        T_A[t] = T_A[t - 1] + alpha_A * (contexts[:, 0][t] - gamma1 * T_A[t - 1]) + f * m[t]
        T_B[t] = T_B[t - 1] + alpha_B * (contexts[:, 1][t] - gamma1 * T_B[t - 1]) + f * m[t]

        T_A[t] = np.clip(T_A[t], 0, 1)
        T_B[t] = np.clip(T_B[t], 0, 1)

    #T_A, T_B = combined_rescale(T_A, T_B, 0, 1)

    return T_A, T_B



# Compare to this !
def TD_momentum_model(contexts, alpha, gamma1, gamma2, f):
    '''
    Evaluates TD Momentum model threat at each timestep in each
```

```
    context only, no changing contexts, assumes the same
    agent in one contex for the whole time, creates threat in other context as momentum term only.

    :param contexts: Sequences of unconditioned stimuli across all contexts. e.g. sefl(sims = 2)[1] for 2nd mouse sim
    :param u: Key of chosen context to compute TD momentum model for. (day1, day6, day7)
    :param alpha: Learning rate
    :param gamma1: Decay rate for threat
    :param gamma2: Decay rate for momentum across all contexts
    :param f: Scaling Constant for momentum term

    :return: TD Momentum model threat estimation levels over all timesteps
    '''
    # Initialise momentum array
    m = np.zeros_like(contexts[:,0])

    T_A = np.zeros_like(contexts[:,0])
    T_B = np.zeros_like(contexts[:,1])


    for t in range(1, len(contexts[:,0])):
        # Set PE to 0 at every time step before computing PE for current step
        PE = 0
        PE += alpha * (contexts[:, 0][t] - T_A[t - 1])
        PE += alpha * (contexts[:, 1][t] - T_B[t - 1])

        m[t] = m[t-1] + gamma2 * PE
        m[t] = np.clip(m[t], 0, 1)    # NEWLY ADDED 25/07/23

        T_A[t] = T_A[t - 1] + alpha * (contexts[:, 0][t] - gamma1 * T_A[t - 1]) + f * m[t]
        T_B[t] = T_B[t - 1] + alpha * (contexts[:, 1][t] - gamma1 * T_B[t - 1]) + f * m[t]

        T_A[t] = np.clip(T_A[t], 0, 1)
        T_B[t] = np.clip(T_B[t], 0, 1)

    #T_A, T_B = combined_rescale(T_A, T_B, 0, 1)   # NEWLY ADDED
```

```
    25/07/23

    return T_A, T_B



# Create context attack sequences
A = np.zeros(1000)
# Insert 25 random 1s at random positions in the array
random_indices = np.random.choice(1000, 10, replace=False)
A[random_indices] = 1

B = np.zeros_like(A)
# Stack for input to TD-Mom Model
contexts_stacked = np.column_stack((A, B))
initial = [0, 0]

#x = np.linspace(0, 90, 162000)

x = np.linspace(0, 10, 1000)

y1, y2 = RISK_SENSITIVE_TD_MOM(init = initial, contexts =
    contexts_stacked, alpha_pos = 0.05, alpha_neg = 0.2, gamma1 =
    0.9999,
                          gamma2 = 0.05, f = 0.1)


y3, y4 = TD_momentum_model(contexts = contexts_stacked, alpha =0.05,
     gamma1 = 0.9999,
                          gamma2 = 0.05, f = 0.1)


fig, axes = plt.subplots(1, 2, figsize=(12, 6))

# Plot for the first graph
axes[0].plot(x, y3, label="A")
axes[0].plot(x, y4, label="B")
axes[0].plot(x, A, label="A input", alpha=0.20)
axes[0].plot(x, B, label="B input", alpha=0.20)
axes[0].set_xlabel("Time", fontsize=15)
axes[0].set_ylabel("Threat", fontsize=15)
axes[0].set_title("TD-Momentum Simulation", fontsize=15)
```

```python
axes[0].set_ylim(0, np.max(y3))
axes[0].legend(loc="upper right")

# Plot for the second graph
axes[1].plot(x, y1, label="A")
axes[1].plot(x, y2, label="B")
axes[1].plot(x, A, label="A input", alpha=0.20)
axes[1].plot(x, B, label="B input", alpha=0.20)
axes[1].set_title("Risk-Sensitive TD-Momentum Simulation", fontsize
    =15)
axes[1].set_xlabel("Time", fontsize=15)
axes[1].set_ylabel("Threat", fontsize=15)
axes[1].set_ylim(0, np.max(y1))
axes[1].legend(loc="upper right")

plt.tight_layout()
plt.show()
```

E.3 Valence-Partitioned TD-Momentum Model

```python
import numpy as np
import matplotlib.pyplot as plt


def VP_TD_MOM(init, contexts, alpha_pos, alpha_neg, gamma1, gamma2,
    f):
    '''
    Evaluates TD Momentum model threat at each timestep in each
    context only, no changing contexts, assumes the same
    agent in one contex for the whole time, creates threat in other
    context as momentum term only.

    :param init: Provides initialisation points for threat readings
    in both contexts (days 6 and 7)
    :param contexts: Sequences of unconditioned stimuli across all
    contexts. e.g. sefl(sims = 2)[1] for 2nd mouse sim
    :param alpha_pos: Learning rate for positively valenced outcomes
    :param alpha_neg: Learning rate for negatively valenced outcomes
    :param gamma1: Decay rate for threat
    :param gamma2: Decay rate for momentum across all contexts
    :param f: Scaling Constant for momentum term:
```

```

    :return: TD Momentum model threat estimation levels over all
    timesteps
    '''
    # Initialise momentum arrays
    m_P = np.zeros_like(contexts[:,0])
    m_N = np.zeros_like(m_P)

    # Overall Threat in each context
    T_A = np.zeros_like(contexts[:,0])
    T_B = np.zeros_like(contexts[:,1])

    # Partitioned values for each context
    VA_P = np.zeros_like(T_A)
    VA_N = np.zeros_like(T_A)

    VB_P = np.zeros_like(T_B)
    VB_N = np.zeros_like(T_B)

    # Set initialisation
    T_A[0] = init[0]
    T_B[0] = init[1]

    for t in range(1, len(contexts[:,0])):
        # Partitioning based on outcome for each context:
        # For context A:
        if contexts[:, 0][t] > 0:
            A_delta_P = contexts[:, 0][t] - gamma1 * VA_P[t - 1]

        if contexts[:, 0][t] <= 0:
            A_delta_P = 0 - gamma1 * VA_P[t - 1]

        if contexts[:, 0][t] < 0:
            A_delta_N = np.abs(contexts[:, 0][t]) - gamma1 * VA_N[t
- 1]

        if contexts[:, 0][t] >= 0:
            A_delta_N = 0 - gamma1 * VA_N[t - 1]



        # For context B:
        if contexts[:, 1][t] > 0:
```

```
        B_delta_P = contexts[:, 1][t] - gamma1 * VB_P[t - 1]

    if contexts[:, 1][t] <= 0:
        B_delta_P = 0 - gamma1 * VB_P[t - 1]

    if contexts[:, 1][t] < 0:
        B_delta_N = np.abs(contexts[:, 1][t]) - gamma1 * VB_N[t - 1]

    if contexts[:, 1][t] >= 0:
        B_delta_N = 0 - gamma1 * VB_N[t - 1]



    # Set PE to 0 at every time step before computing PE for current step
    PE_P = 0
    PE_P += alpha_pos * A_delta_P
    PE_P += alpha_pos * B_delta_P

    m_P[t] = m_P[t-1] + gamma2 * PE_P
    #m_P[t] = np.clip(m[t], 0, 1)  DO WE NEED THESE? BETWEEN -1 AND 1?


    PE_N = 0
    PE_N += alpha_neg * A_delta_N
    PE_N += alpha_neg * B_delta_N

    m_N[t] = m_N[t-1] + gamma2 * PE_N
    #m_P[t] = np.clip(m[t], 0, 1)  DO WE NEED THESE? BETWEEN -1 AND 1?


    # Positive and negative value functions for each context
    VA_P[t] = VA_P[t - 1] + alpha_pos * A_delta_P + f * m_P[t]
    VA_N[t] = VA_N[t - 1] + alpha_neg * A_delta_N + f * m_N[t]

    VB_P[t] = VB_P[t - 1] + alpha_pos * B_delta_P + f * m_P[t]
    VB_N[t] = VB_N[t - 1] + alpha_neg * B_delta_N + f * m_N[t]

    # Generating overall threat prediction for each context
```

```
        T_A[t] = VA_P[t] - VA_N[t]
        T_B[t] = VB_P[t] - VB_N[t]

        #T_A[t] = np.clip(T_A[t], 0, 1)    DO WE NEED THESE? BETWEEN
    -1 AND 1?
        #T_B[t] = np.clip(T_B[t], 0, 1)    DO WE NEED THESE? BETWEEN
    -1 AND 1?

    #T_A, T_B = combined_rescale(T_A, T_B, 0, 1)

    return T_A, T_B, VA_P, VA_N, VB_P, VB_N, m_P, m_N


# Create contexts attack sequences
A = np.zeros(1000)
# Insert 10 random 1s at random positions in the array
random_indices1 = np.random.choice(1000, 5, replace=False)
A[random_indices1] = 1

random_indices2 = np.random.choice(1000, 5, replace=False)
A[random_indices2] = 0.5

random_indices3 = np.random.choice(1000, 5, replace=False)
A[random_indices3] = -1

random_indices4 = np.random.choice(1000, 5, replace=False)
A[random_indices4] = -0.5


B = np.zeros_like(A)
# Stack for input to TD-Mom Model
contexts_stacked = np.column_stack((A, B))
initial = [0, 0]

#x = np.linspace(0, 90, 162000)

x = np.linspace(0, 10, 1000)

y1, y2, y5, y6, y7, y8, mp, mn = VP_TD_MOM(init = initial, contexts
    = contexts_stacked, alpha_pos = 0.05, alpha_neg = 0.2, gamma1 =
    0.9999,
                       gamma2 = 0.05, f = 0.1)
```

```



# VP Plot
fig, axes = plt.subplots(1, 1, figsize=(6, 3))

# Plot for the first graph
axes.plot(x, y1, label="A")
axes.plot(x, y2, label="B")
axes.plot(x, A, label="A input", alpha=0.20)
axes.plot(x, B, label="B input", alpha=0.20)
axes.set_title("Valence-Partitioned TD-Momentum Simulation",
    fontsize=15)
axes.set_xlabel("Time", fontsize=15)
axes.set_ylabel("Threat", fontsize=15)
axes.set_ylim(np.min(y1), np.max(y1))
axes.legend(loc="upper right")

plt.tight_layout()
plt.show()

# momentum plot
fig, axes = plt.subplots(1, 1, figsize=(6, 3))

# Plot for the first graph
axes.plot(x, mp, label="$m^P_t$")
axes.plot(x, mn, label="$m^N_t$")
axes.plot(x, A, label="A input", alpha=0.20)
axes.plot(x, B, label="B input", alpha=0.20)
axes.set_title("Valence-Partitioned Momentum Terms", fontsize=15)
axes.set_xlabel("Time", fontsize=15)
axes.set_ylabel("Momentum Value", fontsize=15)
axes.set_ylim(np.min(y1), np.max(y1))
axes.legend(loc="upper right")

plt.tight_layout()
plt.show()


# Positive and negative valence functions for each context
fig, axes = plt.subplots(1, 2, figsize=(12, 6))

```

```
# Plot for the first graph
axes[0].plot(x, y5, label="$V^P_A$")
axes[0].plot(x, y6, label="$V^N_A$")
axes[0].plot(x, A, label="A input", alpha=0.20)
axes[0].set_xlabel("Time", fontsize=15)
axes[0].set_ylabel("Threat", fontsize=15)
axes[0].set_title("Valence-Partitioned Value Functions - Context A",
    fontsize=15)
#axes[0].set_ylim(0, np.max(y3))
axes[0].legend(loc="upper right")

# Plot for the second graph
axes[1].plot(x, y7, label="$V^P_B$")
axes[1].plot(x, y8, label="$V^N_B$")
axes[1].plot(x, B, label="B input", alpha=0.20)
axes[1].set_xlabel("Time", fontsize=15)
axes[1].set_ylabel("Threat", fontsize=15)
axes[1].set_title("Valence-Partitioned Value Functions - Context B",
    fontsize=15)
#axes[1].set_ylim(np.min(y1), np.max(y1))
axes[1].legend(loc="upper right")

plt.tight_layout()
plt.show()
```