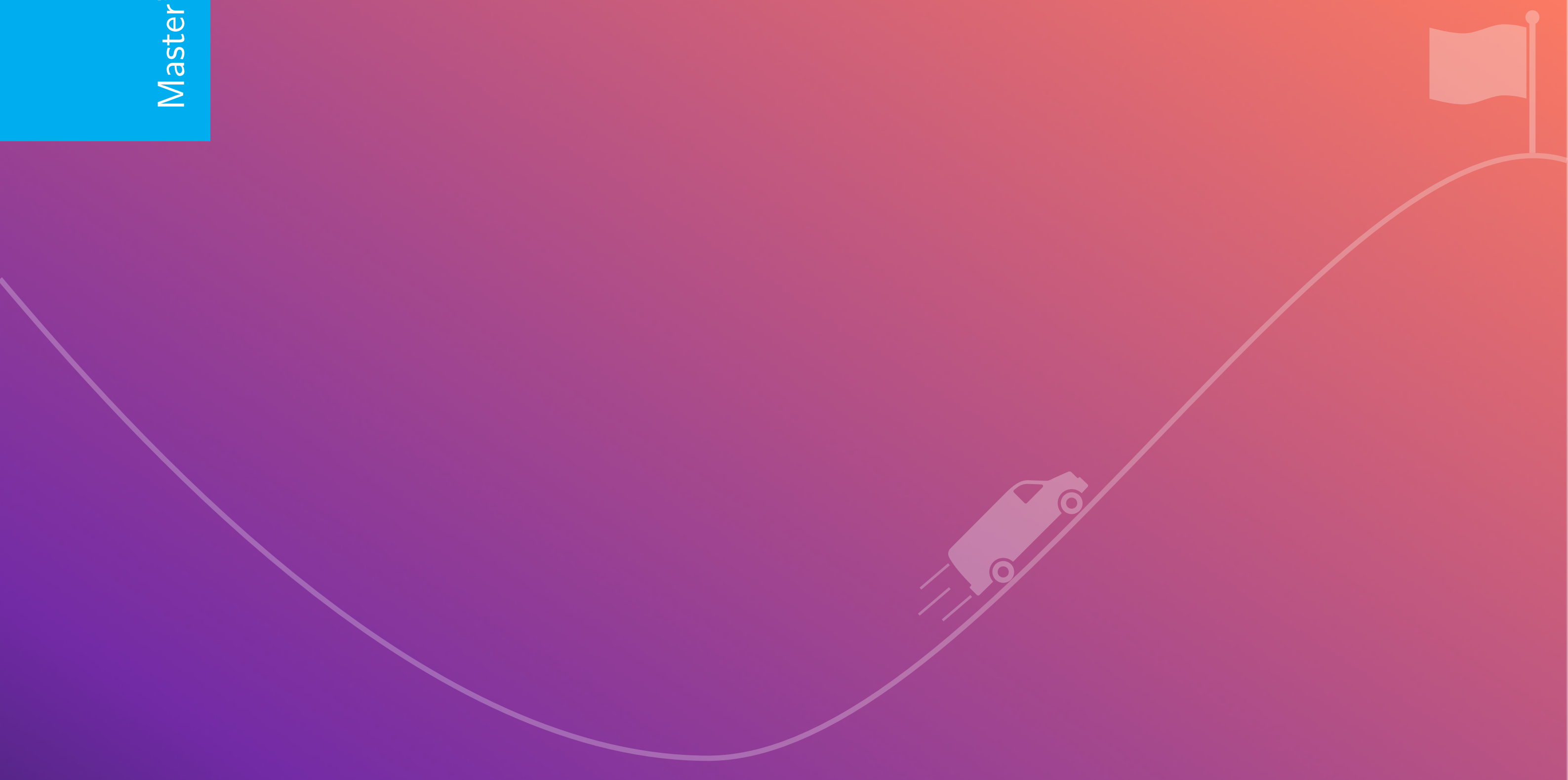
Online reinforcement learning with sparse rewards through an active inference capsule

Alejandro Daniel Noel

Master's thesis



Delft University of Technology

Department of Cognitive Robotics

Online reinforcement learning with sparse rewards through an active inference capsule

by

Alejandro Daniel Noel

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on Friday June 11th, 2021 at 2:00 PM

MSc. Mechanical Engineering
BMD - Biorobotics track

May 28, 2021

Student number:	4456971	
Thesis committee:	Charel van Hoof,	TU Delft, Main supervisor
	Martijn Wisse,	TU Delft, Chairman
	Riccardo Ferrari,	TU Delft, External examiner

Faculty of Mechanical, Maritime and Materials Engineering (3mE)
Delft University of Technology

Online reinforcement learning with sparse rewards through an active inference capsule

Alejandro Daniel Noel
Department of Cognitive Robotics
Delft University of Technology
adanielnoel@gmail.com

Charel van Hoof
Department of Cognitive Robotics
Delft University of Technology
charel.van.hoof@gmail.com

Beren Millidge
MRC Brain Network Dynamics Unit
University of Oxford
beren@millidge.name

Abstract

Intelligent agents must pursue their goals in complex environments with partial information and often limited computational capacity. Reinforcement learning methods have achieved great success by creating agents that optimize engineered reward functions, but which often struggle to learn in sparse-reward environments, generally require many environmental interactions to perform well, and are typically computationally very expensive. Active inference is a model-based approach that directs agents to explore uncertain states while adhering to a prior model of their goal behaviour. This paper introduces an active inference agent which minimizes the novel *free energy of the expected future*. Our model is capable of solving sparse-reward problems with a very high sample efficiency due to its objective function, which encourages directed exploration of uncertain states. Moreover, our model is computationally very light and can operate in a fully online manner while achieving comparable performance to offline RL methods. We showcase the capabilities of our model by solving the mountain car problem, where we demonstrate its superior exploration properties and its robustness to observation noise, which in fact improves performance. We also introduce a novel method for approximating the prior model from the reward function, which simplifies the expression of complex objectives and improves performance over previous active inference approaches.

1 Introduction

The field of Reinforcement Learning (RL) has achieved great success in designing artificial agents that can learn to navigate and solve unknown environments, and has had significant applications in robotics [Kober et al., 2013, Polydoros and Nalpantidis, 2017], game playing [Mnih et al., 2015, Silver et al., 2017, Shao et al., 2019], and many other dynamically varying environments with nontrivial solutions [Padakandla, 2020]. However, environments with sparse reward signals are still an open challenge in RL because optimizing policies over Heaviside or deceptive reward functions such as that in the mountain car problem requires substantial exploration to experience enough reward to learn.

Recently, Bayesian RL approaches [Ghavamzadeh et al., 2015] and the inclusion of novelty in objective functions [Stadie et al., 2015, Burda et al., 2018, Shyam et al., 2019] have begun to explicitly address the inherent exploration-exploitation trade-off in such sparse reward problems. In

Preprint. Under review.

parallel to these developments, active inference (AIF) has emerged from the cognitive sciences as a principled framework for intelligent and self-organising behaviour which naturally often converges with state of the art paradigms in RL (e.g., Friston et al. [2009, 2015], Kaplan and Friston [2018], Tschantz et al. [2020]). AIF agents minimize the divergence between an unbiased generative model of the world and a biased generative model of their preferences (shortly, the *prior*). This objective assigns an epistemic value to uncertain states, which enables directed exploration. Because of its principled foundations and because reward functions can be seen as an indirect way of defining prior models (cf. reward shaping [Ng et al., 1999]), active inference is often presented as a generalization of RL, with KL-control and control-as-inference as close ontological relatives [Millidge et al., 2020b].

1.1 Related work

Until recently, active inference implementations have been constrained to toy problems in theoretical expositions [Friston et al., 2015, 2017a,b]. Based on the work by Kingma and Welling [2014] on amortized variational inference, Ueltzhöffer [2018] proposed the first scalable implementation of AIF using deep neural networks to encode the unbiased generative model and evolution strategies to estimate policy gradients from multiple parallel simulations on a GPU. Later publications proposed more efficient policy optimization schemes, such as amortized policy inference [Millidge, 2019] and applying the cross-entropy method [Tschantz et al., 2019, 2020]. This latter work also uses an improved extension of the model free energy to future states, namely, the *free energy of the expected future* [Millidge et al., 2020a] (cf. divergence minimization [Hafner et al., 2020]). In these papers, active inference is shown to deliver better performance than current state of the art RL algorithms on sparse-reward environments, although they use the goal states as hard-coded priors. We improve upon the model of Tschantz et al. [2020] by demonstrating fully online learning on a single CPU core, by modeling the transition model with gated recurrent units that can capture environment dynamics over longer periods, and by learning the prior model from the (sparse) reward function through a novel reward shaping algorithm.

2 Active inference

The objective of an active inference (AIF[1]) agent is to minimize surprise, defined as the negative log-likelihood of an observation, $-\ln p(y)$. However, it is often intractable to compute this quantity directly. Instead, we apply variational inference and minimize a tractable upper bound for surprisal, namely, the variational free energy (VFE) of a latent model of the world. The agent possesses an approximate posterior distribution $q(x \mid y)$, where x is the latent state that is optimized to minimize the variational free energy. The parameters of this approximate posterior can be thought of as the agent's 'beliefs' about its environment. The variational free energy can be written as:

$$\mathrm{VFE} = \mathbb{E}_{q(x_t|y_t)}\left[-\ln p\left(y_t \mid x_t\right)\right] + \mathrm{D}_{\mathrm{KL}}\left[q(x_t \mid y_t) \| \, p(x_t)\right] \tag{1}$$

which is equivalent to the negative of the expectation lower bound (ELBO) used in variational inference (e.g., [Attias, 1999, Kingma and Welling, 2014]).

Active inference agents select actions that are expected to minimize the path integral of the VFE for future states [Friston, 2012]. There are two common extensions of the VFE to account for future states, the *expected free energy* [Friston et al., 2015] and the *free energy of the expected future* (FEEF) [Tschantz et al., 2020], which we use in this work. Millidge et al. [2020a] argues that the FEEF is the only one consistent with Equation 1 when evaluated at the current time and additionally considers the expected entropy in the likelihood $p\left(y_t \mid x_t\right)$ when selecting a policy.

2.1 Free energy of the expected future

The FEEF is a scalar quantity that measures the KL-divergence between unbiased beliefs about future states and observations and an agent's preferences over those states. The preferences are expressed as a biased generative model of the agent $\tilde{p}\left(y, x\right)$, also known as the *prior*. As in RL, the agent's world is modelled as a partially observed Markov decision process (POMDP) [Kaelbling et al., 1998, Sutton and Barto, 1998, Murphy, 2000], where t is the current time, τ is some timestep in the future and T is the prediction or planning horizon so that $t \leq \tau \leq T$. A policy is a sequence

[1] It is common to abbreviate *active inference* as AIF to avoid confusion with *artificial intelligence*.

of actions $[a_t, \ldots a_\tau, \ldots a_T]$ sampled from a Gaussian policy $\boldsymbol{\pi}$ that is conditionally independent across timesteps (i.e., has diagonal covariance matrix). We use the notation $\pi \sim \boldsymbol{\pi}$ for a random policy sample and its corresponding Gaussian policy, respectively. The FEEF can be separated into an extrinsic (objective-seeking) term and an intrinsic (information-seeking) term when assuming the factorization $\tilde{p}\left(y_\tau, x_\tau\right) \approx q\left(x_\tau \mid y_\tau\right) \tilde{p}\left(y_\tau\right)$:

$$\begin{aligned} \operatorname{FEEF}(\pi)_\tau &= \mathbb{E}_{q(y_\tau, x_\tau \mid \pi)} \mathrm{D}_{\mathrm{KL}}\left[q\left(y_\tau, x_\tau \mid \pi\right) \| \tilde{p}\left(y_\tau, x_\tau\right)\right] \\ &\approx \underbrace{\mathbb{E}_{q(x_\tau \mid \pi)} \mathrm{D}_{\mathrm{KL}}\left[q\left(y_\tau \mid x_\tau\right) \| \tilde{p}\left(y_\tau\right)\right]}_{\text{Extrinsic value}} - \underbrace{\mathbb{E}_{q(y_\tau \mid x_\tau, \pi)} \mathrm{D}_{\mathrm{KL}}\left[q\left(x_\tau \mid y_\tau\right) \| q\left(x_\tau \mid \pi\right)\right]}_{\text{Intrinsic value}} \end{aligned} \quad (2)$$

where the difference between the likelihoods $q\left(y_\tau \mid x_\tau\right)$ and $p\left(y_\tau \mid x_\tau\right)$ in Equation 1 is simply notational.

Minimizing the extrinsic term biases agents towards policies that yield predictions close to their prior (i.e. their desired future). Maximizing the intrinsic term gives agents a preference for states which will lead to a large information gain – i.e., the agent tries to visit the states where it will learn the most. The combination of extrinsic and intrinsic value together in a single objective leads to *goal-directed* exploration, where the agent is driven to explore, but only in regions which are fruitful in terms of achieving its goals. There is an additional exploratory factor implicit in the use of a KL-divergence in the extrinsic term, which pushes the agent towards observations where the generative model is not as certain about the likely outcome [Millidge et al., 2020a] due to the observation entropy term in the KL-divergence.

The optimal Gaussian policy $\boldsymbol{\pi}^*$ is found through the optimization

$$\boldsymbol{\pi}^* = \arg\min_{\boldsymbol{\pi}} \sum_{\tau=t}^{T} \operatorname{FEEF}(\pi)_\tau \quad (3)$$

by means of a maximum likelihood approach. Although the FEEF is not a likelihood, Whittle [1991] shows that treating a path integral of a cost as a negative log-likelihood to minimize is formally equivalent to least squares optimization methods, but more direct to compute.

3 The Active inference capsule

The active inference capsule consists of a variational autoencoder (VAE) which maps the agent's noisy observations to a latent representation, a gated recurrent unit (GRU) [Cho et al., 2014] which predicts future latent states from the current latent state, and a policy optimization scheme that minimizes the FEEF over a trajectory. The VAE and GRU learn an unbiased latent dynamical model in a similar fashion as *world models* by Ha and Schmidhuber [2018]. Additionally, we propose an extension where the prior model is also learned by the agent from the reward signal. A block-diagram of the capsule is shown in Figure 1.

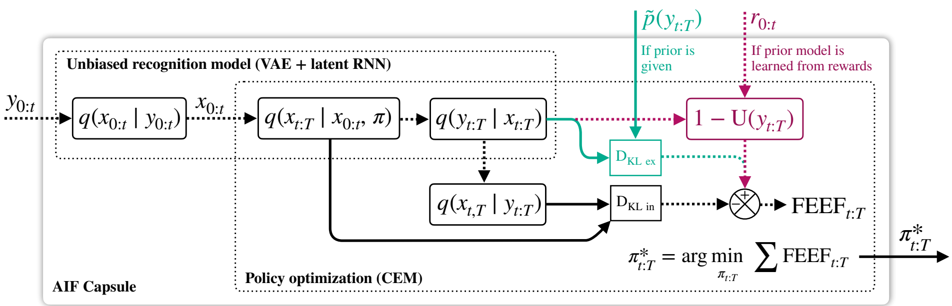


Figure 1: Block-diagram of the active inference capsule. The inputs are time-series of the observations plus the biased priors on future states if given or else the rewards to learn from. The differences between either source of priors are indicated with distinctive colors. The output is a time-series of optimized beliefs over actions (i.e., a Gaussian policy) up to the planning horizon. Continuous lines carry probability distributions whereas discontinuous lines carry real numbers (random samples).

Perception and planning Both perception and planning are treated as inference processes (see Figure 2). During perception, the capsule performs inference on the observations y through the unbiased variational posterior and transition models. This updates the belief on the latent states x, the recurrent states h of the GRU, which integrate temporal relations between latent states, and the parameters of these models through a learning step. During planning, on the other hand, the current recurrent states are used as initial conditions for the transition model to project future trajectories and evaluate the FEEF for policy optimization.

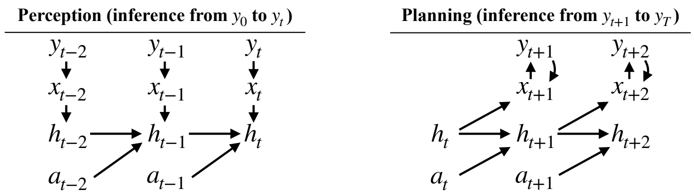


Figure 2: Graphical models of the inference during perception and planning. The future latents are inferred again from the predicted observations, which captures the uncertainty from the variational posterior into the FEEF.

Gaussian variational autoencoder The variational autoencoder (VAE) consists of the variational posterior $q(x_t \mid y_t)$ (encoder) and the likelihood $q(y_t \mid x_t)$ (decoder), both Gaussian and approximated through amortized inference with neural networks [Kingma and Welling, 2014]. As pointed out by Mattei and Frellsen [2018], the optimization of VAEs for continuous outputs is ill-posed. To circumvent the issue, we fix the variance of the decoder to a default value or to the noise level of the input if given (see the hyperparameters in Appendix A). The results are only mildly sensitive to this parameter because the minima of the extrinsic term in Equation 2 with respect to the policy only depends on the mode of the likelihood, which is ultimately unaffected by the noise. It does, nonetheless, affect the spread of the likelihood and therefore the gradient of the optimization landscape.

Recurrent transition model The transition model is factorized into two terms:

$$q\left(x_{t+1} \mid x_t, \pi\right) \equiv q\left(x_{t+1} \mid h_t\right) q\left(h_t \mid h_{t-1}, x_t, \pi\right) \tag{4}$$

The right term is implemented by a GRU which processes the temporal information of the input through a recurrent hidden state h. The left term is implemented by a fully-connected (FC) network which predicts both the update on the latent state and the expected variance. Because the recurrent states are deterministic, the variance of the predicted latent distributions does not include prediction errors, which could be an improvement for future work. Our diagram in Figure 3 differs from Ha and Schmidhuber [2018] in that it predicts an update on the latent state rather than the latent state itself, as done by Tschantz et al. [2020]. Learning is achieved via stochastic gradient descent where the loss is the KL-divergence between the predicted latent distribution and the variational posterior after the observation, so that $q\left(x_{t+1} \mid \pi\right) \approx q\left(x_{t+1} \mid y_{t+1}\right)$.

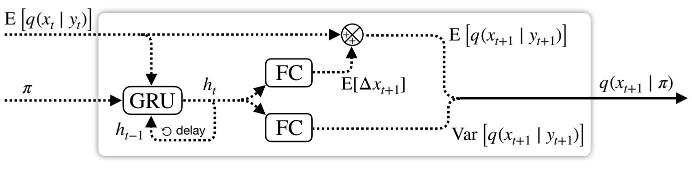


Figure 3: Block-diagram of the transition model.

Model free energy In contrast to variational autoencoders, which only consider present observations, in an AIF capsule the latent states are also inferred from past sensory data. We propose the following modification of Equation 1 that accounts for the transition model by using it in place of the variational posterior and instead using the variational posterior of the current observation as prior for the predicted latent states:

$$\mathrm{VFE}_{\text{capsule}} = \mathbb{E}_{q(x_t \mid x_{t-1}, \pi)}\left[-\ln p\left(y_t \mid x_t\right)\right] + \mathrm{D}_{\mathrm{KL}}\left[q(x_t \mid x_{t-1}, \pi) \| q(x_t \mid y_t)\right] \tag{5}$$

This expression is used for evaluating the learning progress of the true generative model.

3.1 Defining the prior

The AIF capsule supports using both a pre-set prior or else learning one from the reward function. A prior in this context is simply a distribution over goal states. Under active inference, the agent uses its own unbiased world model to generate trajectories that maximize the likelihood of the prior (i.e., reaching the goal). Importantly, when learning the prior the agent can also store information about the optimal solution in a trajectory-independent way by modelling intermediate goal states. Moreover, learning the prior enables the use of active inference agents in situations where manually defining a prior is unfeasible. We propose a novel approach for learning the prior directly based on rewards using Bellman's optimality principle, therefore making a link with model-based reinforcement learning.

3.1.1 Method for learning the prior model from rewards

This section presents a novel method for optimal reward shaping [Ng et al., 1999] through a continuous potential function that preserves the optimal policy and is compatible with the extrinsic value in Equation 2. Let $\mathbf{U}(y)$ be a model of the utility of a state y to a trajectory passing through it. The utility is a scaled sum of potential future rewards. We define a reward r_t as a real number in the range $[-1, 1]$ associated with an observation y_t, where -1 is maximally undesirable and 1 maximally desirable. We use the same range and definitions for the utility. We suggest a similarity between the utility model and the extrinsic value of the FEEF, which instead is a real-valued in the range $[0, \infty)$ where 0 corresponds to preferences being perfectly fulfilled and ∞ to absolutely unpreferred states. Assuming the agent is not too far from its generative model we approximate the relation as

$$1 - \mathbf{U}(y_t) \underset{\sim}{\propto} \mathbb{E}_{q(x_t|\pi)} \mathrm{D}_{\mathrm{KL}} \left[q\left(y_t \mid x_t\right) \| \tilde{p}\left(y_t\right) \right] \tag{6}$$

In other words, both terms have approximately the same landscape. Because the optimal policy does not depend on the absolute values of the FEEF but only on its minima, we can use $1 - \mathbf{U}(y_t)$ during policy optimization as a surrogate for the extrinsic term which implicitly contains the prior.

We model $\mathbf{U}$ as a multi-layered neural network with $\tanh$ activation on the outputs. At every observation, information from the reward signal is infused into the utility model through a stochastic gradient descend (SGD) step. The loss is the mean squared error between the predicted utility and the utility by applying Bellman's equation over a trajectory. In the latter, the discount factor is exponentially decreased for past observations, which pushes the agent to more quickly reach the rewarding states. Moreover, the learning rate in the SGD step is scaled with the absolute value of the reward, which regularizes the magnitude of the model update with the intensity of the stimulus. We also iterate multiple times the process to allow information to propagate back in time more effectively. See algorithm 1 for pseudo-code of our dynamic programming approach of learning the utility model.

Algorithm 1: Learning the utility model from rewards

Input: Observations $y_{t_0:t}$ — rewards $r_{t_0:t}$ — utility model $\mathbf{U}$ — discount factor β — learning rate α — iterations L

for *Iteration* $i = 1 \ldots L$ **do**
    Initialize empty list of expected utilities $\hat{u}$
    for $\tau = t_0 \ldots t$ **do**
        if τ = t **then**
            $\hat{u}_t \leftarrow r_\tau$ (in an online setting, future observations are unavailable)
        else
            $\hat{u}_\tau \leftarrow r_\tau + \beta^{t-\tau}\mathbf{U}(y_{\tau+1})$ (Bellman's equation)
        end
    end
    $\mathcal{L} \leftarrow \mathrm{MSE}(\mathbf{U}(y_{t_0:t}), \hat{u}_{t_0:t})$ (Compute loss)
    $\frac{\partial W_\mathbf{U}}{\partial \mathcal{L}} \leftarrow \mathrm{Backpropagate}(\mathbf{U}, \mathcal{L})$ (Compute weight gradients)
    $W_\mathbf{U} \leftarrow W_\mathbf{U} - \alpha \left| r_t \right| \frac{\partial W_\mathbf{U}}{\partial \mathcal{L}}$ (Update weights)
end

3.1.2 Policy optimization

Policies are optimized using the cross-entropy method (CEM) [Rubinstein, 1997]. Since the algorithm is constrained to output Gaussian policies, the exact shape of the FEEF is not captured but the resulting policies do track its minima [Tschantz et al., 2020]. The pseudocode for the optimization is provided in algorithm 2.

Algorithm 2: Cross-entropy method for policy optimization

Input: Planning horizon T — Optimization iterations I — # policy samples N — # candidate policies K — Transition model $q(x_{t+1} \mid h_t), q(h_t \mid h_{t-1}, x_t, \pi)$ — encoder $p(x_t \mid y_t)$ — decoder $p(y_t \mid x_t)$ — current states $\{x_t, h_{t-1}\}$ — prior $\tilde{p}(y_t)$

Initialize a Gaussian policy $\boldsymbol{\pi} \leftarrow \mathcal{N}(\mathbf{0}, \mathbb{I}_{H\times H})$

for *iteration* $i = 1 \ldots I$ **do**

  for *sample policy* $j = 1 \ldots N$ **do**

    $\pi^{(j)} \sim \boldsymbol{\pi}$

    $\text{FEEF}^{(j)} = 0$

    for $\tau = t \ldots T-1$ **do**

$$\begin{aligned}
h_\tau &\leftarrow \mathbb{E}[q(h_\tau \mid h_{\tau-1}, \pi_\tau^{(j)}, x_\tau)] \\
q(x_{\tau+1} \mid \pi^{(j)}) &\leftarrow q(x_{\tau+1} \mid h_\tau) \\
q(y_{\tau+1} \mid x_{\tau+1}) &\leftarrow \mathbb{E}_{q(x_{\tau+1}|\pi^{(j)})}[q(y_{\tau+1} \mid x_{\tau+1})] \\
q(x_{\tau+1} \mid y_{\tau+1}) &\leftarrow \mathbb{E}_{q(y_{\tau+1}|\pi^{(j)})}[q(x_{\tau+1} \mid y_{\tau+1})] \\
\text{FEEF}_{\tau+1}^{(j)} &\leftarrow \quad \mathbb{E}_{q(x_{\tau+1}|\pi^{(j)})}\mathrm{D}_{\mathrm{KL}}[q(y_{\tau+1} \mid x_{\tau+1}) \| \, \tilde{p}(y_{\tau+1})] \\
&\qquad - \mathbb{E}_{q(y_{\tau+1}|\pi^{(j)})}\mathrm{D}_{\mathrm{KL}}[q(x_{\tau+1} \mid y_{\tau+1}) \| \, q(x_{\tau+1} \mid \pi^{(j)})] \\
\text{FEEF}^{(j)} &\leftarrow \text{FEEF}^{(j)} + \text{FEEF}_{\tau+1}^{(j)} \\
x_{\tau+1} &\leftarrow \mathbb{E}[q(x_{\tau+1} \mid \pi^{(j)})]
\end{aligned}$$

    end

  end

  Select best K policies Refit Gaussian policy $\boldsymbol{\pi} \leftarrow \text{refit}(\hat{\pi})$

end

return $\boldsymbol{\pi}$

---

4 Experiments on the mountain car problem

In this section, we study the performance of the active inference capsule using the continuous mountain car problem from the open-source code library OpenAI Gym [Brockman et al., 2016]. This is a challenging problem for reinforcement learning algorithms because it requires a substantial amount of exploration to overcome the sparse reward function (negative for every additional action, positive only at the goal). Moreover, the task requires the agent to move away from the goal at first in order to succeed. The objective is to reach the goal in less than 200 simulation steps. In our experiments, the agent time-step size is 6 simulation steps and the planning window $H = T - t$ is defined in the agent's time-scale (see subsection 4.1 for details).

Online learning For all tasks, we initialize all the agents with random weights and learn online only. Training an agent for 150 epochs takes about 3 minutes on a single CPU core (Intel I7-4870HQ). In contrast, previous approaches using active inference [Ueltzhöffer, 2018, Tschantz et al., 2019, 2020] and policy gradient methods (e.g., [Liu et al., 2017]) use (offline) policy replay and typically need hours of GPU-accelerated compute while achieving similar convergence. To our knowledge, this is the first model-based RL method to learn online using neural network representations. This is afforded by the high sample efficiency of the FEEF, which directs exploration towards states that are uncertain for both the encoder and transition models.

Given priors versus learned priors Figure 4 shows that agents with a given prior (a Gaussian distribution around the goal state) depend on their planning window to find more optimal policies, whereas agents that learn the prior converge to optimal policies with much shorter planning windows and without such dependency. Figure 5 shows that the given prior misleads agents with short foresight to swing forward first, whereas the learned prior integrates information about the better strategy and can be followed without a full preview of the trajectory to the goal. These results highlight the importance of the prior for model exploitation. The unbiased predictor is an egocentric model of the world, whereas the prior model is an allocentric representation of the agent's intended behaviour. The active inference capsule effectively combines both during policy optimization, therefore defining a prior based only on the final goal blurs the objective for shorter planning windows. This experiment shows that the reward function is a simple means of indirectly modelling a complex prior.

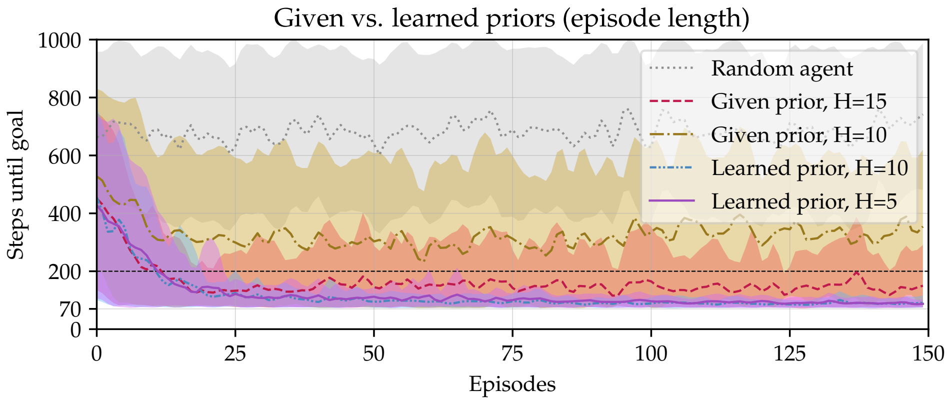


Figure 4: Training curves for different types of agents. When given the prior, agents with a planning window of 90 simulation steps (H=15) can reach the goal within the 200-step limit, whereas agents with only a 60-step (H=10) foresight fail. The shortest possible time to the goal is about 70 simulation steps. Agents that learn the prior converge to the optimal solution even if the planning horizon is significantly earlier than 70 steps ahead, showing that the learned prior also captures information about the optimal trajectories and not just the goal. Despite starting from a randomly initialized model, AIF agents can direct exploration already from the first episode, evidenced by the better initial performance compared to agents with purely random actions.

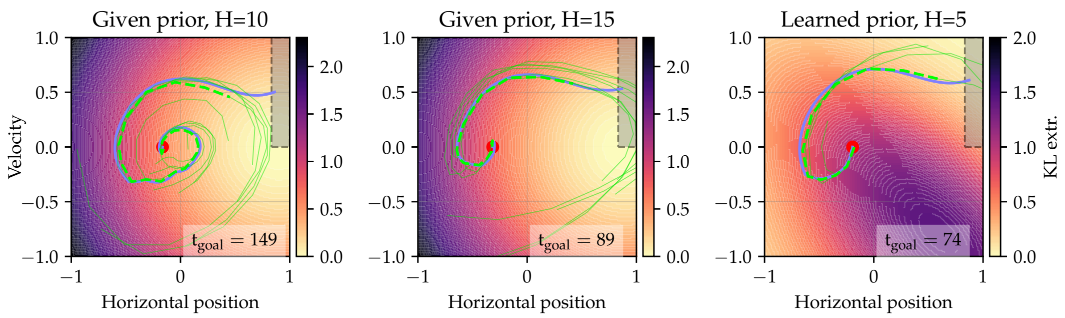


Figure 5: Phase portraits of a trained agent of each type. The contour maps are the extrinsic values of the FEEF, revealing the Gaussian priors and the learned prior, which also captures information about the optimal trajectory (higher cost in being closer to the goal but having to swing back). The red dots are the initial positions (randomized across trials), the thick continuous lines are the true observations, the thick discontinuous lines are reconstructions by the VAE, and the thin lines are the predictions projected onto the observation space through the decoder model.

Exploration properties Figure 4 also reveals that, despite starting without an objective, agents that learn the prior on average find the solution in the first episode faster than agents that take random actions and even agents with a given prior. This is an example of the information-seeking objective of the FEEF. It results in a rapid and directed exploration of the state-space, which accelerates the solution to this sparse reward RL problem.

Effect of observation noise We explore the effect of adding Gaussian noise to the observations. Figure 6 shows that, despite a brief initial disadvantage, the agents with noisy observations match the performance of those with clean sensory data and even converge towards the optimal solution. We think that this robustness to observation noise is supported by the KL-divergence in the extrinsic term, as pointed out by Hafner et al. [2020] in the divergence minimization framework. In fact, rather than impairing the capsule, observation noise actually improves learning of the unbiased model, evidenced by the much faster convergence of the model free energy ($\mathrm{VFE}_{\mathrm{capsule}}$). [An, 1996] showed that additional input noise induces a regularizing effect on the backpropagated errors that can improve parameter exploration and prevent overfitting.

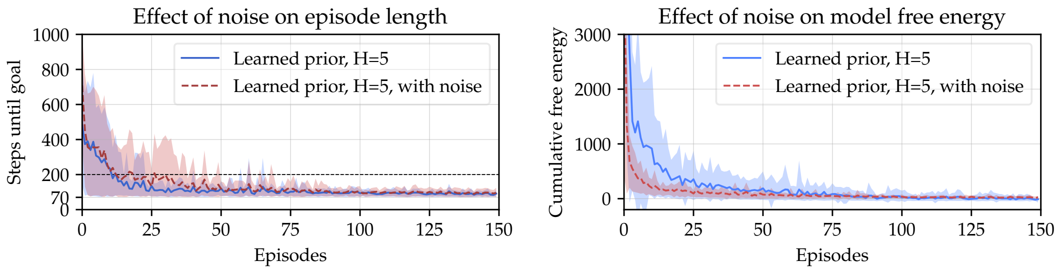


Figure 6: **(Left)** Training curves for agents that learn their own prior, with and without observation noise. Both have very similar convergence, showing that the model is robust to noise. **(Right)** Cumulative free energy of each episode. The model free energy for agents with observation noise converges much faster, possibly due to the additional regularizing effect against local optima.

Ablation study We explore the contributions of the intrinsic and the extrinsic terms in the behaviour of the agents. Figure 7 shows that the intrinsic term alone drives convergent behaviour in the mountain car problem. This is because the goal states are also the rarest (at most once per trial) and therefore directed exploration is both necessary and sufficient to solve the task. Instead, the extrinsic term alone almost never finds the goal state. The extrinsic term promotes exploration of the observation space but not of the latent space (see subsection 2.1), which results in a lower sample efficiency for model exploration. However, if we hot-start the agent for a few steps before disabling the intrinsic term the behaviour becomes bimodal: the prior model can sometimes gather enough experience for the extrinsic term to maintain a convergent behaviour. These results show that, while the extrinsic term is responsible for the convergence to optimal solutions, the intrinsic term is key for making this behaviour robust because it promotes policies with a high entropy, which prevents convergence to local minima, as well as generates sufficient exploration of the state-space to obtain the sparse reward necessary for learning.

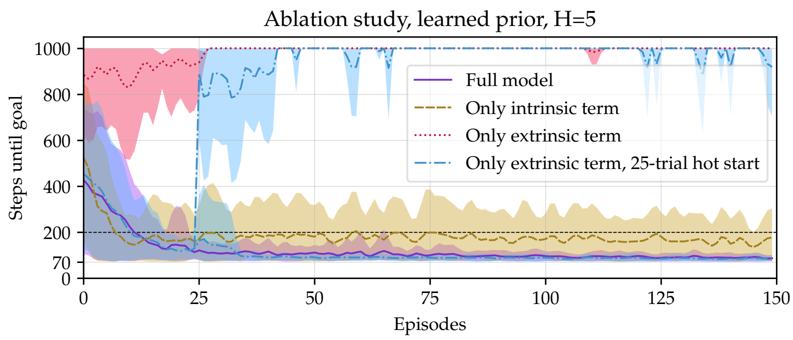


Figure 7: Training curves by selectively disabling the terms of the FEEF. The intrinsic term alone (directed exploration) is enough to solve the mountain car problem. The extrinsic term alone is not enough unless hot-started with some initial episodes with the full model. Then it either overcomes the sparse rewards and converges or it fails to learn a good prior model due to insufficient exploration (the plot is bimodal). The full model combines the convergent but unstable behaviour of the extrinsic term with the robustness furnished by the high sample efficiency of the intrinsic term.

4.1 Implementation details

We update the agent every 6 simulation steps and apply the same action during that period. This simple action-repeat method reduces computation cost, can increase the prediction performance due to higher feature gradients and lower variance of future choices, and is observed in human subjects too [Mnih et al., 2015, Sharma et al., 2017]. Following the same idea, the agent also commits to executing the first two actions of the chosen policy. Effectively, the agent revises its policy every 12 simulation steps. For each experiment, we train 30 agents of each type and plot their mean and the region of one standard deviation, clipped to the minimum or maximum per episode if exceeded. See Appendix A for specific details of the neural networks and hyperparameters.

5 Discussion

We introduced an active inference capsule that can solve RL problems with sparse rewards online and with a much smaller computational footprint than other approaches. This is achieved through minimizing the novel *free energy of the expected future* objective with neural networks, which enables a natural exploration-exploitation balance, a very high sample efficiency and robustness to input noise.

Moreover, the capsule can either directly follow a given prior (i.e., the goal states) or learn one from the reward signal using a novel algorithm based on Bellman's equation. We compare both approaches and show that agents with learned priors converge to optimal trajectories with much shorter planning horizons. This is because learned priors approximate a density map of the rewarding trajectories, whereas given priors typically only provide information of the final goals.

Our results show that the FEEF induces entropy regularization on the policies through uncertainty sampling, which prevents local convergence and accelerates learning. Moreover, our algorithm learns priors that push the agent to achieve its goals as early as possible within the constraints of the problem. The combination of these two characteristics results in a fast and consistent convergence towards the optimal solution in the mountain-car problem, which is a challenge for state of the art RL methods.

Despite the success of the method in the mountain car problem, it remains unclear if the goal-directed exploration properties will scale to high-dimensional inputs or much more complex dynamics. It is also unclear whereas the method we introduced for learning the prior model from the reward signal is generally applicable to any problem. When working with images as inputs, it may be necessary to pre-train the VAE offline on a large dataset and to use a GPU for accelerating computations.

Finally, we believe that the AIF capsule could become a building block for hierarchical RL, where lower layers abstract action and perception into increasingly expressive spatiotemporal commands and higher layers output priors for the lower layers. In this set up, scalability and generality would be achieved by designing wider and deeper networks of AIF capsules, rather than using a large single capsule.

Broader Impact

Active inference and the underlying free energy principle describe the self-organising behaviour of biological systems at different spatiotemporal scales, ranging from microscales (e.g., cells), to intermediate scales (e.g., learning processes), to macroscales (e.g., societal organization and the emergence of new species) [Hesp et al., 2019]. It is a relatively new science with broad-ranging applications in any technology that has to interact with the real world. But because of its complexity and lack of efficient implementations, active inference has mostly remained an explanatory device with limited applicability outside of the scientific scope. Developments like the active inference capsule presented here may soon unlock the benefits of this new technology for nanobiology, robotics, artificial intelligence, financial technologies, and other high-tech markets.

We acknowledge that the development of this technology may raise safety and ethical concerns in the future, although the scope of the present work is still only methodological. Nonetheless, the model-based nature of active inference renders its decisions partially explainable inasmuch as we understand its priors, which are typically easy to interpret since they are expressed in the observation space (e.g., Figure 5). This can be a great advantage over model-free RL methods, which instead are very hard to interpret and to validate for safety-critical applications.

References

Guozhong An. The Effects of Adding Noise during Backpropagation Training on a Generalization Performance. *Neural Computation*, 8(3):643–674, 1996. ISSN 08997667. doi: 10.1162/neco.1996.8.3.643.

Hagai Attias. Inferring Parameters and Structure of Latent Variable Models by Variational Bayes. In *Fifteenth conference on Uncertainty in artificial intelligence*, pages 21–30, jan 1999. URL `http://arxiv.org/abs/1301.6676`.

Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym, 2016. URL `http://arxiv.org/abs/1606.01540`.

Yuri Burda, Harrison Edwards, Amos Storkey, and Oleg Klimov. Exploration by random network distillation. *arXiv preprint*, pages 1–17, 2018. ISSN 23318422.

Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1724–1734, Stroudsburg, PA, USA, jun 2014. Association for Computational Linguistics. doi: 10.3115/v1/D14-1179. URL `http://arxiv.org/abs/1406.1078http://aclweb.org/anthology/D14-1179`.

Stefan Elfwing, Eiji Uchibe, and Kenji Doya. Sigmoid-weighted linear units for neural network function approximation in reinforcement learning. *Neural Networks*, 107(2015):3–11, 2018. ISSN 18792782. doi: 10.1016/j.neunet.2017.12.012.

Karl Friston, Francesco Rigoli, Dimitri Ognibene, Christoph Mathys, Thomas Fitzgerald, and Giovanni Pezzulo. Active inference and epistemic value. *Cognitive Neuroscience*, 6(4):187–214, oct 2015. ISSN 1758-8928. doi: 10.1080/17588928.2015.1020053.

Karl Friston, Thomas FitzGerald, Francesco Rigoli, Philipp Schwartenbeck, and Giovanni Pezzulo. Active Inference: A Process Theory. *Neural Computation*, 29(1):1–49, jan 2017a. ISSN 0899-7667. doi: 10.1162/NECO_a_00912. URL `http://arxiv.org/abs/1803.01446https://www.mitpressjournals.org/doi/abs/10.1162/NECO_a_00912`.

Karl J. Friston. A free energy principle for biological systems. *Entropy*, 14(11):2100–2121, 2012. ISSN 10994300. doi: 10.3390/e14112100.

Karl J. Friston, Jean Daunizeau, and Stefan J. Kiebel. Reinforcement Learning or Active Inference? *PLoS ONE*, 4(7):e6421, jul 2009. ISSN 1932-6203. doi: 10.1371/journal.pone.0006421.

Karl J. Friston, Marco Lin, Christopher D. Frith, Giovanni Pezzulo, J. Allan Hobson, and Sasha Ondobaka. Active Inference, Curiosity and Insight. *Neural Computation*, 29(10):2633–2683, oct 2017b. ISSN 0899-7667. doi: 10.1162/neco_a_00999. URL `http://arxiv.org/abs/1803.01446https://direct.mit.edu/neco/article/29/10/2633-2683/8300`.

Mohammed Ghavamzadeh, Shie Mannor, Joelle Pineau, and Aviv Tamar. Bayesian Reinforcement Learning: A Survey. *Foundations and Trends® in Machine Learning*, 8(5-6):359–483, 2015. ISSN 1935-8237. doi: 10.1561/2200000049.

David Ha and Jurgen Schmidhuber. World models. *arXiv preprint*, 2018. ISSN 23318422. doi: 10.1016/b978-0-12-295180-0.50030-6.

Danijar Hafner, Pedro A. Ortega, Jimmy Ba, Thomas Parr, Karl J. Friston, and Nicolas Heess. Action and Perception as Divergence Minimization, sep 2020. URL `http://arxiv.org/abs/2009.01791`.

Casper Hesp, Maxwell J.D. Ramstead, Axel Constant, Paul Badcock, Michael Kirchhoff, and Karl J. Friston. A Multi-scale View of the Emergent Complexity of Life: A Free-Energy Proposal. In *Evolution, Develompent and Complexity*, pages 195–227. Springer International Publishing, 2019. doi: 10.1007/978-3-030-00075-2_7.

Leslie Pack Kaelbling, Michael L. Littman, and Anthony R. Cassandra. Planning and acting in partially observable stochastic domains. *Artificial Intelligence*, 101(1-2):99–134, may 1998. doi: 10.1016/S0004-3702(98)00023-X. URL `https://linkinghub.elsevier.com/retrieve/pii/S000437029800023X`.

Raphael Kaplan and Karl J. Friston. Planning and navigation as active inference. *Biological Cybernetics*, 112(4):323–343, 2018. ISSN 14320770. doi: 10.1007/s00422-018-0753-2. URL `https://doi.org/10.1007/s00422-018-0753-2`.

Diederik P. Kingma and Max Welling. Auto-Encoding Variational Bayes. In *International Conference on Learning Representations (ICLR)*, 2014. URL `http://arxiv.org/abs/1312.6114`.

Jens Kober, J. Andrew Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *International Journal of Robotics Research*, 32(11):1238–1274, 2013. ISSN 02783649. doi: 10.1177/0278364913495721.

Yang Liu, Prajit Ramachandran, and Jian Peng. Stein variational policy gradient. *arXiv preprint*, 2017. ISSN 23318422.

Pierre Alexandre Mattei and Jes Frellsen. Leveraging the exact likelihood of deep latent variable models. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pages 3859–3870, 2018.

Beren Millidge. Deep active inference as variational policy gradients. *Journal of Mathematical Psychology*, 96:102348, 2019. ISSN 10960880. doi: 10.1016/j.jmp.2020.102348.

Beren Millidge, Alexander Tschantz, and Christopher L. Buckley. Whence the Expected Free Energy? *arXiv preprint*, 2020a. ISSN 23318422. doi: 10.1162/neco_a_01354.

Beren Millidge, Alexander Tschantz, Anil K Seth, and Christopher L Buckley. On the Relationship Between Active Inference and Control as Inference. In *International Workshop on Active Inference*, pages 3–11, 2020b. doi: 10.1007/978-3-030-64919-7_1. URL `http://link.springer.com/10.1007/978-3-030-64919-7_1`.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015. ISSN 14764687. doi: 10.1038/nature14236. URL `http://dx.doi.org/10.1038/nature14236`.

K P Murphy. A survey of POMDP solution techniques. Technical report, 2000.

Andrew Y. Ng, Daishi Harada, and Stuart Russell. Policy invariance under reward transformations : Theory and application to reward shaping. *Sixteenth International Conference on Machine Learning*, 3:278–287, 1999. ISSN 1098-6596.

Sindhu Padakandla. A Survey of Reinforcement Learning Algorithms for Dynamically Varying Environments. *arXiv preprint*, pages 1–15, 2020. ISSN 23318422.

Athanasios S. Polydoros and Lazaros Nalpantidis. Survey of Model-Based Reinforcement Learning: Applications on Robotics. *Journal of Intelligent and Robotic Systems: Theory and Applications*, 86(2):153–173, 2017. ISSN 15730409. doi: 10.1007/s10846-017-0468-y.

Reuven Y. Rubinstein. Optimization of computer simulation models with rare events. *European Journal of Operational Research*, 99(1):89–112, 1997. ISSN 03772217. doi: 10.1016/S0377-2217(96)00385-2.

Kun Shao, Zhentao Tang, Yuanheng Zhu, Nannan Li, and Dongbin Zhao. A Survey of Deep Reinforcement Learning in Video Games. *arXiv preprint*, 2019.

Sahil Sharma, Aravind Srinivas, and Balaraman Ravindran. Learning to Repeat: Fine Grained Action Repetition for Deep Reinforcement Learning. *Iclr*, pages 34–47, feb 2017. URL `http://arxiv.org/abs/1702.06054`.

Pranav Shyam, Wojciech Jaskowski, and Faustino Gomez. Model-based active exploration. *36th International Conference on Machine Learning, ICML 2019*, 2019-June:10136–10152, 2019.

David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George Van Den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, 2017. ISSN 14764687. doi: 10.1038/nature24270. URL `http://dx.doi.org/10.1038/nature24270`.

Bradly C. Stadie, Sergey Levine, and Pieter Abbeel. Incentivizing Exploration In Reinforcement Learning With Deep Predictive Models. *ArXiv preprint*, pages 1–11, 2015. URL `http://arxiv.org/abs/1507.00814`.

Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: an Introduction*. MIT Press, Cambridge, MA, 1 edition, 1998.

Alexander Tschantz, Manuel Baltieri, Anil K. Seth, and Christopher L. Buckley. Scaling active inference. *arXiv preprint*, pages 1–13, nov 2019.

Alexander Tschantz, Beren Millidge, Anil K. Seth, and Christopher L. Buckley. Reinforcement Learning through Active Inference. *arXiv preprint*, feb 2020.

Kai Ueltzhöffer. Deep active inference. *Biological Cybernetics*, 112(6):547–573, 2018. ISSN 14320770. doi: 10.1007/s00422-018-0785-7.

P. Whittle. Likelihood and Cost as Path Integrals. *Journal of the Royal Statistical Society: Series B (Methodological)*, 53(3):505–529, jul 1991. ISSN 00359246. doi: 10.1111/j.2517-6161.1991.tb01842.x. URL `http://doi.wiley.com/10.1111/j.2517-6161.1991.tb01842.x`.

A Implementation details and hyperparameters

The variational posterior model has a hidden layer with SiLU activations, which are typically better than ReLU activations in RL settings [Elfwing et al., 2018], and two output layers for mean and standard deviation. The likelihood model has the same structure but outputs a fixed standard deviation (0.05 by default, 0.1 in the case of noisy inputs). The GRU of the transition model has input size $\dim(x) + \dim(a)$ and hidden size $\dim(z)$ parametrized by $2H \cdot \dim(x)$, where H is the planning window in the agent's time-scale. The FC layers map from $\dim(z)$ to $\dim(x)$. The observations are the position and velocity ($\dim(y) = 2$) and the actions are the horizontal force ($\dim(a) = 1$). The learned prior model consists of a single hidden layer with SiLU activations and Tanh activation on the outputs. The full list of hyperparameters is shown in Table 1.

Table 1: Agent hyperparameters

General hyperparameters	
Latent dimensions $\dim(x)$	2
VAE hidden layer size	20
Observation noise std.	0 or 0.1
Time ratio simulation / agent	6
VAE learning rate (ADAM)	0.001
Transition model learning rate (ADAM)	0.001
Policy hyperparameters	
Planning window H	6, 10 or 15
Actions before replanning	2
Policy samples N (CEM)	700 for $H \in \{6, 10\}$ 1500 for $H = 15$
Candidate policies K (CEM)	70
Optimization iterations I (CEM)	2
Hyperparameters for learned priors	
Hidden layer size (learned priors)	40
Learning rate (SGD)	0.1
SGD steps per reward	15
Discount factor β	0.995

B Code

The repository of the code has the following structure:

```
aif_capsule/
|-- README.md                     # not in appendix
|-- __init__.py                   # not in appendix
|-- run_capsule_mountain_car.py
|-- models/
|   |-- __init__.py               # not in appendix
|   |-- active_inference_capsule.py
|   |-- prior_model.py
|   |-- transition_model.py
|   |-- vae.py
|   `-- vae_dense_obs.py
|-- mountain_car/
|   |-- __init__.py               # not in appendix
|   |-- plot_multiple_trainings.py  # not in appendix (used during testing)
|   |-- plotting.py
|   `-- training.py
|-- utils/
|   |-- __init__.py               # not in appendix
|   |-- args_class.py             # not in appendix (used internally to mimmic comand line inputs)
|   |-- model_saving.py
|   |-- signal_smoothing.py       # not in appendix (used for smoothing plot lines)
|   |-- silu.py
|   |-- timeline.py
|   `-- value_map.py
`-- paper_results/                # not in appendix (contains trained models and plots)
```

The following subsections contain the code of the .py files relevant to training and evaluating AIF agents. The script run_capsule_mountain_car.py facilitates operating the software from the command line. Its capabilities include:

Training a single agent:

```
> python run_capsule_mountain_car.py --settings='./paper_results/settings_learned_prior_H5.json' --save_dirpath='./paper_results/simulation_results/'
```

Training a batch of independent agents to gather training statistics

```
> python run_capsule_mountain_car.py --settings='./paper_results/settings_learned_prior_H5.json' --save_dirpath='./paper_results/simulation_results/' --batch_agents=30
```

Generating a video from a trained model (see an example here)

```
> python run_capsule_mountain_car.py --settings='./paper_results/settings_learned_prior_H5.json' --save_dirpath='./paper_results/simulation_results/'
↪  --make_video=True --model_load_filepath='./paper_results/simulation_results/model_learned_prior_H5.pt'
```

The subdirectory paper_results/ contains settings and results for all the agents displayed in the paper. See subsection B.6 for a full list of command line options.

B.1 Active inference capsule

models/active_inference_capsule.py

```python
from typing import Union, Iterable

import torch
import torch.nn as nn
import torch.distributions as distr
import torch.autograd.profiler as profiler

from models.vae import VAE
from models.transition_model import PredictorGRU
from utils.timeline import Timeline


class ActiveInferenceCapsule(nn.Module):
    """
    Learns the observation (VAE), transition and prior models, and
    generates policies with minimal Free Energy of the Expected Future [1].

    Use:
        Call step() at every time-step. The method returns the action to take next.

    References:
        - [1] B. Millidge et al, "Whence the Expected Free Energy?," 2020, doi: 10.1162/neco_a_01354.
    """

    def __init__(self,
                 vae: VAE,
                 prior_model,
                 policy_dim: int,
                 time_step_size: float,
                 action_window: int,
                 planning_horizon: int,
                 n_policy_samples: int,
                 policy_iterations: int,
                 n_policy_candidates: int,
                 use_kl_intrinsic=True,  # Use for ablation studies
                 use_kl_extrinsic=True):   # Use for ablation studies
        super(ActiveInferenceCapsule, self).__init__()
        self.max_predicted_log_prob = 0.0

        # internal models
        self.vae = vae                                # p(x | y) and p(y | x)
        self.transition_model = PredictorGRU(    # p(x | π)
            latent_dim=vae.latent_dim,
            policy_dim=policy_dim,
            dynamic_dim=planning_horizon * vae.latent_dim * 2,  # Make large enough for representing trajectories (heuristic)
            num_rnn_layers=1)
        self.prior_model = prior_model        # Given or learnable prior

        # Policy settings
        self.planning_horizon = planning_horizon
        self.n_policy_samples = n_policy_samples
        self.policy_iterations = policy_iterations
        self.n_policy_candidates = n_policy_candidates
        self.action_window = action_window
        self.use_kl_intrinsic = use_kl_intrinsic
        self.use_kl_extrinsic = use_kl_extrinsic

        # Short-term memory
        self.policy = None
        self.policy_std = None      # For logging only
        self.next_FEEFs = None      # For logging only
        self.next_locs = None       # For logging only
        self.next_scales = None     # For logging only
        self.new_observations = []
        self.new_actions = []
        self.new_times = []

        # Long-term memory
        self.time_step_size = time_step_size
        self.logged_history = Timeline()

        self.reset_states()

    def reset_states(self):
        self.transition_model.reset_states()
        if not isinstance(self.prior_model, distr.Distribution):
            self.prior_model.reset_states()
```

```python
        self.policy = torch.zeros((self.planning_horizon, self.policy_dim))
        self.policy_std = torch.zeros((self.planning_horizon, self.policy_dim))
        self.next_FEEFs = torch.zeros(self.planning_horizon)
        self.next_locs = torch.zeros((self.planning_horizon, self.latent_dim))
        self.next_scales = torch.zeros((self.planning_horizon, self.latent_dim))
        self.new_observations = []
        self.new_actions = []
        self.new_times = []
        self.logged_history = Timeline()

    @property
    def observation_dim(self):
        return self.vae.observation_dim

    @property
    def latent_dim(self):
        return self.vae.latent_dim

    @property
    def policy_dim(self):
        return self.transition_model.policy_dim

    @property
    def dynamic_dim(self):
        return self.transition_model.dynamic_dim

    def step(self, time, observation: Union[torch.Tensor, Iterable], action: Union[torch.Tensor, Iterable] = None, reward: float = 0.0) -> torch.Tensor:
        """
        observation.shape = [observation_dim]
        action.shape = [action_dim]
        """
        observation = observation if isinstance(observation, torch.Tensor) else torch.tensor(observation, dtype=torch.float32).view(-1)
        self.logged_history.log(time, 'perceived_observations', observation)
        if not isinstance(self.prior_model, distr.Distribution):
            self.prior_model.learn(observation, reward)

        action = action if isinstance(action, torch.Tensor) or action is None else torch.tensor(action, dtype=torch.float32)
        if action is not None:
            self.new_actions.append(action)
            self.new_observations.append(observation)
            self.new_times.append(time)
            self.logged_history.log(time, 'perceived_actions', action)
        else:
            # The agent is passively observing and not evaluating the outcome of any policy yet
            self.new_observations = []
            self.new_actions = []
            self.transition_model.reset_states() # Invalidate the previous dynamic states

        if len(self.new_actions) == self.action_window:
            # Evaluate outcomes of last policy and draw a new policy
            new_observations = torch.stack(self.new_observations)
            new_actions = torch.stack(self.new_actions)
            with profiler.record_function("Learn observations"):
                _, new_posterior = self.perceive_observations(new_observations)  # + learning step on self.vae if in training mode
            with profiler.record_function("Retrospect_actions"):
                expected_x_mean, expected_x_std = self.transition_model.learn_policy_outcome(new_posterior, new_actions)  # + learning step on self.transition_model if in training mode
            with profiler.record_function("Sample_policy"):
                self.policy, self.policy_std, self.next_FEEFs, self.next_locs, self.next_scales = self.sample_policy()
                pred_t = time + self.time_step_size * torch.arange(0, self.planning_horizon)

            # Log all relevant variables
            expected_posterior = distr.Normal(expected_x_mean, expected_x_std)
            expected_likelihood = self.vae.decode_density(expected_posterior.mean)
            # expected_likelihood = distr.Normal(self.vae.decode(expected_posterior.mean), 1.0)
            new_VFEs = (-expected_likelihood.log_prob(new_observations) + distr.kl_divergence(expected_posterior, new_posterior)).sum(1)
            # new_VFEs = (-expected_likelihood.log_prob(new_observations) + distr.kl_divergence(expected_posterior, self.vae.px)).sum(1)
            self.logged_history.log(self.new_times, 'VFE', new_VFEs.detach())
            self.logged_history.log(self.new_times, 'perceived_locs', new_posterior.mean.detach())
            self.logged_history.log(self.new_times, 'perceived_stds', new_posterior.stddev.detach())
            self.logged_history.log(self.new_times, 'filtered_observations_locs', expected_likelihood.mean.detach())
            self.logged_history.log(self.new_times, 'filtered_observations_stds', expected_likelihood.stddev.detach())
            self.logged_history.log(self.new_times, 'observations', self.new_observations)
            prediction = Timeline()
            prediction.log(pred_t, 'policy', self.policy)
            prediction.log(pred_t, 'policy_std', self.policy_std)
            prediction.log(pred_t, 'FEEF', self.next_FEEFs)
            prediction.log(pred_t, 'pred_locs', self.next_locs)
            prediction.log(pred_t, 'pred_stds', self.next_scales)
            self.logged_history.log(time, 'predictions', prediction)

            # Reset action window percepts
            self.new_observations = []
            self.new_actions = []
```

```
            self.new_times = []

        action = self.policy[len(self.new_actions)]
        action_std = self.policy_std[len(self.new_actions)]
        self.logged_history.log(time, 'actions_loc', action)
        self.logged_history.log(time, 'actions_std', action_std)
        self.logged_history.log(time, 'expected_locs', self.next_locs[len(self.new_actions)])
        self.logged_history.log(time, 'expected_stds', self.next_scales[len(self.new_actions)])
        self.logged_history.log(time, 'expected_FEEF', self.next_FEEFs[len(self.new_actions)])
        return action

    def kl_extrinsic(self, y):
        if isinstance(self.prior_model, distr.Normal):
            kl_extrinsic = distr.kl_divergence(distr.Normal(y, 1.0), self.prior_model).sum(dim=-1)  # Sum over components, keep time-steps and batches
        else:
            # Surrogate for non-random modelled priors
            kl_extrinsic = self.prior_model.extrinsic_kl(y).sum(dim=-1)  # Sum over components, keep time-steps and batches
        return kl_extrinsic

    def _forward_policies(self, policies: torch.Tensor) -> (torch.Tensor, torch.Tensor, torch.Tensor):
        """Forward-propagate a batch of policies in time and compute their FEEFs
        Note:
            policies.shape = [planning_horizon, n_policies, policy_dim]

        References:
        - [1] B. Millidge et al, “Whence the Expected Free Energy?,” 2020, doi: 10.1162/neco_a_01354.
        """
        with profiler.record_function("Policy propagation"):
            next_x_means, next_x_stds = self.transition_model.predict(policies)
            policy_posterior = distr.Normal(next_x_means, next_x_stds)
        with profiler.record_function("FEEF"):
            with profiler.record_function("Latent reconstruction"):
                next_likelihoods = self.vae.decode_density(next_x_means)
                next_posteriors = self.vae.infer_density(next_likelihoods.mean)

            # Compute both KL components
            kl_extrinsic = self.kl_extrinsic(next_likelihoods.mean)
            kl_intrinsic = distr.kl_divergence(next_posteriors, policy_posterior).sum(dim=2)  # Sum over components, keep time-steps and batches
            # Disable components if doing an ablation study
            kl_extrinsic = kl_extrinsic if self.use_kl_extrinsic else torch.zeros_like(kl_extrinsic)
            kl_intrinsic = kl_intrinsic if self.use_kl_intrinsic else torch.zeros_like(kl_intrinsic)

            FEEFs = kl_extrinsic - kl_intrinsic
        return FEEFs, next_x_means, next_x_stds

    def sample_policy(self):
        """
        Implementation of the Cross Entropy Method.
        Similarly as done in [2].
        References:
            [2] Tschantz, A., Millidge, B., Seth, A. K., & Buckley, C. L. (2020). Reinforcement Learning through Active Inference. ArXiv. http://arxiv.org/abs/2002.12636
        """

        mean_best_policies = torch.zeros([self.planning_horizon, self.policy_dim])
        std_best_policies = torch.ones([self.planning_horizon, self.policy_dim])
        for i in range(self.policy_iterations):
            policy_distr = distr.Normal(mean_best_policies, std_best_policies)
            policies = policy_distr.sample([self.n_policy_samples, ]).transpose(0, 1)
            FEEFs, next_x_means, next_x_stds = self._forward_policies(policies.clamp(-1.0, 1.0))  # Clamp needed to prevent policy explosion, since higher magnitudes are unknown to the predictor and yield higher intrinsic value
            min_FEEF, min_FEEF_indices = FEEFs.sum(0).topk(self.n_policy_candidates, largest=False, sorted=False)  # sum over timesteps to get integrated FEEF for each policy, then pick the indices of the lowest
            mean_best_policies = policies[:, min_FEEF_indices].mean(1)
            std_best_policies = policies[:, min_FEEF_indices].std(1)

        # One last forward pass to gather the stats of the policy mean
        FEEFs, next_x_means, next_x_stds = self._forward_policies(mean_best_policies.unsqueeze(1))
        return mean_best_policies, std_best_policies, FEEFs.detach().squeeze(1), next_x_means.detach().squeeze(1), next_x_stds.detach().squeeze(1)

    def perceive_observations(self, y: torch.Tensor) -> (torch.Tensor, torch.Tensor):
        """
        Returns the variational free energy for each observation and the encoded latents
        Trains the variational autoencoder on the new observations if in training mode
        """
        if self.training:
            with profiler.record_function("Learn VAE"):
                VFE, qx_y = self.vae.learn(y)
        else:
            with profiler.record_function("Forward VAE loss"):
                VFE, qx_y = self.vae.loss(y)

        # Return last states detached from graph to avoid spurious backpropagation in other functions
        qx_y.loc.detach_()
```

```
        qx_y.scale.detach_()
        return VFE.detach(), qx_y
```

B.2 Transition model

models/transition_model.py

```
import torch
import torch.nn as nn
import torch.distributions as distr
import torch.optim as optim
import torch.autograd.profiler as profiler


class PredictorGRU(nn.Module):
    """
    Predictor model for latent state sequences conditioned on a policy (i.e., a sequence of control actions)

    Args:
        latent_dim: number of dimensions of the latent space
        policy_dim: number of dimensions of the policy space
        dynamic_dim: number of dimensions of the dynamic states (i.e., hidden states of the `RNN`)
        num_rnn_layers: number of RNN layers
    """

    def __init__(self, latent_dim: int, policy_dim: int, dynamic_dim: int, num_rnn_layers: int):
        super(PredictorGRU, self).__init__()
        # Hyperparameters
        self.latent_dim = latent_dim
        self.policy_dim = policy_dim
        self.dynamic_dim = dynamic_dim
        self.num_rnn_layers = num_rnn_layers

        # Neural networks:
        self.gru = nn.GRU(input_size=latent_dim + policy_dim, hidden_size=dynamic_dim, num_layers=num_rnn_layers, dropout=0.05 if num_rnn_layers > 1 else 0)
        self.mean_net = nn.Linear(dynamic_dim, latent_dim)
        self.std_net = nn.Linear(dynamic_dim, latent_dim)
        self.optimizer = optim.Adam(self.parameters(), lr=0.001)

        # States
        self.prev_dyn_state = None  # The dynamical states before the latest observation
        self.latest_latent = None     # The latent state corresponding to the latest observation
        self.reset_states()

    def reset_states(self):
        self.prev_dyn_state = torch.zeros((self.num_rnn_layers, 1, self.dynamic_dim))
        self.latest_latent = torch.zeros((1, 1, self.latent_dim))

    def forward(self, x: torch.Tensor, policy: torch.Tensor, dyn: torch.Tensor = None):
        """
        Predicts the latent state one time-step ahead
        :param x: [sequence_length, n_policies, latent_dim]
        :param policy: [sequence_length, n_policies, policy_dim]
        :param dyn: An initial dynamic state (e.g., from the previous call). Shape: [num_rnn_layers, n_policies, dynamic_dim]
        :return: the mean and variance of the density of the predicted latent, and the dynamic state
        """
        pred, dyn = self.gru(torch.cat((x, policy), dim=2), dyn)
        return self.mean_net(pred) + x, nn.functional.softplus(self.std_net(pred)) + 1e-6, dyn

    def predict(self, policy: torch.Tensor):
        """
        Predicts the latent state several time-steps ahead from the first_latent under the provided policy
        :param policy: [sequence_length, n_policies, policy_dim]
        :return: the mean and standard deviation of the predicted latents
        """
        steps, n_policies = policy.shape[:2]
        next_mu = []
        next_var = []

        # If policies are batched, use the same states for each batch
        if n_policies == 1:
            prev_dyn = self.prev_dyn_state
            first_latent = self.latest_latent
        else:
            prev_dyn = self.prev_dyn_state.expand((self.num_rnn_layers, n_policies, self.dynamic_dim))
            first_latent = self.latest_latent.expand((1, n_policies, self.latent_dim))

        for i in range(steps):
            latent_mean, latent_std, dyn = self(first_latent if i == 0 else next_mu[i - 1], policy[[i]], prev_dyn)
```

```python
            next_mu.append(latent_mean)
            next_var.append(latent_std)
            prev_dyn = dyn.detach()

        return torch.cat(next_mu), torch.cat(next_var)

    def learn_policy_outcome(self, px_y: distr.Distribution, policy: torch.Tensor):
        """
        Updates the dynamic state and trains the neural networks if in training mode.
        :param px_y: a distribution over a sequence of latent states. [sequence_length, latent_dim]
        :param policy: a sequence of actions preceding the observations. [sequence_length, policy_dim]
        :param
        """

        with torch.no_grad():
            prior_latents = torch.cat((self.latest_latent, px_y.mean[:-1].unsqueeze(1)), dim=0)  # A sequence of latents that are prior to performing the actions

        with profiler.record_function("Retrospection"):
            pred_x_mean, pred_x_std, prev_dyn_state = self(prior_latents, policy.unsqueeze(1), self.prev_dyn_state)

        if self.training:
            with profiler.record_function("Transition backprop"):
                self.optimizer.zero_grad()
                pred_px_y = distr.Normal(pred_x_mean.squeeze(1), pred_x_std.squeeze(1))
                loss = distr.kl_divergence(pred_px_y, px_y).sum()  # Total divergence is sum of divergences for each latent component at every time-step
                loss.backward()
                self.optimizer.step()

        self.prev_dyn_state = prev_dyn_state.detach()
        self.latest_latent = px_y.mean[-1].reshape((1, 1, self.latent_dim)).detach()  # The result of the last action. Not included in the dynamic state yet

        return pred_x_mean.squeeze(1).detach(), pred_x_std.squeeze(1).detach()
```

B.3 Variational autoencoder

models/vae.py

```python
# Base VAE class definition

import torch
import torch.nn as nn
import torch.distributions as distr

"""
Base class for variational autoencoders
Adapted from https://github.com/iffsid/mmvae
"""


class VAE(nn.Module):
    def __init__(self, enc, dec, observation_dim, latent_dim):
        super(VAE, self).__init__()
        self.enc = enc
        self.dec = dec
        self._latent_dim = latent_dim
        self._observation_dim = observation_dim
        self.modelName = None
        self._px_params = None  # defined in subclass
        self.llik_scaling = 1.0
        self.data_shape = None  # defined in subclass, e.g., [-1, 3, 420, 600]

    @property
    def px(self):
        return distr.Normal(*self._px_params)

    @property
    def latent_dim(self):
        return self._latent_dim

    @property
    def observation_dim(self):
        return self._observation_dim

    def forward(self, y) -> (distr.Distribution, distr.Distribution, torch.Tensor):
        qx_y_params = self.enc(y)
        qx_y = distr.Normal(*qx_y_params)
        x_samples = qx_y.rsample()
        py_x = distr.Normal(*self.dec(x_samples))
        return qx_y, py_x, x_samples
```

```python

    def loss(self, y):
        qx_y, py_x, latents = self(y)
        lpy_x = py_x.log_prob(y) * self.llik_scaling
        kld = distr.kl_divergence(qx_y, self.px)
        VFEs = kld.sum(-1) - lpy_x.sum(-1)  # Computes variational free energy of each observation
        return VFEs, qx_y

    def learn(self, y):
        pass

    def generate(self, n_samples):
        latents = self.px.rsample(torch.Size([n_samples]))
        return self.decode(latents)

    def reconstruct(self, data):
        latents = self.infer(data)
        recon = self.decode(latents)
        return recon

    def infer_density(self, data):
        return distr.Normal(*self.enc(data))

    def decode_density(self, latents):
        return distr.Normal(*self.dec(latents))

    def decode(self, latents):
        recon = self.decode_density(latents).mean
        return recon

    def infer(self, data):
        latents = self.infer_density(data).mean
        return latents
```

`models/vae_dense.py` (variational autoencoder with densely-connected layers)

```python
import torch
import torch.nn as nn
import torch.optim as optim

from models.vae import VAE
from utils.silu import SiLU


class Enc(nn.Module):
    def __init__(self, observation_dim, latent_dim):
        super(Enc, self).__init__()
        self.enc = nn.Sequential(
            nn.Linear(observation_dim, latent_dim * 10),
            SiLU()
        )
        self.c1 = nn.Linear(latent_dim * 10, latent_dim)
        self.c2 = nn.Linear(latent_dim * 10, latent_dim)

    def forward(self, y):
        e = self.enc(y)
        return self.c1(e), nn.functional.softplus(self.c2(e)) + 1e-6


class Dec(nn.Module):
    def __init__(self, observation_dim, latent_dim, observation_noise_std=None):
        super(Dec, self).__init__()
        if observation_noise_std is None or observation_noise_std is False:
            self.observation_noise_std = torch.full([observation_dim], 0.05)
        elif isinstance(observation_noise_std, float):
            self.observation_noise_std = torch.full([observation_dim], observation_noise_std)
        elif isinstance(observation_noise_std, (tuple, list, torch.Tensor)):
            assert len(observation_noise_std) == observation_dim
            self.observation_noise_std = torch.tensor(observation_noise_std, dtype=torch.float32)
        else:
            raise TypeError('Expected None, False, float, tuple, list or torch.Tensor for observation_noise_std')

        self.dec = nn.Sequential(
            nn.Linear(latent_dim, latent_dim * 10),
            SiLU()
        )
        self.c1 = nn.Linear(latent_dim * 10, observation_dim)

    def forward(self, x):
        d = self.dec(x)
```

```python
        return self.c1(d), self.observation_noise_std


class DenseObservation_VAE(VAE):
    def __init__(self, observation_dim, latent_dim, observation_noise_std=None):
        super(DenseObservation_VAE, self).__init__(
            enc=Enc(observation_dim, latent_dim),
            dec=Dec(observation_dim, latent_dim, observation_noise_std),
            observation_dim=observation_dim,
            latent_dim=latent_dim
        )
        self._px_params = [torch.zeros(1, latent_dim),  # mu
                           torch.ones(1, latent_dim) * 0.3]  # var
        self.modelName = 'Dense observation VAE'
        self.data_shape = [-1, observation_dim]
        self.llik_scaling = latent_dim / observation_dim  # Scale factor for the log-likelihood in the loss function
        self.optimizer = optim.Adam(self.parameters(), lr=0.001)

    def learn(self, y):
        self.optimizer.zero_grad()
        VFE, qx_y = self.loss(y)
        VFE.mean().backward()
        self.optimizer.step()
        qx_y.loc.detach_()
        qx_y.scale.detach_()
        return VFE.detach(), qx_y
```

B.4 Learned prior

models/prior_model.py

```python
import torch
import torch.nn as nn
import torch.optim as optim

from utils.silu import SiLU


class PriorModelBellman(nn.Module):
    def __init__(self, observation_dim, learning_rate=0.001, iterate_train=1, discount_factor=0.99):
        super(PriorModelBellman, self).__init__()
        self.observation_dim = observation_dim
        self.learning_rate = learning_rate
        self.iterate_train = iterate_train
        self.discount_factor = discount_factor
        self.nn = nn.Sequential(
            nn.Linear(observation_dim, observation_dim * 20),
            SiLU(),
            nn.Linear(observation_dim * 20, observation_dim),
            nn.Tanh()
        )
        self.optimizer = optim.SGD(self.parameters(), lr=self.learning_rate)
        self.observations = []
        self.rewards = []

    def reset_states(self):
        self.observations = []
        self.rewards = []

    def forward(self, y):
        return self.nn(y)

    def extrinsic_kl(self, y):
        return 1.0 - self.forward(y) # map from [-1, 1] to [2, 0]

    def learn(self, y: torch.Tensor, r: float):
        self.observations.append(y)
        self.rewards.append(r)
        if abs(r) > 0.1:
            observations = torch.stack(self.observations)
            rewards = torch.tensor(self.rewards)
            rewards = rewards.expand((self.observation_dim, observations.shape[0])).transpose(0, 1)
            for i in range(self.iterate_train):
                disc = torch.pow(self.discount_factor, torch.arange(observations.shape[0], dtype=torch.float32).flip(0))
                disc = disc.expand((self.observation_dim, observations.shape[0])).transpose(0, 1)  # Apply same for each observation dim
                forward_ = self.forward(observations)
                with torch.no_grad():
                    pred_next_v = torch.cat([forward_[1:], torch.zeros((1, self.observation_dim))], dim=0)
                    r_ = rewards + disc * pred_next_v
```

```python

                self.optimizer.zero_grad()
                for g in self.optimizer.param_groups:
                    g['lr'] = self.learning_rate * abs(r)
                loss = nn.functional.mse_loss(forward_, r_)
                loss.backward()
                self.optimizer.step()
```

B.5 Training script

`mountain_car/training.py`

```python
import os
import sys
import json

import gym
import torch
import torch.distributions as distr
import numpy as np
import tqdm

from models.active_inference_capsule import ActiveInferenceCapsule
from utils.timeline import Timeline
from utils.value_map import ValueMap
from utils.model_saving import load_capsule_parameters
from models.vae_dense_obs import DenseObservation_VAE
from models.prior_model import PriorModelBellman
import mountain_car.plotting as plots


def args_to_simulation_settings(args):
    """
    Converts command line arguments into a settings dictionary to construct the agents and run simulations.
    """
    with open(args.settings, 'r') as f:
        settings = json.load(f)

    if 'PriorModelBellman' in settings['agent']['prior_model']:
        prior_model = PriorModelBellman(**settings['agent']['prior_model']['PriorModelBellman'])
    elif 'Normal' in settings['agent']['prior_model']:
        prior_model = distr.Normal(torch.tensor(settings['agent']['prior_model']['Normal']['loc']), torch.tensor(settings['agent']['prior_model']['Normal']['std']))
    else:
        raise KeyError("Unknown prior_model. Make sure it's either PriorModelBellman or Normal")

    settings['agent']['prior_model'] = prior_model
    settings['agent']['vae'] = DenseObservation_VAE(**settings['agent']['vae'])
    settings['simulation']['episode_callbacks'] = [plots.show_phase_portrait, plots.show_prediction_vs_outcome] if args.display_plots else []
    if args.load_existing:
        if args.model_load_filepath != '':
            filepath = args.model_load_filepath
        else:
            filepath = os.path.join(args.save_dirpath, f'model_{settings["experiment_name"]}.pt')

        if os.path.exists(filepath):
            settings['simulation']['model_load_filepath'] = filepath
        else:
            raise FileNotFoundError(f'Previous model <{filepath}> not found.')
    else:
        settings['simulation']['model_load_filepath'] = None
    settings['simulation']['save_dirpath'] = None if args.save_dirpath == '' else args.save_dirpath
    settings['simulation']['model_name'] = settings['experiment_name']
    settings['simulation']['save_all_episodes'] = args.save_all_episodes
    settings['simulation']['verbose'] = args.verbose
    settings['simulation']['display_simulation'] = args.display_simulation
    return settings


def make_model_filepath(dirpath, name, instance=None, episode=None):
    return os.path.join(dirpath, "model_{}{}{}.pt".format(name,
                                                          f"_id{instance}" if instance is not None else '',
                                                          f"_ep{episode:03d}" if episode is not None else ''))


def run_training(agent_parameters,
                 time_compression,
                 observation_noise_std=None,
                 include_cart_velocity=True,
                 model_id=None,
```

```
                 hot_start_episodes=0,
                 episodes=1,
                 episode_callbacks=(),
                 frame_callbacks=(),
                 save_dirpath=None,
                 model_name='',
                 model_load_filepath=None,
                 save_all_episodes=False,
                 load_vae=True,
                 load_transition_model=True,
                 load_prior_model=True,
                 train_parameters=True,
                 verbose=True,
                 display_simulation=False):
    """
    Main training routine that trains an agent over a number of episodes on the mountain car environment.
    """

    env = gym.make('MountainCarContinuous-v0').env
    observations_mapper = ValueMap(in_min=torch.tensor((-1.2, -0.07)), in_max=torch.tensor((0.6, 0.07)),
                                   out_min=torch.tensor((-1.0, -1.0)), out_max=torch.tensor((1.0, 1.0)))
    rewards_mapper = ValueMap(in_min=-100, in_max=100, out_min=-1, out_max=1)
    aif_agent = ActiveInferenceCapsule(**agent_parameters)

    # Load previous model
    if model_load_filepath is not None:
        load_capsule_parameters(aif_agent, model_load_filepath, load_vae, load_transition_model, load_prior_model)
        if verbose:
            loaded_models = []
            loaded_models += ['vae'] if load_vae else []
            loaded_models += ['transition_model'] if load_transition_model else []
            loaded_models += ['prior_model'] if load_prior_model and not isinstance(aif_agent.prior_model, distr.Normal) else []
            print(f"\nLoaded <{', '.join(loaded_models)}> from previous save at <{model_load_filepath}>")

    if save_dirpath is not None and train_parameters:
        torch.save(aif_agent.state_dict(), make_model_filepath(save_dirpath, model_name, model_id, 0 if save_all_episodes else None)) # save episode 0 (no training yet)

    use_kl_intrinsic = aif_agent.use_kl_intrinsic
    use_kl_extrinsic = aif_agent.use_kl_extrinsic
    if not train_parameters:
        aif_agent.eval()
    training_history = Timeline()
    max_episode_steps = 1000
    for episode in range(episodes):
        env.reset()
        aif_agent.reset_states()
        if episode < hot_start_episodes:
            aif_agent.use_kl_intrinsic = True
            aif_agent.use_kl_extrinsic = True
        else:
            aif_agent.use_kl_intrinsic = use_kl_intrinsic
            aif_agent.use_kl_extrinsic = use_kl_extrinsic
        observations_mapper = observations_mapper if observations_mapper is not None else lambda x: x
        state = observations_mapper(torch.from_numpy(env.state).float())
        state_noisy = state if observation_noise_std is None else state + torch.normal(0.0, torch.tensor(observation_noise_std))
        action = aif_agent.step(0, state_noisy if include_cart_velocity else state_noisy[[0]])
        total_reward = 0
        episode_history = Timeline()
        episode_history.log(0, 'true_observations', state)
        episode_history.log(0, 'noisy_observations', state_noisy)
        iterator = tqdm.tqdm(range(max_episode_steps), file=sys.stdout, disable=not verbose)
        iterator.set_description(f'Running episode {episode}/{episodes}')
        for i in iterator:
            if display_simulation:
                frame = env.render(mode='rgb_array')
            else:
                frame = None
            t = i + 1
            observation, reward, done, _ = env.step(action)
            observation = observations_mapper(torch.from_numpy(observation).float())
            reward = rewards_mapper(reward)
            episode_history.log(t, 'true_observations', observation)
            episode_history.log(t - 1, 'true_actions', action)
            obs_noise = observation if observation_noise_std is None else observation + torch.normal(0.0, torch.tensor(observation_noise_std))
            episode_history.log(t, 'noisy_observations', obs_noise)
            if i % time_compression == 0:
                action = aif_agent.step(t, obs_noise if include_cart_velocity else obs_noise[[0]], action, reward)
                action = np.clip(action, env.min_action, env.max_action)
            total_reward += reward

            for callback in frame_callbacks:
                callback(**dict(agent=aif_agent,
                                env=env,
```

```
                                episode_reward=reward,
                                episode_history=episode_history,
                                observations_mapper=observations_mapper,
                                frame=frame))

            if done:
                if i % time_compression != 0:  # if last state did not fall in the update, update anyways
                    aif_agent.step(t, obs_noise if include_cart_velocity else obs_noise[[0]], action=None, reward=reward)
                iterator.set_postfix_str(f'reward={total_reward:.2f}')
                break
            elif i == max_episode_steps - 1:
                iterator.set_postfix_str(f'reward={total_reward:.2f}')

        for callback in episode_callbacks:
            callback(**dict(agent=aif_agent,
                            env=env,
                            episode_reward=total_reward,
                            episode_history=episode_history,
                            observations_mapper=observations_mapper))

        if save_dirpath is not None and train_parameters:
            torch.save(aif_agent.state_dict(), make_model_filepath(save_dirpath, model_name, model_id, episode if save_all_episodes else None))
        VFE, expected_FEEF = aif_agent.logged_history.select_features(['VFE', 'expected_FEEF'])[1]
        training_history.log(episode, 'cumulative_VFE', sum(VFE).item())
        training_history.log(episode, 'cumulative_FEEF', sum(expected_FEEF).item())
        training_history.log(episode, 'steps_per_episode', len(episode_history.times))
        training_history.log(episode, 'rewards', total_reward)

    env.close()
    return training_history
```

B.6 Main script

run_capsule_mountain_car.py

```
import os
import re
import sys
import copy
import glob
import shutil
import pickle
import argparse
import subprocess
from time import time
import multiprocessing

from tqdm import tqdm
import matplotlib.pyplot as plt

from mountain_car.training import args_to_simulation_settings
from mountain_car.training import run_training
import mountain_car.plotting as plots

"""
This is the main script, which accepts options from the command line and can
    - Train a single agent: showing training progress and optionally plotting insights for each episode
    - Train a batch of agents: spawns simulations on parallel threads and collects training statistics
    - Generate a video: takes a trained model and generates a .mp4 video of one episode
"""

parser = argparse.ArgumentParser()
parser.add_argument("--settings", type=str, default="./paper_results/settings_learned_prior_H5.json")
parser.add_argument("--batch_agents", type=int, default=1)
parser.add_argument("--max_cpu", type=int, default=-1)
parser.add_argument("--make_video", type=bool, default=False)
parser.add_argument("--display_plots", type=bool, default=False)
parser.add_argument("--load_existing", type=bool, default=False)
parser.add_argument("--save_dirpath", type=str, default='./paper_results/simulation_results/')
parser.add_argument("--model_load_filepath", type=str, default='')
parser.add_argument("--save_all_episodes", type=bool, default=False)
parser.add_argument("--verbose", type=bool, default=True)
parser.add_argument("--display_simulation", type=bool, default=False)
args = parser.parse_args()
args.load_existing = args.make_video or args.load_existing # make sure we are loading a model when making a video
settings = args_to_simulation_settings(args)


# ------- TRAINING A SINGLE AGENT ---------
```

```python
def train_single_agent():
    global settings
    global args
    run_training(
        agent_parameters=settings['agent'],
        **settings['simulation']
    )


def _run_training_process(training_id):
    global settings
    global args
    settings_copy = copy.deepcopy(settings)
    settings_copy['simulation']['verbose'] = False
    settings_copy['simulation']['model_id'] = training_id
    return run_training(agent_parameters=settings_copy['agent'], **settings_copy['simulation'])


# ------- TRAINING A BATCH OF AGENTS ---------
def train_many_agents():
    global settings
    global args
    num_cpu_cores = args.max_cpu if args.max_cpu > 0 else multiprocessing.cpu_count()
    num_processes = min(args.batch_agents, num_cpu_cores)
    if not os.path.exists(args.save_dirpath):
        os.makedirs(args.save_dirpath)

    print(f'\nRunning {args.batch_agents} simulations of {settings["experiment_name"]} in {num_processes} parallel processes...')
    t0 = time()
    with multiprocessing.Pool(processes=num_processes) as pool:
        all_results = []
        for new_result in tqdm(pool.imap_unordered(_run_training_process, range(args.batch_agents)), total=args.batch_agents, file=sys.stdout):
            all_results.append(new_result)
            with open(os.path.join(args.save_dirpath, f'results_{settings["experiment_name"]}.pickle'), 'wb') as f:
                pickle.dump(all_results, f)
    t1 = time() - t0
    print(f'Finished {settings["simulation"]["episodes"] * args.batch_agents} episodes in {t1/60:.1f} minutes ({t1 / (settings["simulation"]["episodes"] *
    ↪  args.batch_agents):.2f}s/episode)')


# ------- MAKING A VIDEO ---------
def make_video():
    global settings
    global args
    if settings['simulation']['model_load_filepath'] is None or not os.path.exists(settings['simulation']['model_load_filepath']):
        raise RuntimeError('Provide a trained model through model_load_filepath for generating a video')

    frames_path = os.path.join(args.save_dirpath, f'frames_{settings["experiment_name"]}')
    video_path = os.path.join(args.save_dirpath, f'video_{settings["experiment_name"]}.mp4')

    if not os.path.exists(frames_path):
        os.makedirs(frames_path)  # Make a directory to save the frame images

    # If a video was previously made for the same model, delete it and clear all frames
    for file_name in glob.glob(os.path.join(frames_path, '*.png')):
        os.remove(file_name)
    if os.path.exists(video_path):
        os.remove(video_path)

    # Callback function to draw each episode frame
    def save_video_frame(agent, episode_history, observations_mapper, frame, **kwargs):
        fig = plots.make_video_frame(agent, episode_history, frame, observations_mapper)
        match = re.search(r'_ep(\d*).pt', settings['simulation']['model_load_filepath'])
        if match:  # the model contains the episode number
            fig.subplots_adjust(top=0.88)
            fig.suptitle(f'Episode {int(match.group(1))}', y=0.98)
        fig.savefig(os.path.join(frames_path, f'frame_{len(episode_history.times):04d}.png'), dpi=200)
        plt.close(fig)

    settings['simulation']['observation_noise_std'] = settings['simulation']['observation_noise_std'] or 0.05  # When noise is None, set it to 0.05 for display
    settings['simulation']['frame_callbacks'] = [save_video_frame]
    settings['simulation']['display_simulation'] = True
    settings['simulation']['train_parameters'] = False
    settings['simulation']['episodes'] = 1
    # Run the episode, rendering and saving each frame
    run_training(
        agent_parameters=settings['agent'],
        **settings['simulation']
    )

    # Freeze the last second of video by repeating the last frame fps times
    last_frame = sorted(glob.glob(os.path.join(frames_path, f'frame_*.png')))[-1]
    last_id = int(last_frame[-8:-4])
    for i in range(1, 25):
```

```python
        shutil.copy(last_frame, os.path.join(frames_path, f'frame_{last_id + i:04d}.png'))

    # Generate video from saved frames
    subprocess.call([
        'ffmpeg',
        '-i', os.path.join(frames_path, 'frame_%04d.png'),  # input images
        '-r', '25',  # output frame rate
        '-pix_fmt', 'yuv420p',
        '-b', '5000k',  # 5Mb bitrate
        video_path
    ])


if __name__ == "__main__":
    if args.batch_agents == 1 and args.make_video is False:
        train_single_agent()
    elif args.batch_agents > 1 and args.make_video is False:
        train_many_agents()
    elif args.batch_agents == 1 and args.make_video is True:
        make_video()
    elif args.batch_agents < 1:
        raise RuntimeError('At least one agent requires (batch_agents=1)')
    else:
        raise RuntimeError('Cannot make video with batch_agents != 1')
```

B.7 Plotting functions

mountain_car/plotting.py

```python
from typing import List, Union
import os

import matplotlib.pyplot as plt
import matplotlib.patches as patches
from mpl_toolkits.axes_grid1 import make_axes_locatable
import torch
import numpy as np

from models.active_inference_capsule import ActiveInferenceCapsule
from utils.timeline import Timeline
from utils.signal_smoothing import smooth


def _plot_observations_actions(axis, agent: ActiveInferenceCapsule, merged_history: Timeline):
    # 1) Plot policy
    times_act_loc, (act_loc, act_std) = merged_history.select_features(['actions_loc', 'actions_std'])
    act_loc, act_std = torch.stack(act_loc).view(-1), torch.stack(act_std).view(-1)
    axis.fill_between(times_act_loc, act_loc - act_std, act_loc + act_std, color='r', alpha=0.3, linewidth=0)
    pl_pol = axis.plot(times_act_loc, act_loc, 'r--', linewidth=1, label='Policy')

    # 2) Plot executed actions
    times_actions, true_actions = merged_history.select_features('true_actions')
    pl_act = axis.plot(times_actions, true_actions, 'b-.', linewidth=1, label='Executed action')

    axis.set_ylabel('action', color='r', rotation=90)
    axis.tick_params(axis='y', labelcolor='r')
    axis_obs = axis.twinx()

    # 3) Plot true observations
    times_true_observations, true_observations = merged_history.select_features('true_observations')
    times_noisy_observations, noisy_observations = merged_history.select_features('noisy_observations')
    true_observations = torch.stack(true_observations)
    noisy_observations = torch.stack(noisy_observations)
    pl_pos = axis_obs.plot(times_true_observations, true_observations[:, 0], color='k', linewidth=1.0, label='true position')
    pl_pos_noise = axis_obs.plot(times_noisy_observations, noisy_observations[:, 0], color='k', linestyle='--', linewidth=1.0, label='noisy observation')
    # 4) Plot expected observations
    times_filtered, (filtered_locs, filtered_stds) = merged_history.select_features(['filtered_observations_locs', 'filtered_observations_stds'])
    if len(times_filtered) > 0:
        locs_, stds_ = torch.stack(filtered_locs)[:, 0], torch.stack(filtered_stds)[:, 0]
        axis_obs.fill_between(times_filtered, locs_ + stds_, locs_ - stds_, color='k', alpha=0.3)
    else:
        locs_ = []
    pl_rec = axis_obs.plot(times_filtered, locs_, 'k', linestyle='dotted', linewidth=1.0, label='Likelihood $p(y\mid x)$')
    axis_obs.set_ylabel('position', rotation=90)  # we already handled the x-label with ax1

    # Make common legend for both axes
    lns = pl_pol + pl_act + pl_pos + pl_pos_noise + pl_rec
    labs = [ln.get_label() for ln in lns]
    axis.legend(lns, labs, loc='lower right', framealpha=0.4)
```

```python
    axis.set_ylim((-2.5, 2.5))
    axis.set_yticks([-2, -1, 0, 1, 2])
    axis_obs.set_ylim((-1.1, 1.1))
    axis.grid(linewidth=0.5, alpha=0.5)
    axis.set_title('Observations and policy')

    return axis


def _plot_latent_prediction(axis, latent_idx, merged_history: Timeline):
    # 1) plot prediction tubes
    times_replanning, predictions = merged_history.select_features('predictions')
    for j, prediction in enumerate(predictions):
        times_pred, (pred_locs, pred_stds) = prediction.select_features(['pred_locs', 'pred_stds'])
        pred_locs, pred_stds = torch.stack(pred_locs)[:, latent_idx], torch.stack(pred_stds)[:, latent_idx]  # convert to tensor and select latent dimension i
        axis.fill_between(times_pred, pred_locs - pred_stds, pred_locs + pred_stds, color='k', alpha=0.1, linewidth=1)
        axis.plot(times_pred, pred_locs, 'k--', alpha=0.7, linewidth=0.3, label='Predicted latent' if j == 1 else None)

    # 2) plot expected latents
    times_expectations, (expected_locs, expected_stds) = merged_history.select_features(['expected_locs', 'expected_stds'])
    expected_locs, expected_stds = torch.stack(expected_locs)[:, latent_idx], torch.stack(expected_stds)[:, latent_idx]  # convert to tensor and select latent dimension i
    axis.fill_between(times_expectations, expected_locs - expected_stds, expected_locs + expected_stds, color='r', linewidth=0, alpha=0.3)
    axis.plot(times_expectations, expected_locs, color=(0.8, 0, 0), linestyle='--', linewidth=1.0, alpha=0.8, label='Expected latent')

    # 3) Plot perceived latents
    times_percepts, (perceived_locs, perceived_stds) = merged_history.select_features(['perceived_locs', 'perceived_stds'])
    perceived_locs, perceived_stds = torch.stack(perceived_locs)[:, latent_idx], torch.stack(perceived_stds)[:, latent_idx]  # convert to tensor and select latent dimension i
    axis.fill_between(times_percepts, perceived_locs - perceived_stds, perceived_locs + perceived_stds, color='b', linewidth=0, alpha=0.4)
    axis.plot(times_percepts, perceived_locs, 'b', linewidth=1.0, label='Perceived latent')

    axis.grid(linewidth=0.5, alpha=0.5)
    axis.set_title(f'Latent {latent_idx + 1}')
    axis.legend(loc='lower right', framealpha=0.3)


def _plot_phase_portrait(fig, axis, agent: ActiveInferenceCapsule, episode_history: Timeline, observations_mapper, label_cbar=True, show_t_goal=False, **kwargs):
    axis.set_aspect(1.0)
    grid_points = 30
    # 1) Plot heat map of extrinsic KL divergence
    sample_positions = torch.linspace(-1.0, 1.0, grid_points)  # positions in the agent space (-1.2, 0.6) -> (-1.0, 1.0)
    sample_velocities = torch.linspace(-1.0, 1.0, grid_points)  # positions in the agent space (-1.2, 0.6) -> (-1.0, 1.0)
    kl_extrinsic = torch.zeros((grid_points, grid_points))
    with torch.no_grad():
        for i in range(grid_points):
            for j in range(grid_points):
                if agent.observation_dim == 1:  # Case where the agent cannot observe the velocity
                    kl_extrinsic[i, j] = agent.kl_extrinsic(sample_positions[[j]]).sum()
                else:
                    kl_extrinsic[i, j] = agent.kl_extrinsic(torch.stack((sample_positions[j], sample_velocities[i]))).sum()

    kl_bar_max = 1.0 if kl_extrinsic.max() < 1.5 else 2.0
    clev = torch.linspace(0.0, max(kl_bar_max, kl_extrinsic.max().item()), 100)
    cs = axis.contourf(sample_positions.expand((grid_points, grid_points)), sample_velocities.expand((grid_points, grid_points)).transpose(1, 0), kl_extrinsic, clev, cmap='magma_r')
    for c in cs.collections:
        c.set_rasterized(True)

    divider = make_axes_locatable(axis)
    cax = divider.append_axes("right", size="5%", pad=0.05)
    cbar = fig.colorbar(cs, cax=cax, ticks=torch.linspace(0.0, kl_bar_max, 5))
    if label_cbar:
        cbar.set_label('KL extr.')

    # 2) Plot trajectories
    times_true_observations, true_observations = episode_history.select_features('true_observations')
    true_observations = torch.stack(true_observations)
    axis.plot(true_observations[:, 0], true_observations[:, 1], color=(0.5, 0.5, 1.0), linewidth=1.5)
    axis.scatter([true_observations[0, 0]], [true_observations[0, 1]], color='r')

    if agent.observation_dim == 2:
        perceived_locs = agent.logged_history.select_features('perceived_locs')[1]
        if len(perceived_locs) > 1:  # Skip if no planning done yet
            perceived_locs = torch.stack(perceived_locs)
            perceived_observations = agent.vae.decode(perceived_locs).detach()
            axis.plot(perceived_observations[:, 0], perceived_observations[:, 1], color=(0.0, 1.0, 0.0), linestyle='--', linewidth=1.5)

        # Plot predicted trajectories
        times_predictions, predictions = agent.logged_history.select_features('predictions')
        for j, prediction in enumerate(predictions):
            _, pred_locs = prediction.select_features('pred_locs')
            pred_locs = torch.stack(pred_locs)
            perceived_locs = agent.logged_history.get_frame(times_predictions[j])['perceived_locs']
```

```python
            # pred_locs = torch.stack(pred_locs)
            pred_obs = agent.vae.decode(torch.cat([perceived_locs.unsqueeze(0), pred_locs])).detach()
            axis.plot(pred_obs[:, 0], pred_obs[:, 1], color=(0.0, 0.9, 0.0), alpha=0.5, linewidth=0.5, label='Perceived latent' if j == 1 else None)

    # 3) Plot goal box
    x1, y1 = observations_mapper(torch.tensor((0.45, 0.0)))  # thresholds for mountain-car goal, transformed to problem coordinates
    x2, y2 = 1.0, 1.0
    axis.add_patch(patches.Polygon([[x1, y1], [x2, y1], [x2, y2], [x1, y2]], edgecolor='k', facecolor=(0.6, 0.6, 0.6), linestyle='--', alpha=0.5))

    axis.set_xlabel('Horizontal position')
    axis.set_ylabel('Velocity', labelpad=-3)
    axis.set_title('Phase portrait with extrinsic value')
    if show_t_goal:
        axis.text(0.1, -0.9, f'$\mathrm{{t_{{goal}}}}={int(times_true_observations[-1])}$', backgroundcolor=(1.0, 1.0, 1.0, 0.4))
    axis.set_xlim([-1, 1])
    axis.set_ylim([-1, 1])
    axis.grid(color=(0.5, 0.5, 0.5), alpha=0.5, linewidth=0.2)
    return axis


def show_prediction_vs_outcome(agent: ActiveInferenceCapsule, episode_history: Timeline, **kwargs):
    fig = plt.figure(figsize=(8, 6))
    merged_history = episode_history.merge(agent.logged_history)

    # 1) Plot actions and states
    ax2 = fig.add_subplot(agent.latent_dim + 1, 1, 1)
    _plot_observations_actions(ax2, agent, merged_history)

    for i in range(agent.latent_dim):
        ax1 = fig.add_subplot(agent.latent_dim + 1, 1, i + 2)
        _plot_latent_prediction(ax1, i, merged_history)
        ax2.set_xlim(*ax1.get_xlim())  # Make sure the timelines match between the action-state plot and the latent-predictions plots

    fig.tight_layout()
    plt.show()
    plt.close()


def show_phase_portrait(agent: ActiveInferenceCapsule, episode_history: Timeline, observations_mapper, **kwargs):
    fig = plt.figure(figsize=(5, 4))
    axis = fig.gca()
    _plot_phase_portrait(fig, axis, agent, episode_history, observations_mapper)
    axis.set_title('Given prior, H=90')
    fig.tight_layout()
    plt.savefig('phase_portrait.pdf')
    plt.show()
    plt.close()


def show_FEEF_vs_FE(agent: ActiveInferenceCapsule, **kwargs):
    fig = plt.figure(figsize=(5, 4))
    ax = fig.gca()
    times_FEEF, expected_FEEF = agent.logged_history.select_features('expected_FEEF')
    times_FE, VFE = agent.logged_history.select_features('VFE')
    expected_FEEF = torch.stack(expected_FEEF).view(-1)
    VFE = torch.stack(VFE).view(-1)
    ax.plot(times_FE, VFE, 'b-', label='VFE')
    ax.plot(times_FEEF, expected_FEEF, 'r--', label='FEEF')
    ax.legend()
    fig.tight_layout()
    plt.show()
    plt.close()


def plot_training_history(timelines: Union[Timeline, List[Timeline]], save_path=None, show=True, ax=None, label=None, color=(0.2, 0.4, 1.0), linestyle='-',
    alpha=0.3, smoothing=0, hotstart_tmlns=None):
    timelines = [timelines] if isinstance(timelines, Timeline) else timelines
    all_rewards = []
    times = None
    for timeline in timelines:
        times, durations = timeline.select_features('steps_per_episode')
        all_rewards.append(durations)

    all_rewards = np.array(all_rewards)
    r_mean = smooth(all_rewards.mean(0), smoothing)
    r_std = all_rewards.std(0)
    r_max = smooth(np.array([min(a, b) for a, b in zip(r_mean + r_std, all_rewards.max(0))]), smoothing)
    r_min = smooth(np.array([max(a, b) for a, b in zip(r_mean - r_std, all_rewards.min(0))]), smoothing)

    ax = ax or plt.figure(figsize=(6, 4)).gca()
    if len(all_rewards) > 1:
        ax.fill_between(times, r_min, r_max, color=color, linewidth=0, alpha=alpha)
    ax.plot(times, r_mean, color=[color[0]*0.75, color[1]*0.75, color[2]*0.75], linestyle=linestyle, linewidth=1.0, label=label)
    ax.set_yticks([0, 70, 200, 400, 600, 800, 1000])
```

```python
    ax.set_ylim((0, 1000.0))

    if ax is None:
        plt.grid(linewidth=0.4, alpha=0.5)
        plt.axhline(200, color='k', linestyle='--', linewidth=0.5)
        plt.suptitle(f'Mountain car. Statistics of {len(timelines)} agents', y=0.94)
        plt.xlabel('Episodes')
        plt.ylabel('Steps until goal')
        if os.path.exists(os.path.dirname(save_path)):
            plt.savefig(save_path)
        if show:
            plt.show()
        plt.close()
    else:
        return ax


def plot_training_free_energy(timelines: Union[Timeline, List[Timeline]], save_path=None, show=True, ax=None, label=None, color=(0.2, 0.4, 1.0), linestyle='-', smoothing=0):
    timelines = [timelines] if isinstance(timelines, Timeline) else timelines
    all_free_energies = []
    times = None
    for timeline in timelines:
        times, free_energy = timeline.select_features('cumulative_VFE')
        all_free_energies.append(free_energy)

    all_free_energies = np.array(all_free_energies)
    r_mean = smooth(all_free_energies.mean(0), smoothing)
    r_std = all_free_energies.std(0)
    r_max = smooth(np.array([min(a, b) for a, b in zip(r_mean + r_std, all_free_energies.max(0))]), smoothing)
    r_min = smooth(np.array([max(a, b) for a, b in zip(r_mean - r_std, all_free_energies.min(0))]), smoothing)

    ax = ax or plt.figure(figsize=(6, 4)).gca()
    if len(all_free_energies) > 1:
        ax.fill_between(times, r_min, r_max, color=color, linewidth=0, alpha=0.3)
    ax.plot(times, r_mean, color=[color[0]*0.75, color[1]*0.75, color[2]*0.75], linestyle=linestyle, linewidth=1.0, label=label)

    if ax is None:
        plt.grid(linewidth=0.4, alpha=0.5)
        plt.axhline(200, color='k', linestyle='--', linewidth=0.5)
        plt.suptitle(f'Mountain car. Statistics of {len(timelines)} agents', y=0.94)
        plt.xlabel('Episodes')
        plt.ylabel('Cumulative free energy')
        if os.path.exists(os.path.dirname(save_path)):
            plt.savefig(save_path)
        if show:
            plt.show()
        plt.close()
    else:
        return ax


def plot_cumulative_free_energies(timeline: Timeline):
    episodes, (cumulative_VFE, cumulative_FEEF) = timeline.select_features(['cumulative_VFE', 'cumulative_FEEF'])
    fig = plt.figure(figsize=(6, 4))
    ax = fig.gca()
    ax.plot(episodes, cumulative_VFE, 'b-', label='Cumulative FE')
    ax.plot(episodes, cumulative_FEEF, 'r--', label='Cumulative FEEF')
    ax.legend()
    ax.set_xlabel('Episode')
    ax.set_ylabel('Free energy')
    plt.show()
    plt.close()


def make_video_frame(agent: ActiveInferenceCapsule, episode_history: Timeline, render, observations_mapper):
    fig = plt.figure(figsize=(6, 5))
    gs = fig.add_gridspec(5, 2)
    ax_phase = fig.add_subplot(gs[:3, 0])
    ax_frame = fig.add_subplot(gs[:3, 1])
    ax_action = fig.add_subplot(gs[3:, :])

    _plot_phase_portrait(fig, ax_phase, agent, episode_history, observations_mapper)
    ax_phase.set_xticks([-1, -0.5, 0, 0.5, 1.0])
    ax_phase.set_yticks([-1, -0.5, 0, 0.5, 1.0])

    ax_frame.set_title('Simulation frame')
    ax_frame.imshow(render)
    ax_frame.get_xaxis().set_visible(False)
    ax_frame.get_yaxis().set_visible(False)

    ax_action = _plot_observations_actions(ax_action, agent, episode_history.merge(agent.logged_history))
    ax_action.set_xlim((0, 140))
    fig.tight_layout()
```

```python

    # plt.show()
    return fig


if __name__ == '__main__':
    import pickle

    with open('./experiments/batch_run/results.pickle', 'rb') as f:
        tmlns = pickle.load(f)
    plot_training_history(tmlns, save_path='./experiments/batch_run/run_stats.pdf', show=True)
```

B.8 Utils

utils/model_saving.py

```python
import torch
import torch.distributions

"""
Utility class to save and selectively load parts of the model
"""


def save(model, model_save_filepath):
    torch.save(model.state_dict(), model_save_filepath)


def _get_sub_model_state_dict(state_dict, sub_model_path):
    sub_model_dict = {}
    idx_next_child = len(sub_model_path)
    for key, value in state_dict.items():
        if key[:idx_next_child] == sub_model_path:
            sub_model_dict[key[idx_next_child + 1:]] = value
    return sub_model_dict


def load_capsule_parameters(model, model_save_filepath, load_vae=True, load_transition_model=True, load_prior_model=True):
    state_dict = torch.load(model_save_filepath)
    if load_vae:
        model.vae.load_state_dict(_get_sub_model_state_dict(state_dict, 'vae'))
    if load_transition_model:
        model.transition_model.load_state_dict(_get_sub_model_state_dict(state_dict, 'transition_model'))
    if load_prior_model and not isinstance(model.prior_model, torch.distributions.Distribution):
        if len([key for key in state_dict.keys() if 'prior_model.' in key]) > 0:
            model.prior_model.load_state_dict(_get_sub_model_state_dict(state_dict, 'prior_model'))
```

utils/silu.py

```python
import torch.nn as nn
import torch

"""
Implementation of the SiLU activation function for neural networks
[1] Elfwing, S., Uchibe, E., & Doya, K. (2018). Sigmoid-weighted linear units for neural network function approximation in reinforcement learning. Neural Networks, 107(2015), 3-11. https://doi.org/10.1016/j.neunet.2017.12.012
"""


class _SiLU(nn.Module):
    def forward(self, x):
        return x * torch.sigmoid(x)


# The SiLu activation (also known as Swish) was introduced in Pytorch 1.8. If not available, provide own implementation
if hasattr(torch.nn, 'SiLU'):
    SiLU = torch.nn.SiLU
else:
    SiLU = _SiLU
```

utils/timeline.py

```python
from typing import Union, Any, List
from collections.abc import Iterable
from collections import defaultdict


"""
Utility class for logging time-series data and doing some handy operations with it
"""


class Timeline:
    def __init__(self, time_max_decimals=5):
        self.tml = defaultdict(dict)
        self.time_max_decimals = time_max_decimals

    @property
    def times(self):
        return sorted(list(self.tml.keys()))

    def get_frame(self, time):
        return self.tml[time]

    def log(self, time: Union[int, float, List[Union[int, float]], Any], key: str, value: Union[Any, List[Any]]):
        """
        Add an entry for a particular time-step. Raises exception if the key already exists for that time-step.
        It is also possible to add entries in batch by providing a list of times and corresponding values.
        """
        if issubclass(type(time), Iterable):
            if issubclass(type(value), Iterable):
                assert len(time) == len(value), "Length of arrays <time> and <value> does not match"
            else:
                value = [value for _ in time]
            for i in range(len(time)):
                self.log(float(time[i]), key, value[i])
        else:
            time = round(float(time), self.time_max_decimals)
            if key in self.tml[time].keys():
                raise KeyError(f'key <{key}> already exists at t={time}')
            else:
                self.tml[time][key] = value

    def select_features(self, keys: Union[str, List[str]]):
        """
        Returns all times and values for the given keys
        """
        keys = [keys] if isinstance(keys, str) else keys
        times = []
        values = [[] for _ in keys]
        for time in sorted(self.times):  # make sure the return time-series is sorted (increasing time)
            if not all([key in self.tml[time].keys() for key in keys]):
                continue    # Only include time-step if all keys have a datapoint in it
            times.append(time)
            for i, key in enumerate(keys):
                values[i].append(self.tml[time][key])
        if len(values) == 1:
            return times, values[0]
        else:
            return times, values

    def delete_feature(self, key):
        for time in self.times:
            if key in list(self.tml[time].keys()):
                del self.tml[time][key]

    def merge(self, other: 'Timeline'):
        new_times = sorted(list(set(self.times + other.times)))
        self_dt = self.times[-1] - self.times[-2]
        other_dt = other.times[-1] - other.times[-2]
        new_dt = new_times[-1] - new_times[-2]
        self_resampled = self.resample(new_times, extend_steps_ends=int(round(self_dt / new_dt, self.time_max_decimals)) - 1)
        other_resampled = other.resample(new_times, extend_steps_ends=int(round(other_dt / new_dt, self.time_max_decimals)) - 1)
        merged = Timeline()
        for t in new_times:
            for key, value in self_resampled.tml[t].items():
                merged.log(t, key, value)
            for key, value in other_resampled.tml[t].items():
                merged.log(t, key, value)
        return merged

    def resample(self, new_times, extend_steps_ends=0, keep_outside_times=True):
```

```
        """
        Returns a new Timeline object sampled at the new times.
        The values are selected from the closest available logged time-step.
        If the logs contain Timeline objects as values, these will be expanded recursively.
        If keep_outside_times==True (default), the times outside the range of new_times will be kept.
        """
        new_tml = Timeline()
        new_delta_t = 0 if not extend_steps_ends else new_times[-1] - new_times[-2]
        if extend_steps_ends:
            new_times = list(new_times) + [new_times[-1] + i * new_delta_t for i in range(1, extend_steps_ends + 1)]
        # Resample non-timeline objects
        resampled_times = []
        for new_t in new_times:
            try:
                closest_t = self.times[max([i for i, t in enumerate(self.times) if t <= new_t])]  # Largest available time lower than new_t
                resampled_times.append(closest_t)
            except ValueError:
                continue  # No time-step found before new_t, skip
            for key, value in self.tml[closest_t].items():
                if not isinstance(value, Timeline):
                    new_tml.log(new_t, key, value)
        # Include times outside the range of new_times
        if keep_outside_times:
            outside_times = list(set(self.times) - set(resampled_times))
            for time in outside_times:
                for key, value in self.tml[time].items():
                    if not isinstance(value, Timeline):
                        new_tml.log(time, key, value)
        # Recursively resample timeline objects
        for current_t in self.times:
            for key, value in self.tml[current_t].items():
                if isinstance(value, Timeline):
                    start_t = min(value.times)
                    end_t = max(value.times) + extend_steps_ends * new_delta_t
                    resampled = value.resample([t for t in new_times if start_t <= t <= end_t])
                    closest_t = new_times[min([i for i, t in enumerate(new_times) if t >= current_t])]  # Smallest new time larger than current_t
                    new_tml.log(closest_t, key, resampled)
        return new_tml


if __name__ == '__main__':
    import torch
    import numpy as np

    tml = Timeline()
    tml.log(np.arange(0.0, 0.4, 0.1), 'torch', torch.randn(4))
    tml.log(0.1, 'val', 0.1)
    tml.log(0.2, 'val', 0.4)
    tml_sub = Timeline()
    tml_sub.log([0.2, 0.3], 'val2', [0.1, 0.5])
    tml.log(0.2, 'sub', tml_sub)
    tml.log(0.3, 'val', 0.2)

    tml_other = Timeline()
    tml_other.log(np.arange(0.0, 0.4, 0.025), 'torch2', torch.randn(16))
    tml_merged = tml.merge(tml_other)

    tml2 = tml.resample(np.arange(0.0, 0.3, 0.05), extend_steps_ends=2)
    times_, values_ = tml2.select_feature('sub')
    print(times_)
    print(values_)
```

utils/value_map.py

```
import torch.nn as nn


"""
Maps a value from a range onto another
"""


class ValueMap(nn.Module):
    def __init__(self, in_min, in_max, out_min, out_max):
        super(ValueMap, self).__init__()
        self.in_min = in_min
        self.in_max = in_max
        self.out_min = out_min
```

```python
        self.out_max = out_max
        self.in_width = in_max - in_min
        self.out_width = out_max - out_min

    def forward(self, value):
        return (value - self.in_min) / self.in_width * self.out_width + self.out_min

    def inverse(self, value):
        return (value - self.out_min) / self.out_width * self.in_width + self.in_min
```