DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics: Robotics, Cognition, Intelligence

Generalized Adaptive Skill Prior Meta-Reinforcement Learning

Mikhail Eibozhenko

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics: Robotics, Cognition, Intelligence

Generalized Adaptive Skill Prior Meta-Reinforcement Learning

Author:	Mikhail Eibozhenko
Supervisor:	Prof. Dr.-Ing. Alois Knoll
Advisor:	Xiangtong Yao
Submission Date:	01.08.2024

I confirm that this master's thesis in informatics: robotics, cognition, intelligence is my own work and I have documented all sources and material used.

Munich, 01.08.2024 Mikhail Eibozhenko

Acknowledgments

I wish to express my sincere gratitude to my advisor Xiangtong Yao for his expert guidance throughout this thesis and to supervisor Prof. Dr.-Ing Alois Knoll whose support in allowing me to explore this topic and providing all necessary resources for its development and evaluation was invaluable. Additionally, I am deeply grateful to my family and friends, whose unwavering belief in me and continuous support sustained me throughout this fascinating journey.

Abstract

Modern Artificial Intelligence approaches in robotics aim for efficient task execution and adaptability in task perception, yet often lack flexible high-level policies. These methods either accelerate learning, improve task accuracy, or focus on Meta-Learning, leading to highly abstract models that are less adapted for real-world tasks, relying heavily on handcrafted action primitives or hyperparameter initialization. This thesis introduces the **Generalized Adaptive Skill Prior Meta-Reinforcement Learning (GASP Meta-RL)** framework, a novel approach for nonparametric Bayesian clustering in Skill Embedding spaces, supporting entropy-based Hierarchical Reinforcement Learning. We propose a comprehensive framework, combining state-of-the-art algorithms to unite Deep Learning, Reinforcement Learning, and Clustering techniques under one powerful solution. Our model, trained on unstructured and unlabeled complex robotic manipulation tasks, demonstrates competitive performance by autonomously inferring latent patterns and deriving a flexible, adaptive high-level policy, comparable to human subconscious decision-making in manipulation actions.

Keywords: Dirichlet Process Mixture Model, Hierarchical Reinforcement Learning, Meta-Reinforcement Learning, Robotics, Skills, Variational Autoencoder

Contents

1. Introduction

In the end of the 18th century, the industrial revolution opened new horizons for the automation processes for humanity [1][2]. Technological growth and mass demand brought conveyors and electricity-powered engines into existence over the span of one and a half centuries [3], followed by digitalization in the modern age [4]. Homo Sapiens are driven by progress and scientific curiosity, which always pushes them to conquer new milestones in achieving a full understanding and control over available resources [5].

Recent breakthroughs in pipelines automated by robots and digitalization related to the 4th Industrial Revolution [4] serve as the basis for research topics and philosophical thoughts about the 5th Industrial Revolution [6], given the active development and study of the Artificial Intelligence (AI) field [7]. It is tempting to imagine a world where robots and humans collaboratively solve tasks, and AI-guided manipulators help with human routines.

In recent years, many papers have been dedicated to addressing various problems that would arise from realizing these ambitions, presenting their vision and proposed solutions [8][9][10][11][12]. Each of them concentrates on a different aspect of the problem. For example, how to teach a machine to recognize actions performed by humans? Proposed in [8] method delves into semantic decomposition of complex action sequences demonstrated by humans to extract so-called "Semantic Event Chains" and recognize patterns. Another branch researches the algorithms to give certain rationality to the agent's behavior - thus giving rise to the study of Reinforcement Learning (RL) [9], where agent learns by interacting with the environment. Trying to copy human demonstrations gave birth to the field of Imitation Learning [10], with the main task of reconstructing the reward function with Inverse Reinforcement Learning (IRL) techniques [13] or mimicking the demonstrated behavior with Behavioral Cloning (BC) variations [14]. Unsupervised clustering algorithms are focused on machine perception and clustering of unstructured data [15] to ease the understanding of surrounding world and proceed to a new level of abstraction.

Achieving the optimal synergy between these studies is not an easy task. Restricting the vision of the problem to manipulators performing human-like tasks, one should mention the fundamental work of S. Thrun and A. Schwartz [16], proposing the structural solution for RL in real-world task scenarios and introducing "Skill" term as a partially defined action policy that occurs in more than one task. Learning such Skills may potentially decompose every long complex manipulation sequence into fundamental action blocks to be performed sequentially. Finding patterns to transfer between seemingly different complex tasks gave rise to Meta-Learning approaches [17]. The scientific world of Artificial Intelligence thrives with various algorithms allowing the creation of advanced agents which will adaptively and autonomously learn latent patterns for their efficient application on routine tasks.

However, as it tends to be, with more specifications of demands, fewer practical solutions

can be found. While interest in Industry 5.0 AI-driven robotics continues to rise [6], there are not yet enough working end-to-end models capable of perceiving the latent structure of the task rather than memorizing patterns. The main scope of this work is to make a contribution to the development of end-to-end intelligent robotic architectures by introducing the **Generalized Adaptive Skill Prior (GASP) Meta-Reinforcement Learning** framework based on state-of-the-art Hierarchical Skill-Prior Reinforcement Learning (SPiRL) [18] and a Dirichlet Process Based Incremental Deep Clustering Algorithm via Variational Auto-Encoder (DIVA) [19] models.

To accomplish this, a broad range of disciplines should be considered. This work is organized into separate blocks leading to a comprehensive understanding of Reinforcement Learning in robotics with deep classification of latent properties of executed tasks.

Chapter 2 (Background) is specifically designed as an overview of all related fields starting from foundations to state-of-the-art algorithms involved in the realization of this work. In section 2.1 we will focus on Deep Learning architectures, beginning from fundamental concepts and concluding with important computing blocks for our method. Section 2.2 decomposes the main Reinforcement Learning algorithms and describes the necessary theory for further understanding of Meta-Reinforcement Learning and Hierarchical Reinforcement Learning terms used in our work. Finally, in section 2.3 we analyse the concept of clustering, analytically break down clustering approaches and apply this understanding to our work. This chapter contains primarily theoretical information about the algorithms our solution is based on and structurally serves as a bottom-up guideline for comprehensive understanding and analysis of related approaches. In further chapters, we will prioritize clarification of concepts and consistently refer to the foundations discussed here.

Chapter 3 (Related work) presents and decomposes two main frameworks addressed throughout our work. We will discuss the approaches of K. Pertsch et al. [18] in accelerating Reinforcement Learning by introducing Skill Prior in Hierarchical Reinforcement Learning as well as Dirichlet Process-based Latent space classification proposed by Z. Bing et al. [19].

In chapter 4 (Methodology) we expand our solution for Generalized Adaptive Skill Prior Meta-Reinforcement learning by summarizing all of the above into one model capable of simultaneous Skill Embedding extraction and Skill clustering with nonparametric Bayesian deep model for further Hierarchical Reinforcement Learning.

In chapter 5 (Results) we show that our approach is at least as good as proposed in [18] but contains a deeper understanding of the meta-structure of the given tasks. We conduct several experiments to check the plausibility of our model and show its flexibility. At the end we leave some comments about the possible influence on domains our approach relates to and discuss further work.

Finally, we will conclude our work with a comprehensive summary recalling our motivation, underlying concepts, our solution and the results we obtained during this work.

2. Background

We will begin our work with a comprehensive analysis of theoretical studies on which our solution is built. In order to comprehend the concept behind Skill encryption and Skills learning, we must trace back to the ideas of Artificial Neural Network (ANN), discuss the default modules for data processing with respect to the given problem. The Skill generalization should serve as a generative model for later Skill library formulation; thus, it is important to study generative models such as the Variational Autoencoder (VAE). These frameworks will be described in detail in section 2.1. Then we will analyze the underlying theory of Reinforcement Learning as a decision making framework component in section 2.2. We will begin with analyzing the basic concepts of Reinforcement Learning, then we will go over more advanced algorithms, and finally present Hierarchical Reinforcement Learning, High-level policy Skill-based approaches, and describe how our algorithm inherits Meta-Reinforcement Learning properties. Finally, we will present selected unsupervised classification algorithms in section 2.3. We will analyze the basics behind clustering as well as discuss the most promising algorithms that help to build up latent space clustering.

The purpose of this chapter is twofold. Firstly, we will analyze scientific approaches related to our problem and build a solid theoretical foundation for the algorithms involved in our solution, and secondly, we will accurately decompose the algorithms used in our approach for future references and collect the most important mathematical proofs to keep our solution report clean and readable.

2.1. Deep Learning Architectures

Deep Learning is a subpart of the Machine Learning field focused on Artificial Neural Networks (ANN). Originally, the term "neural network" was inspired by attempts at the mathematical representation of the function of the biological neuron in 1943 [20]. In 1958, the first multilayer perceptron with randomized non-trainable weights was introduced by Frank Rosenblatt [21].

Today's ANNs impact on modern technology is hard to overestimate. In the last half century deep learning architectures have grown in size, new approaches have been introduced over time [22] and today various methods are applied in many areas - from Computer Vision [23] to Autonomous Driving [24].

In this chapter we will analyze the underlying concepts of Deep Learning as well as discuss their impact on robotics applications, particularly concentrating on their utility for our work. In the first section of this chapter 2.1.1 we will discuss the basics of Artificial Neural Networks and examine the generative approach to latent space modeling for encoding Skills. The

second part 2.1.3 will focus on deep sequential processing where the ANN architectures capable of handling sequential data will be presented and analyzed.

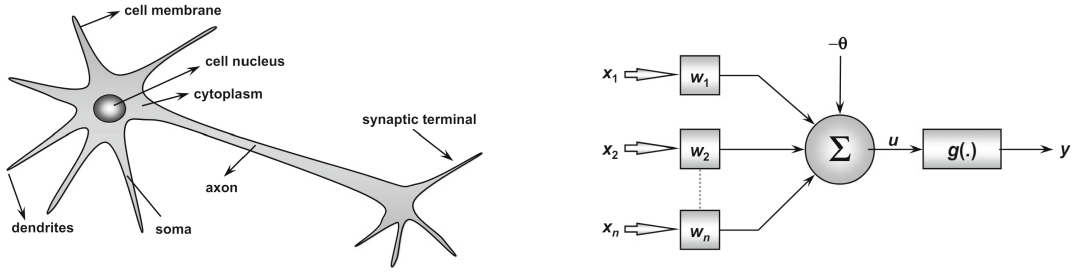
2.1.1. Foundations of Deep Learning

Inspired by biological neurons, the first artificial neuron architectures were bounded by unnecessary naturally occurring constraints [15]. Subsequent mathematical models however not only bypassed these shortcomings, but have also been proven to be the universal approximators [25], securing the interest of researchers up until this day.

The mathematical model of a single neuron proposed by Rosenblatt [21] called a perceptron played an important role in the history of ANN formation, but originally relates to the classification algorithms [15]. From the perspective of our work ANNs are approximators with learnable parameters. We will concentrate on a few models that are mainly used for similar applications [18][19] and begin with a simple multilayer network.

Multilayer Feedforward Network

We will begin with the definition of the artificial neuron. As stated above, its architecture was inspired by the biological neuron (see Figure 2.1).



(a) Biological Neuron. (b) Artificial Neuron.

Figure 2.1.: Biology-inspired mathematical model of artificial neuron [26].

Input vector $\mathbf{x}$ is multiplied with weights $\mathbf{w}$, and bias $-\theta$ (often b notation is used) is added to the weighted sum, and the result u is passed through a non-linear activation function $g(u)$ (often denoted as $a(u)$) [26]. Altogether, this is called a *forward pass* and is denoted as

$$y(\mathbf{x}, \mathbf{w}) = a(\mathbf{x}^T\mathbf{w} + b) \tag{2.1}$$

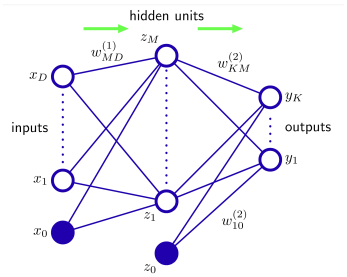
The combination of multiple neurons in one layer forms a single-layer Feed-forward neural network. The combination of multiple layers forms a multilayer Feed-forward neural network [26]. The usage of Feed-forward Neural Network (FFNN) framework is overwhelmingly popular in most modern robotic applications [26][18][19][27][28]. It is a basic example suitable for discussing and analyzing general features applicable to most ANNs:

- **Architecture** of ANN can be described figuratively as a black-box: *inputs* $\boldsymbol{x}$ enter the network and *outputs* $\boldsymbol{y}$ are the result of network processing [15]. Different ANNs have different architectures [29]. We already mentioned that the FFNN architecture consists of several layers of neurons. The first layer is called the *input layer*, and the last one is the *output layer*. All layers in between are called *hidden layers* [15] (see Figure 2.2 (a)). The architecture of a neural network is highly dependent on the goal (for example, for linear regression of a one-dimensional function, one neuron per input/output layer suffices [15]) and on the complexity of the approximation (deep ANNs with a large number of hidden layers can potentially learn more complex functions [29]).

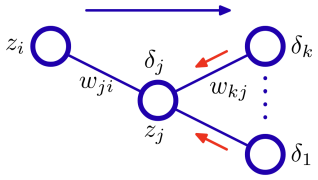
- **Activation functions** are functions that add non-linearity to the forward pass [15]. Each neuron may have an individual activation function (see Figure 2.1 (b)), but in practice activation functions are shared across a layer [15]. Again, activation functions are highly dependent on the task, but always introduce non-linearity to avoid the redundancy of a linear combination [29]. Common choices for activation functions among others are sigmoid $\sigma(z) = \frac{1}{1+e^{-z}}$, ReLU $a(z) = \max(0, z)$ and hyperbolic tangent $a(z) = \tanh(z)$[15]. Given parametric weights, inputs and activation functions, our FFNN can perform a *forward pass* to compute the predicted output.

- **Loss functions** determine if the network has succeeded in its prediction and enable training of the weights [15][29]. Their main function is to estimate the error based on a chosen metric and *backpropagate* it through the network, adjusting weights for better approximation. Of course, the loss function depends on the goal of the network [15]. For regression problems Mean Square Error (MSE) $E(\boldsymbol{w}) = \frac{1}{2}\sum_{n=1}^{N} ||\hat{\boldsymbol{y}}(\boldsymbol{x}_n, \boldsymbol{w}) - \boldsymbol{y}||$ is often chosen, where the symbols $\hat{\boldsymbol{y}}$ denote network predictions and $\boldsymbol{y}$ denotes true labels. For supervised clustering, Binary Cross Entropy (BCE) $E(\boldsymbol{w}) = -\sum_{n=1}^{N}\sum_{k=1}^{K} y_{kn} \ln \hat{y}_k(\boldsymbol{x}_n, \boldsymbol{w})$ for $t_k \in \{0, 1\}$ is an applicable choice [15]. The topic of loss functions and their optimization is extensive, and for educational purposes, we will mention the example of Stochastic Gradient Descent optimization (SGD) [29] where $\partial E(\boldsymbol{w})/\partial \boldsymbol{w}$ is computed on small batches of data called *mini-batches*. Based on the chain rule we can update the weights of our network, a process called the *backpropagation* (see Figure 2.2 (b)). With this, our network can predict and learn in an iterative manner [15].

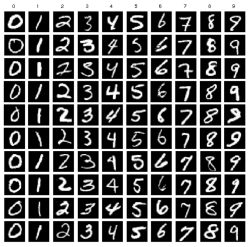
- **Additional Enhancements** are often used for stabilization and evaluation of the learning process. For example, random *weight initialization* may result in exploding or vanishing gradient during the optimization step, therefore it is important to keep the variance of activations the same in each layer of the ANN [30]. *Dataset* is also worthy of attention - it is common practice to divide it into training data, validation data and test data for research purposes [31] (see Figure 2.2 (c)). Another example is *data augmentation*, which is particularly powerful in image processing neural networks, where additional data pre-processing may enhance the quality of the model [32]. Other techniques such as *pruning*, *kernels* and *regularization techniques* [29] can also improve the accuracy of the model.



(a) Feed-forward Neural Network Architecture [15].



(b) Backpropagation example [15].



(c) MNIST Data set snippet as presented in [33].

Figure 2.2.: (a) Main building blocks of ANNs are neurons combined in layers. (b) Forward pass (blue arrow) computes the prediction of an ANN, while the chain rule derivation of the loss function propagates through the network and adjusts the weights accordingly (red arrows). (c) Datasets vary by their nature. For example, the MNIST dataset consists of input images x_n and their labels y_n.

Artificial Neural Networks are a dynamic and extensive field of study which is impossible to sufficiently summarize in a few lines without losing information. Therefore, we have presented the intuition behind the generalized process using an example of a Feed-forward neural network. As we proceed with more specific examples, we will be more granular about the underlying processes.

2.1.2. Autoencoders

Dimensionality reduction is one of the topics of the data science dedicated to the most efficient (with minimal loss of information) low-dimensional representation of high-dimensional data [34]. The motivation is intuitive - why store a large amount of information that requires detailed analysis if it can be described efficiently, reducing weight and computational cost for later usage? Conventional algorithms include for example linear dimensional reduction methods such as Principle Component Analysis (PCA) [35] and more efficient due to its non-linearity t-SNE [36]. However, recalling that multilayer feedforward networks are universal approximators [25] raises the question of whether FFNN can be used for dimensionality reduction.

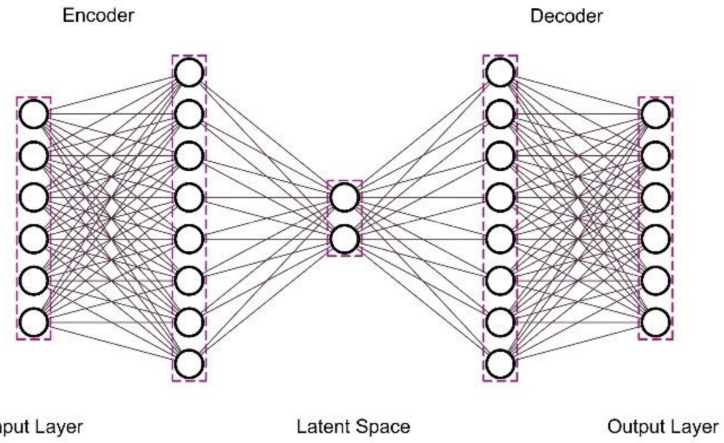
Indeed, Autoencoder is designed to process unlabeled data to compress it into latent representation and then reconstruct the initial pattern [14][37]. The only architectural restriction that needs to be made is ensuring the hidden layer in the latent space is narrower than the input and output layers, which are required to be the same size [29].

As seen in Figure 2.3, the Autoencoder has a deep neural architecture shaped like a sandglass. The goal of the Autoencoder is to pass the data through the bottleneck and reconstruct it with minimal loss at the output [14]. Since the data is unlabeled, the Autoencoder learns in an unsupervised manner [29] with the reconstruction loss often being MSE between recreated

$\hat{\boldsymbol{y}}$ and the original data $\boldsymbol{x}$:

$$E(\boldsymbol{w}) = \frac{1}{N}\sum_{n=1}^{N} ||\hat{\boldsymbol{y}}(\boldsymbol{x}_n, \boldsymbol{w}) - \boldsymbol{x}|| \tag{2.2}$$

Since the low-dimensional representation is encoded in the bottleneck, this layer is called the *latent space*. The layers from input to latent space attempt to encode the information and those from latent space to output attempt to decode to decode it; therefore these substructures are defined as *encoder* and *decoder* respectively [38].



[H]

Figure 2.3.: Graphical representation of Autoencoder architecture [38].

Unfortunately, the Autoencoder is nothing more than a deep architecture for non-linear dimensionality reduction [37]. Moreover, a one-layer Autoencoder is indeed equivalent to PCA [29]. The latent space is not human interpretable and serves only as information compression.

In 2008, Vincent presented a Denoising Autoencoder [39] that was trained to reconstruct the original input from a corrupted one. Nevertheless, the true potential of the Autoencoder was discovered in 2013 with the paper "Auto-encoding variational bayes" [40] where authors proposed restricting the latent space to a probabilistic distribution, turning the model into a generative one.

Variational Autoencoders

The idea of [40] is simple - instead of projecting input x onto a fixed latent vector, it can be projected onto a latent distribution p_θ parameterized by θ. Thus, the relation between the latent dimension $\boldsymbol{z}$ and input $\boldsymbol{x}$ can be described by Bayes' theorem

$$p(\boldsymbol{z}|\boldsymbol{x}) = \frac{p(\boldsymbol{x}|\boldsymbol{z})p_\theta(\boldsymbol{z})}{\int_z p(\boldsymbol{x}|\boldsymbol{z})p_\theta(\boldsymbol{z})d\boldsymbol{z}} \tag{2.3}$$

In this case $p(\boldsymbol{z}|\boldsymbol{x})$ is the posterior distribution, $p(\boldsymbol{x}|\boldsymbol{z})$ is the likelihood and $p_\theta(\boldsymbol{z})$ is the prior parameterized by θ [29]. What advantages does this probabilistic model offer? Firstly,

the latent space is now distributed according to a known probability $z \sim p_\theta(z)$ making it is human interpretable. Secondly, by sampling z_i from the prior and applying the learned likelihood $p(x|z_i)$ we can generate new $\hat{x}$ that is hopefully similar to the original data x [14]. Therefore $p(z|x)$ is actually a probabilistic version of the encoder and $p(x|z)$ represents the decoder of a VAE [41].

But how exactly can we train such probabilistic model? The computation of the posterior $p(z|x)$ requires an estimation of $p_\theta(x) = \int_z p(x|z)p_\theta(z)dz$ which is computationally expensive [41]. Therefore, we might introduce a learnable posterior $q_\phi(z|x)$ parameterized by the learnable parameter ϕ and ensure with the inverse Kullback–Leibler divergence that $q_\phi(z|x)$ is close to $p_\theta(z|x)$ [41].

Kullback–Leibler (KL) divergence is a powerful tool from statistics [42] to measure how one probability distribution differs from a second distribution defined as

$$\mathbb{KL}(q(z)||p(z)) = \int q(z) \log \frac{q(z)}{p(z)} dz \tag{2.4}$$

Notably, KL divergence is not symmetric [42]. Figure 2.4 illustrates the mean-seeking and mode-seeking behaviors of KL divergence formulations.

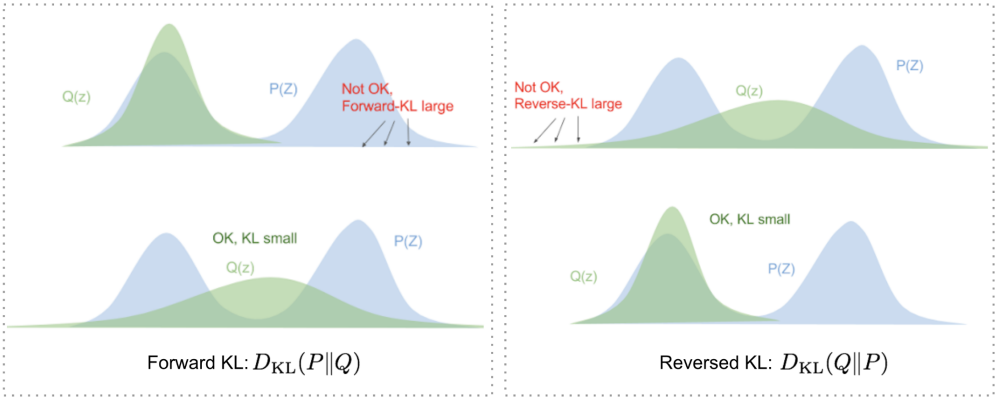


Figure 2.4.: Forward and Inverse KL divergence measure the distribution similarities differently - Forward KL(P||Q) forces Q(z) to cover the entire P(z) (mean-seeking) while reverse KL (Q||P) fits Q(z) under P(z) (mode-seeking) [41].

Equipped with this knowledge about KL divergence, we recall that our goal is to minimize $\mathbb{KL}(q_\phi(z|x)||p_\theta(z|x))$. The result can be written as [41]

$$\log p_\theta(x) - \mathbb{KL}(q_\phi(z|x)||p_\theta(z|x)) = \mathbb{E}_{z\sim q_\phi(z|x)} \log p_\theta(z|x) - \mathbb{KL}(q_\phi(z|x)||p_\theta(z)) \tag{2.5}$$

Before formulating the resulting loss function, recall that sampling is stochastic and cannot backpropagate the error. The trick here is called *reparametrization* and involves finally determining the probabilities we are working with. A common choice for VAE prior $p_\theta(z)$ and decoder $q_\phi(z|x)$ is the Gaussian distribution [40][18]. This further specifies the KL divergence [14] and allows sampling $z = \mu + \sigma \odot \epsilon$ from Gaussian noise $\epsilon \sim \mathcal{N}(0, I)$ [41].

For the VAE loss function, the probabilistic problem is converted into an optimization problem via maximization of the Evidence Lower Bound (ELBO), so named because the loss function is always a lower bound of $\log p_\theta(\boldsymbol{x})$ [41]:

$$ELBO(\boldsymbol{\theta}, \boldsymbol{\phi}) = \underbrace{\mathbb{E}_{z\sim q_\phi(z|\boldsymbol{x})} \log p_\theta(\boldsymbol{z}|\boldsymbol{x})}_{\text{Reconstruction Error}} - \underbrace{\mathbb{KL}(q_\phi(\boldsymbol{z}|\boldsymbol{x})||p_\theta(\boldsymbol{z}))}_{\text{Deviation from prior}} \leq \log p_\theta(\boldsymbol{x}) \tag{2.6}$$

As mentioned above, for Gaussian distributed prior and posterior equation (2.6) can be rewritten as

$$ELBO(\boldsymbol{\theta}, \boldsymbol{\phi}) = \underbrace{\mathbb{E}_{z\sim q_\phi(z|\boldsymbol{x})} \log p_\theta(\boldsymbol{z}|\boldsymbol{x})}_{\text{Reconstruction Error}} + \underbrace{\frac{1}{2}\sum_{d=1}^{D}(1 + \log(\sigma_{\phi_d}^2) - \mu_{\phi_d}^2 - \sigma_{\phi_d}^2)}_{\text{Deviation from Gaussian distribution}} \tag{2.7}$$

for a D- dimensional multivariate Gaussisan distribution [40]. Due to the reparametrization trick the derivative $\nabla ELBO(\boldsymbol{\theta}, \boldsymbol{\phi})$ can be computed for the usual optimization of $\boldsymbol{\theta}*, \boldsymbol{\phi}* = \arg\max_{\theta,\phi} ELBO(\boldsymbol{\theta}, \boldsymbol{\phi})$ [40].

To summarize, Variational Autoencoders introduced a probabilistic approach to basic Autoencoders, enforcing the latent space to follow a distribution. This simple augmentation introduced an ELBO formulation and redefined the shape of the encoder and decoder to output the parameters of a distribution rather than the latent space vector itself (see Figure 2.5).

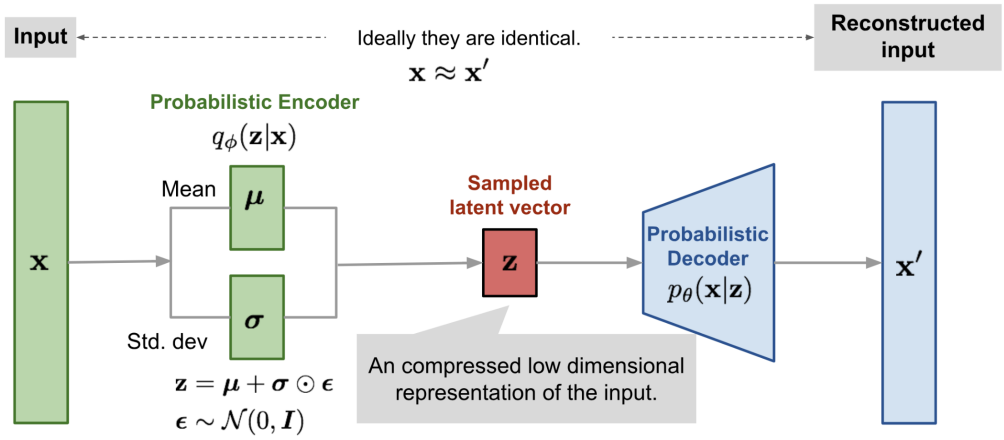


Figure 2.5.: The schematic drawing of VAE architecture [41].

Gaussian Mixture Variational Autoencoder

While constraining latent space to be normally distributed provides a powerful tool for many robotic applications [43][44], the Evidence Lower Bound equation (2.6) allows for the processing of arbitrary probabilistic distribution. In section 2.3 we will discuss Gaussian

Mixtures, which allow the combination of several Gaussian distributed components into one probabilistic model.

The idea of combining a Gaussian Mixture with a VAE dates back at least to the work of Nat Dilokthanakul et al. [45]. This algorithm, called Gaussian Mixture Variational Autoencoder (GMVAE), simultaneously decomposes latent space into several components using Gaussian Mixture Model clustering and optimizes the Evidence Lower Bound for a Variational autoencoder.

In modern approaches GMVAE has became a useful tool when latent distribution can be decomposed into several clusters for further efficiency [18][19]. We will discuss Gaussian Mixture Model clustering in detail in section 2.3.2 and refer to this approach many times as we replace Gaussian Mixtures with nonparametric Bayesian method in Variational Autoencoder Skill encryption.

2.1.3. Recurrent Neural Networks

Up to this point, we have considered a vector-like D-dimensional input, each instance of which is independent of the others. But what if there is a time dependency? Machine sequence recognition is a wide topic, including manipulation sequence recognition [8], weather prediction or speech recognition [15]. Sequential data differs by the fact that they do not adhere to the independence assumption, meaning that consecutive instances carry additional information encoded in their sequence [15].

The simple FFNN discussed in the last chapter cannot capture this information. In 1990 Elman proposed an Artificial Neural Network with memory [46]. This type of network uses a delay in computations by recurrently attaching layers to themselves, enabling the processing of sequence input [29].

The simplest Recurrent Neural Networks (RNN) copy the output of previous layers and feed it back at the current step of computation, which does not affect the backpropagation because mathematically, such recurrence is similar to adding another layer to the architecture [14]. The 3-layer network proposed by [46] delays hidden layer estimation, but nothing prevents from creating other architectures as shown in Figure 2.6. The resulting outputs of hidden layers are then computed as follows:

$$\begin{aligned} \boldsymbol{h}_t &= \sigma_h(\boldsymbol{W}_h \boldsymbol{x}_t + \boldsymbol{U}_h \boldsymbol{h}_{t-1} + \boldsymbol{b}_h) \\ \boldsymbol{y}_t &= \sigma_y(\boldsymbol{W}_y \boldsymbol{h}_t + \boldsymbol{b}_y) \end{aligned} \tag{2.8}$$

where $\boldsymbol{h}_t$ is the output of the hidden layer at timestep t, $\boldsymbol{W}_h$ are weights, $\boldsymbol{b}_h$ are biases of this layer, and $\boldsymbol{U}_h$ are weights of the delayed connection.

Despite maintaining the universal property [25], simple recurrent networks take a long time to train and fail to capture long-term dependencies because of vanishing gradients over time, as was empirically proven in 1997 by Hochreiter and Schmidhuber [47]. Their work proposed a new solution which became a fundamental approach for sequence recognition - Long-Short Term Memory Cells.

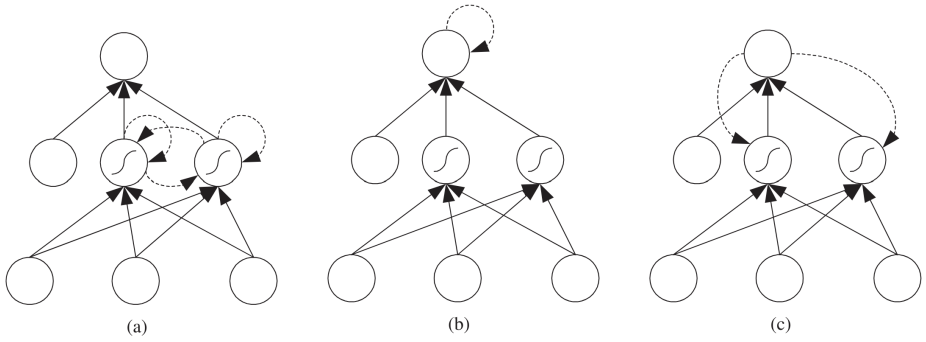


Figure 2.6.: The simpliest Recurrent Networks with (a) a hidden layer connected with a delay to itself, as in [46], (b) an output layer connected with delay to itself, (c) an output layer connected with delay to the hidden layer [29].

Long Short-Term Memory Cell

The Long Short-Term Memory (LSTM) was specifically designed to avoid the vanishing gradient problem of the original Recurrent Neural Network that occurs due to temporal differences [47]. LSTMs are a type of Recurrent Neural Network with additional parallel layers, allowing forgetting and updating of the memory during the training [48].

The main idea behind LSTM is to introduce the cell state $\boldsymbol{c}_t$ - a memory unit with which previous states and inputs can interact - and replace the recurrent hidden layer with four special layers called gates [47]. A graphical representation of an LSTM is shown in Figure 2.7.

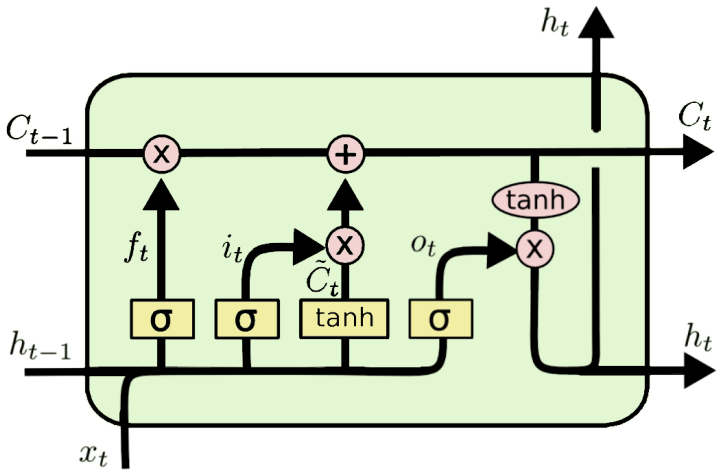


Figure 2.7.: The architecture of an LSTM [48].

To understand the concepts of LSTM, let us analyze each component in detail:

- **Input vector** $\boldsymbol{x}_t \in \mathbb{R}^D$ is a D-dimensional input instance of a sequence at time step t.
- **Output vector of the LSTM** $\boldsymbol{h}_t \in (-1,1)^h$ is a h-dimensional output of the LSTM cell at time step t.
- **Cell state vector** $\boldsymbol{c}_t \in \mathbb{R}^h$ is the current state of the LSTM and can be seen as an information conveyor [48]. Different gates manipulate the state vector in an intuitive manner.
- **Forget gate's activation vector** $\boldsymbol{f}_t \in (0,1)^h$ results from passing a concatenated input $\boldsymbol{x}_t$ and $\boldsymbol{h}_{t-1}$ through a layer called the *forget gate* with sigmoid activation function [47]. The result is multiplied with the cell state vector, intuitively regularizing whether previous information should be erased or kept [48].
- **Input gate's activation vector** $\boldsymbol{i}_t \in (0,1)^h$ controls which inputs should be updated after passing a concatenated input $\boldsymbol{x}_t$ and $\boldsymbol{h}_{t-1}$ through a layer called the *input gate* with a sigmoid activation function [48].
- **Cell input activation vector** $\tilde{\boldsymbol{c}}_t \in (-1,1)^h$ results from passing a concatenated input $\boldsymbol{x}_t$ and $\boldsymbol{h}_{t-1}$ through a layer called the *cell input layer* with a tanh activation function. Multiplied by $\boldsymbol{i}_t$, this corresponds to the actual information that will update the current cell state $\boldsymbol{c}_t$ [48].
- **Output's gate activation vector** $\boldsymbol{o}_t \in (0,1)^h$ results from passing a concatenated input $\boldsymbol{x}_t$ and $\boldsymbol{h}_{t-1}$ through a layer called the *output* with a sigmoid activation function. Essentially, the corresponding gate controls the information flowing from the memory $\tanh(\boldsymbol{c}_t)$ into the output vector $\boldsymbol{h}_t$ [48].

As a result, the overall process can be written as follows:

$$\begin{aligned}
\boldsymbol{f}_t &= \sigma(\boldsymbol{W}_f[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_f) \\
\boldsymbol{i}_t &= \sigma(\boldsymbol{W}_i[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_i) \\
\boldsymbol{o}_t &= \sigma(\boldsymbol{W}_o[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_o) \\
\tilde{\boldsymbol{c}}_t &= \tanh(\boldsymbol{W}_c[\boldsymbol{h}_{t-1}, \boldsymbol{x}_t] + \boldsymbol{b}_c) \\
\boldsymbol{c}_t &= \boldsymbol{f}_t \odot \boldsymbol{c}_{t-1} + \boldsymbol{i}_t \odot \tilde{\boldsymbol{c}}_t \\
\boldsymbol{h}_t &= \boldsymbol{o}_t \odot \tanh(\boldsymbol{c}_t)
\end{aligned} \tag{2.9}$$

where $\boldsymbol{W}_k$ and $\boldsymbol{b}_k$ are respective weights and biases and componentwise multiplication is executed [48].

LSTMs have wide application in many sequence processing tasks and many variations and extensions have followed [48]. In our work, we will utilize LSTM blocks for learning unlabeled robotic manipulation sequences and later encoding them into the latent space.

2.2. Reinforcement Learning

Reinforcement Learning (RL) as a field emerged several decades ago [9]; however, it has now become a strong foundation for modern artificial intelligence solutions, partially due to the ever-increasing computing power and accessibility of virtual simulations, which significantly simplify the training process [49]. But what exactly is Reinforcement Learning?

Any RL algorithm can be abstractly reduced to one underlying concept - the agent interacts with the environment and receives observations from it, based on which the agent performs an action [9] (see Figure 2.8). The variety of action policies, interactions, and agent definitions has given birth to a broad spectrum of algorithms in this field. To gather important information for our task, we will closely analyze the underlying concepts of RL and go from the basic definition to Meta-Reinforcement Learning.

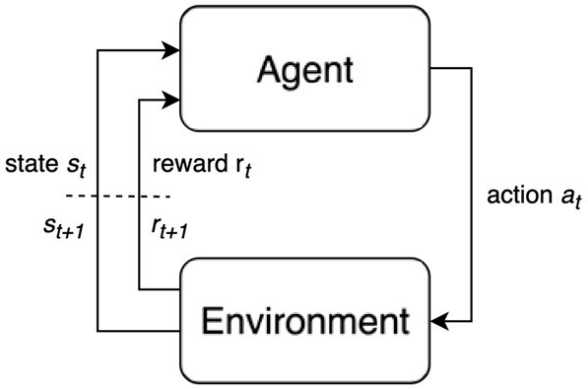


Figure 2.8.: Block diagram of the Reinforcement Learning concept as presented in [49].

2.2.1. Markov Decision Process

Let us define a discrete stochastic control problem as a tuple $\{S, A, P(.), R(.), \}$, where

- $S = \{s_0, s_1, \ldots, s_t, \ldots\}$ is the set of states.
- $A = \{a_0, a_1, \ldots, a_t, \ldots\}$ is the set of actions.
- $P(s_{t+1}|s_t, a_t)$ is the state transition function of the environment.
- $R(s_t, a_t, s_{t+1})$ is the immediate (or expected) reward function of the environment.

Then the probabilistic sequential decision-making process can be formulated as

$$s_{t+1} \sim P(s_{t+1}|(s_0, a_0), (s_1, a_1), \ldots, (s_t, a_t)) \tag{2.10}$$

which can be interpreted as at time step t, the next state s_{t+1} is a sample from the probability distribution P given the entire history of states and actions [49]. The first-order Markov

Decision Process (MDP) is a subset of such mathematical models that satisfy the Markov Property given by

$$s_{t+1} \sim P(s_{t+1}|(s_t, a_t)) \tag{2.11}$$

or simply - at time step t, the next state s_{t+1} depends only on the previous state s_t and action a_t. This assumption has two important consequences. Firstly, referring only to the last pair of state and action significantly reduces both the computational and memory costs. Secondly, from a logical point of view, it is unnecessary for decision-making to consider the whole history of states and actions. To comprehend this, let us ask ourselves if it is important for a human to consider what they had for breakfast a week ago before making a decision about what they will wear today.

The goal of the agent is to maximize the return $R(\tau)$, the weighted accumulated reward defined by

$$R(\tau) = \sum_{t=0}^{T} \gamma^t r_t \tag{2.12}$$

where γ is a discount factor $\gamma \in [0, 1]$. Another important definition is the objective function $J(\tau)$, which is the expected value of the return over many episodes:

$$J(\tau) = \mathbb{E}_{\tau\sim\pi}[R(\tau)] = \mathbb{E}_{\tau}[\sum_{t=0}^{T} \gamma^t r_t] \tag{2.13}$$

Maximizing the objective is the same as maximizing the return. This is done by retrieving and optimizing one or several functions [49]:

1. Policy π
2. Value function $V^{\pi}(s)$ or $Q^{\pi}(s, a)$
3. The environment model $P(s'|s, a)$

The policy suggests to the agent the best possible action given the circumstances. Value functions serve as support for state and action evaluation. The environment model simulates the information about the environment, yielding action predictions [49].

These functions are highly algorithm- and task-dependent; thus, they will be explained with reference to each algorithm individually in the following sections. In this way, the Markov Decision Process lies at the core of the Reinforcement Learning field.

In conclusion, it is important to mention some deviations such as higher-order MDP and Partially observable Markov Decision Process (POMDP) [49]. Higher-order MDP models are often used when the Markov Property (2.11) is insufficient in terms of complex actions sequences. The N-th order Markov Decision Process is defined as

$$s_{t+1} \sim P(s_{t+1}|(s_t, a_t), (s_{t-1}, a_{t-1}), \ldots, (s_{t-N+1}, a_{t-N+1})) \tag{2.14}$$

which is helpful if a decision cannot be made based only on the last pair of states and actions due to task complexity and if important dependencies are encoded further in the past.

Partially observable Markov Decision Processes, on the other hand, extend the definition of the model by introducing the uncertainty of the current state of the model via an observation model [50]. This addition is vital due to two reasons: firstly, not all information for decision-making is provided to the model; secondly, due to sensor noise and environment imperfections, the model cannot be perfectly observable [49]. Figure 2.9 explains the difference between MDP and POMDP using the example of a chess game [51].

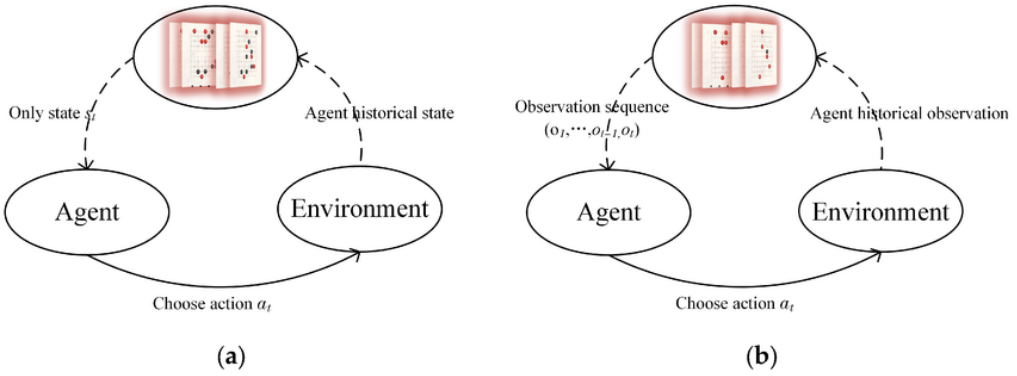


Figure 2.9.: (a) MDP has access to the fully observable game state. (b) POMDP has access only to limited information about the current game state.

POMDP-driven RL agents are widely applied from robotic systems [52] to automotive intelligent agents [53] due to their robustness and versatility.

2.2.2. Foundations of Reinforcement Learning

As mentioned in section 2.2.1, the Markov Decision Process is the underlying concept of many fundamental RL Algorithms [49]. In this chapter we will delve further into the basics of Reinforcement Learning, discuss the taxonomy of RL algorithms and briefly analyze some examples based on their relation to our topic. The variety of approaches in RL has led to a systematical distinction between algorithm families. Firstly, RL Algorithms are distinguished by their environment perception:

- **Model-based RL Algorithms** use a learned or a priori known model of the dynamics of the environment [49]. Having the environment model $P(s'|s,a)$ at its disposal, an agent may deduce the best possible sequence of actions $a_1, a_2, \ldots a_n$ from the current state s for reward maximization.

  Since modelling the dynamic Environment is a complex task in robotic applications, we will only mention some related works [54][55] as well as Dynamic Programming (DP) with Policy Iteration and Asynchronous Dynamic Programming [56].

- **Model-free RL Algorithms** are a broad collection of methods that do not directly use a model of the environment, but rather interactively obtain crucial information from interaction. This family of approaches includes Monte Carlo - based and Temporal Difference - based methods [56].

The main idea of Monte Carlo algorithms is to obtain experience by sampling (s, a, r) tuples from simulated or actual interactions with the environment [49]. Some robotic control algorithms utilize this approach, e.g., Monte Carlo Expectation Maximization (MCEM) [57] and Monte Carlo Tree Search (MCTS) [58], however, sampling may become inefficient in robotic applications due to the curse of dimensionality [55].

Temporal Difference Algorithms, on the other hand, learn by updating their estimates based on the difference between the current and predicted estimates [56]. Therefore, like Monte Carlo Methods, they do not require an environment model and like Dynamic Programming, they bootstrap - dynamically update their estimates based on parts of learned estimates. Among the famous Temporal Difference algorithms widely used in robotics RL are Q-Learning [59] and SARSA [55], which will be described later in this section.

Secondly, Model-free RL Algorithms differ by their optimization techniques:

- **Policy-based RL Algorithms** focus on the direct maximization of the objective $J(\tau)$ by optimizing the policy $\pi(a|s)$. Such algorithms are also called Policy Search [55] or Policy Approximation [56] methods due to their gradient-based approach to solving the problem.

  The big advantage of such methods is their versatility - they can be applied to either discrete or continuous tasks, which makes them promising for robotic applications. The initial Policy-based algorithm dates back at least to REINFORCE [60], which served as a basis for new algorithms such as the Actor-Critic algorithms family covered later in this section, especially Soft Actor-Critic [27] which is widely used in robotic RL applications [18][27] and will be explicitly discussed in section 2.2.3.

- **Value-based RL Algorithms** propose estimating the utility of states s or state-action pairs (s, a) by learning a respective value function $V^\pi(s)$ or $Q^\pi(s, a)$ [49]. Such algorithms are more sample-efficient, but in high dimensional problems such as robotics they become computational costly [55].

  Differences in value-based approaches are mostly conditioned by update rules and predictions [56]. The most famous among them are Q-Learning, SARSA and Deep Q-Learning, on examples of which we will study the main principles of RL in following sections. Despite difficulties in robotic applications, some new promising algorithms like QT-Opt [61] continue to arise.

Lastly, RL Algorithms differ by leveraging their experience to learn:

- **On-Policy RL Algorithms** utilize only the current policy π, restricting to the strategy chosen in the latest step [49].

- **Off-Policy RL Algorithms** can use all collected data in training, independent of the current policy π [56]. We will analyze the differences between these two families of approaches using the Cliff Walking example [62].

The world of RL Algorithms is expansive, but with help of presented taxonomy we will keep track on only the useful methods for this work. Figure 2.10 summarizes the above.

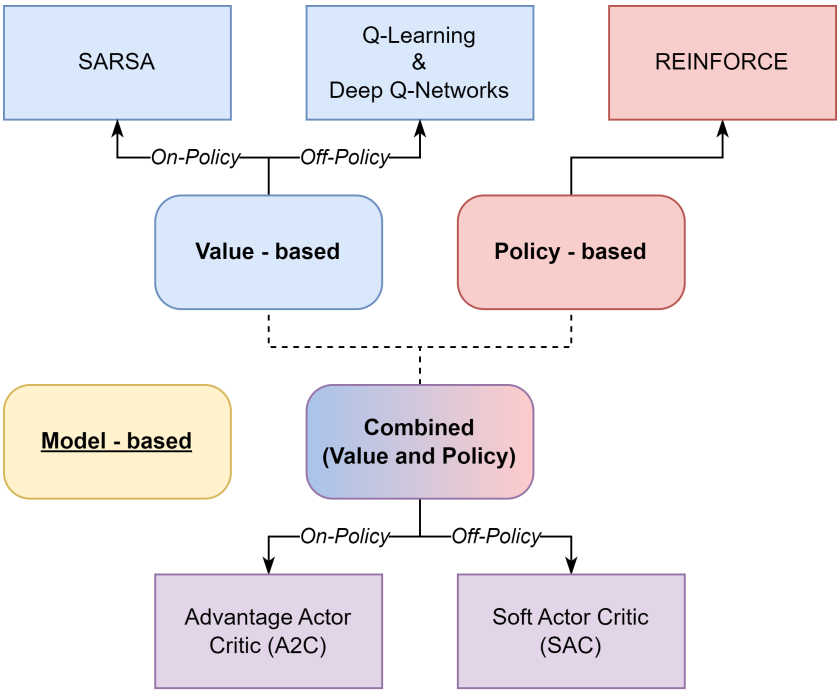


Figure 2.10.: Selection of Reinforcement Learning Algorithms and their corresponding families.

In the following sections, we will examine some basic algorithms, analyze their structure and approach, and discuss their applicability to our work.

Q-Learning

Q-learning is a model-free value-based off-policy temporal difference algorithm [56]. This algorithm was developed by Chris Watkins in 1989 [63] and three years later its convergence proof was published [64].

As a traditional value-based algorithm, Q-learning uses temporal difference control to optimize the state-action value function $Q^{\pi}(s,a)$. As mentioned above, Temporal Difference combines Monte Carlo sampling ideas and Dynamic Programming, which is built on the Bellman Equation [56]. In Dynamic Programming, the Bellman Equation is a recursive formula for the value-function update given as

$$V_{\pi}(s) = \mathbb{E}_{\pi}[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s] \quad \forall s \tag{2.15}$$

$$Q_{\pi}(s,a) = \mathbb{E}_{\pi}[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} | s_t = s, a_t = a] \quad \forall s,a \tag{2.16}$$

where $v_\pi(s)$ is the state value and $q_\pi(s,a)$ is the state-action value function. For Dynamic Programming-based approaches such as Policy or Value Iteration these update rules are sufficient as the environment model is known [56].

As discussed at the beginning of this chapter, temporal difference control models have no a-priory knowledge about the environment and must derive the value function from collected experience. In general [56], they follow the update rule

$$NewEstimate \leftarrow OldEstimate + StepSize \underbrace{[Target - OldEstimate]}_{EstimateError} \tag{2.17}$$

Q-learning utilizes the following update rule for the state-action value function:

$$Q(s_t,a_t) \leftarrow [1-\alpha]Q(s_t,a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1},a)] \tag{2.18}$$

where α is a step-size parameter, $Q(s_t,a_t)$ is the state-action value of the current state s_t given action a_t, r_t is the reward obtained from step t and γ is a discount factor.

Let us discuss the logic behind this value function formulation. After performing a step (thus interacting with the environment and receiving a reward), the agent updates the state-action value by $[1-\alpha]Q(s_t,a_t) + \alpha r_{t+1}$. The recursive term $\alpha\gamma \max_a Q(s_{t+1},a)$ checks all possible actions a resulting in new state s_{t+1} with the corresponding $Q(s_{t+1},a)$ and chooses the most promising one by adding $\alpha\gamma \max_a Q(s_{t+1},a)$ to the $Q(s,a)$. As is clearly seen from equation (2.18), Temporal Difference-based Q-learning is a bootstrapping RL algorithm,utilizing part of the already learned estimates for the new value $Q(s,a)$, which lowers the variance compared to Monte Carlo-based methods [49].

Now, as the Q-function is defined, it is important to define the agent policy, specifically, how the agent will decide which action to perform based on the current value of the function. The most obvious decision would be to choose the most promising action with respect to the given Q-function information, a.k.a greedy policy:

$$\pi(a) = \arg\max_a Q(s,a) \tag{2.19}$$

However such policy provides no guarantees that the agent will explore the whole state-action space equally, resulting in insufficient information for correctly updating $Q(a,s)$ [49]. A common choice for an exploration-exploitation balance is the ϵ-greedy policy, which follows the current policy with a probability of $1-\epsilon$ and chooses the random action otherwise [56].

Q-learning is an off-policy RL-algorithm due to the $\max_a Q(s_{t+1})$ term, meaning that it updates the Q-function optimally, regardless of the chosen strategy [56]. As explained above, agent considers all possible outcomes of possible actions and chooses the most rewarding action to update $Q(s,a)$. Note how this is independent of the actual chosen action a that the agent will perform by following the policy $\pi(s)$, greedy or not [49]. As of now, Q-learning is still applicable in some continuous robotic applications with the help of action and state discretization [59][65]. Next, we will analyze a very similar algorithm for on-policy Temporal Difference control - SARSA.

SARSA

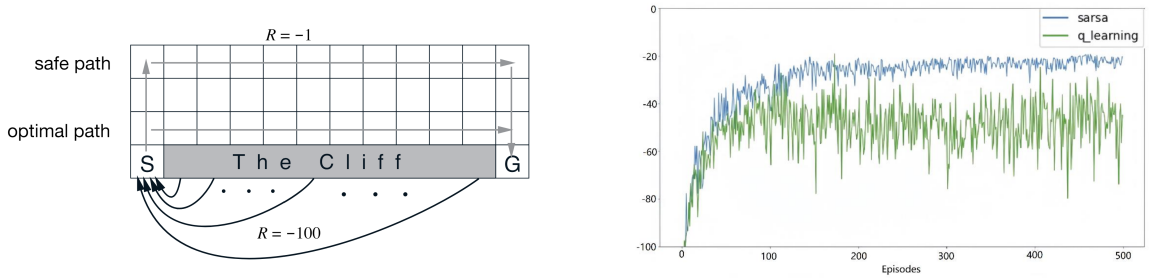
SARSA (State-Action-Reward-State-Action) is a model-free value-based on-policy temporal difference algorithm [49]. Proposed in 1994 by Rummery and Niranjan [66] five years after the Q-learning algorithm, SARSA has one notable difference from its predecessor - the on-policy update function.

SARSA follows the value update rule presented below [56]:

$$Q(s_t, a_t) \leftarrow [1 - \alpha]Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1})] \tag{2.20}$$

Compared to Q-Learning update rule (2.18), SARSA does not choose the $\max_a Q(s_{t+1}, a)$, but updates the value function according to the current policy ($Q(s_{t+1}, a_{t+1})$), which places it in the class of on-policy algorithms [55]. But how does such a change affect the performance of the algorithm? In their work, authors asserted that the standard Q-learning algorithm lacks robustness with respect to the ϵ-greedy policy [66]. This is indeed correct, since the off-policy Q-learning update is guided by $\max_a Q(s_{t+1}, a)$, which does not take into account the possible ϵ-greedy action selection resulting in a probable significant penalty, despite learning the optimal policy [56]. This problem is well demonstrated in the Cliff Walking thought experiment, comparing on- and off-policy algorithms using the SARSA and Q-learning examples [56][62].

Imagine a grid world with start and goal states, divided by a cliff with terminal states and a significant negative reward (Figure 2.11 (a)). The agent may circumvent the cliff, but every step would cost it a small negative reward, forcing the agent to walk far from the cliff. Since we want to explore the space more efficiently, we will apply the ϵ-greedy policy to both algorithms.



(a) Cliff Walking Problem [56].

(b) Sum of rewards during an episode [62].

Figure 2.11.: Comparison of SARSA and Q-Learning Performances.

As a result, Q-learning finds the optimal path, walking near the cliff, ignoring the possibility of falling in by executing the ϵ-greedy action due to the $\max_a Q(s_{t+1}, a)$-term. Conversely, SARSA recoginzes such an outcome because of its on-policy value-function update $Q(s_{t+1}, a_{t+1})$ and behaves more cautiously, choosing the safe path. As a result, SARSA obtains more rewards per episode than Q-learning (Figure 2.11 (b)).

By reducing ϵ, both algorithms converge to the optimal policy [56]. Comparing these two methods, it is difficult to state which approach is better. Paths generated by SARSA are longer but safer, while Q-learning proposes optimal but riskier solutions [62][65].

As with traditional Q-Learning, unmodified SARSA is still utilized in some RL-based robotic applications like the pick and place task [65]. Over time, however, several novel modifications of SARSA have proven to be more efficient. For example, Expected SARSA updates the value function as follows [56]:

$$Q(s_t, a_t) \leftarrow [1 - \alpha]Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \mathbb{E}[Q(s_{t+1}, a_{t+1})|s_{t+1}]] \tag{2.21}$$

or

$$Q(s_t, a_t) \leftarrow [1 - \alpha]Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \sum_a \pi(a|s_{t+1})Q(s_{t+1}, a)] \tag{2.22}$$

Simply put, Expected SARSA uses the expected value corresponding to the current policy (which also makes it an on-policy approach) to update the value function. It is proven that under the same conditions, Expected SARSA converges faster than the traditional approach [65]. Another example is Smoothed SARSA [67]. This approach utilizes the POMDP we discussed in section 2.2.1 to model the uncertainties of state estimations and modify SARSA to take those into consideration.

Both SARSA and Q-learning store their values in lookup tables, making the policy decision process a simple scan of corresponding states and actions. Our next approach utilizes a neural network architecture for policy estimation.

Deep Q-Networks

Deep Q-Network (DQN) is a model-free value-based off-policy temporal difference method proposed in 2013 by Volodymyr Mnih et al. [68].

Deep Q-Networks algorithms are based on Q-Learning, which, as seen in section 2.2.2, guide the agent towards the optimal strategy according to Sutton and Barto [56], but introduces the neural network architecture for value function approximation [49]. We also already analyzed some of the Artificial Neural Networks architectures in section 2.1, and in this part we will discuss the differences ANNs can bring to Reinforcement Learning.

Today's state of the art of RL algorithms is predominantly based on deep architectures, especially in continuous problems [55]. The primary reason behind this is the problems with overwhelmingly big or continuous state-action spaces. Firstly, even with an appropriately balanced exploration-exploitation strategy, the agent does not visit many states, thus resulting in undetermined $Q(s, a)$ in those states-action pairs. Secondly, tabular methods lack the extrapolation between cell values, meaning that similar state-action pairs may result in differing Q-values based on the agent trajectory [49]. For example, in the original paper, the state space of the problem was one of the six Atari games [68]. Even if the state space was discrete, it contained a great number of states, each corresponding to a different possible frame.

The main idea of Deep Q-Networks is to replace the tabular representation of the value function $Q(s, a)$ with a deep neural network, predicting the Q-function for each (even not visited) state based on the collected experience (see Figure 2.12).

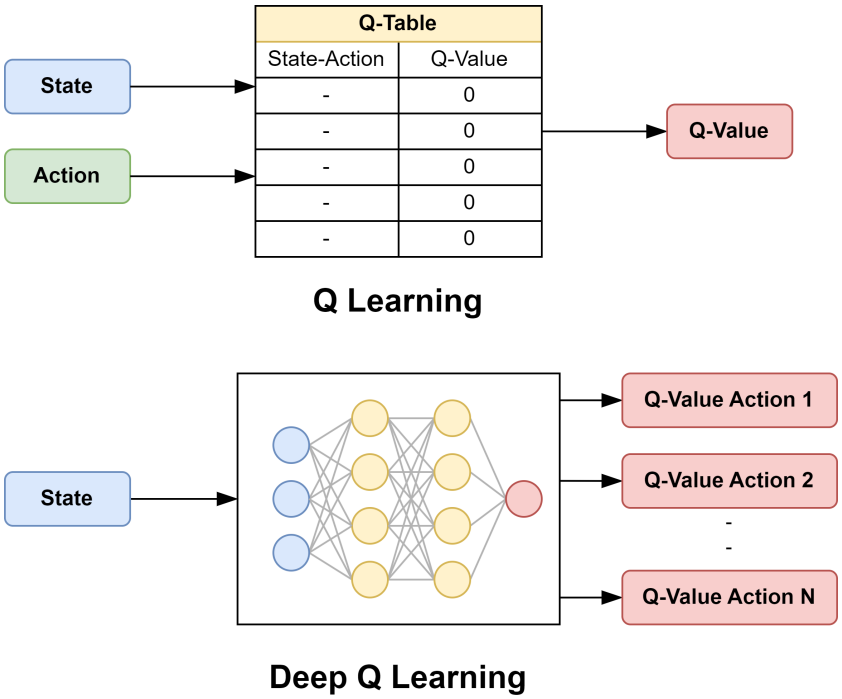


Figure 2.12.: Q-learning compared with DQN.

Standard Deep Q-Learning still uses the Bellman Equation for value function estimation, but additionally, it optimizes the parameters of the NN module every n steps, utilizing collected experience to predict the Q-values of unseen state-action pairs [49]. Pseudocode of Deep Q-Learning Algorithm can be seen in Figure 2.13.

Balancing at the junction between standard RL and deep learning, the DQN algorithm opens up a view of the field of Deep Reinforcement Learning algorithms, utilizing many tricks from both domains striving for the best possible solutions. For instance, like in the original paper[68], the NN block can be realized as a Convolutional Neural Network for Computer Vision-based robotic manipulation [69], ϵ-greedy exploration is substituted with Boltzmann exploration [70][49]. Notably, training two twin Deep Learning Architectures for Q-value estimation is called Double Q-learning [71] functions by simultaneous prediction and choosing the best Q-value for update. Double Q-learning networks [72] with Prioritized Experience Replays [73] are used, for example, for mobile robotics [74] and as a building block for more complex algorithms [27].

The last step towards understanding the fundamental concepts of Reinforcement Learning algorithms will be in the Actor Critic family of algorithms, allowing us to process continuous tasks and proven to be more suitable for Hierarchical Reinforcement Learning and Skill-based approaches (see section 2.2.4) which is one of the main topics of the work.

Algorithm Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for

Figure 2.13.: Deep Q-Learning Algorithm pseudocode [68].

Actor Critic Algorithms

The family of Actor-Critic algorithms combines a Policy-based Actor with Value-based Critic [49]. Most Actor-Critic algorithms use on-policy updates [27].

We will start by understanding the underlying concepts of the Policy-based Actor. The standard policy-based approach REINFORCE was introduced by [60] and proposes the straightforward idea - tune the optimal strategy $\pi(a|s)$ for objective function $J(\pi)$ maximization. This involves function optimization, so $\pi_\theta(a|s)$ is defined by tunable parameters θ. An obvious solution would be applying a universal approximator represented by an ANN [49]. Let us formulate the problem to solve:

$$\max_\theta J(\pi_\theta) = \max_\theta \mathbb{E}_{\tau \sim \pi_\theta}[R(\tau)] \tag{2.23}$$

By applying the gradient ascent update, we obtain the update rule for REINFORCE [56]:

$$\theta \leftarrow \theta + \alpha \nabla_\theta J(\pi_\theta) \quad \textit{where} \tag{2.24}$$

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_t[R_t(\tau) \nabla_\theta \log \pi_\theta(a_t|s_t)] \tag{2.25}$$

As mentioned above, it is common to utilize ANNs in Reinforcement Learning. The architecture of the ANN will handle the updates,but it is worth mentioning precedent collection for more stable batch updates (recall section 2.1.1).

But what happens in the case of sparse rewards? REINFORCE is a Monte Carlo-based approach [56], meaning that its estimates derived from rewards are typically of higher variance than those from value functions.

Advantage Actor-Critic (A2C) introduces a learned reinforcing signal generated by an advantage function $A^{\pi}(s,a)$, which is more effective than the accumulated reward [56]. The core idea is to substitute $R_t(\tau)$ with an Advantage function (Actor) and generate it with value-based network (Critic)[49]:

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_t[A_t^{\pi_\theta} \nabla_\theta \log \pi_\theta(a_t|s_t)] \quad where \tag{2.26}$$

$$A_t^{\pi_\theta} = A^{\pi_\theta}(s_t,a_t) = Q^{\pi_\theta}(s_t,a_t) - V^{\pi_\theta}(s_t) \tag{2.27}$$

We point out the difference between equations (2.25) and (2.26) where the REINFORCE Monte-Carlo update rule transforms into the A2C Actor policy approximation.

The choice of the advantage function $A^{\pi}(s,a)$ is determined by natural logic: since $\mathbb{E}_{a\in A}[A^{\pi}(s_t,a)] = 0$, or put differently, if all actions are equivalent, the advantage is 0, therefore the probability of their selection will not be updated [49]. Statistically, advantage function-based algorithms have one of the lowest possible variances [75], with the downside of requiring the critic module to estimate $A^{\pi}(s,a)$.

Since $A^{\pi_\theta}(s_t,a_t) = Q^{\pi_\theta}(s_t,a_t) - V^{\pi_\theta}(s_t)$ is based on two value functions, the naive choice would be to tune them by two separate deep frameworks as discussed in the previous section, but in practice that would mean ensuring both networks yield matching estimations and have enough computational resources for parallel training [49].

The A2C algorithm suggests using an ANN for the estimation of $V^{\pi_\theta}(s_t)$ and $Q^{\pi_\theta}(s_t,a_t)$ via Temporal Difference n-step bootstrapping:

$$Q^{\pi_\theta}(s_t,a_t) = \mathbb{E}_{\tau\sim\pi}[\sum_{k=0}^{n-1} \gamma^k r_{t+k}] + \gamma^{n+1} V^{\pi}(s_{t+n+1}) \tag{2.28}$$

resulting in an advantage function approximation [49][76]:

$$A^{\pi_\theta}(s_t,a_t) = Q^{\pi_\theta}(s_t,a_t) - V^{\pi_\theta}(s_t) \approx \sum_{k=0}^{n-1} \gamma^k r_{t+k} + \gamma^{n+1} V^{\pi}(s_{t+n+1}) - V^{\pi}(s_t) \tag{2.29}$$

The hyperparameter n determines the number of rewards accounted for and greatly influences the model's performance. A large value for n corresponds to high variance, while a low n increases the bias [49]. The authors of [75] proposed a generalized solution for the advantage function approximation (GAE), which avoids the direct specification of n, but introduces λ. Generalized advantage estimation is given in the equations (2.30) and (2.31):

$$A_{GAE}^{\pi}(s_t,a_t) = \sum_{k=0}^{\infty} (\gamma\lambda)^k \delta_{t+k} \quad where \tag{2.30}$$

$$\delta_t = r_t + \gamma V^{\pi}(s_t) \tag{2.31}$$

It is difficult to tune n since it is not obvious up to what point estimations should be included. In Generalized advantage estimation, $\lambda \in [0,1]$ is a parameter that behaves similarly to n, but provides a much more intuitive tool for tuning - low $\lambda \to 0$ corresponds to high bias since

it converges the model to Temporal Difference $GAE(\lambda, 0) = r_t + \gamma V^\pi(s_{t+1} - V^\pi(s_t))$, while large $\lambda \to 1$ steers n to infinity: $GAE(\lambda, 1) = \sum_{k=0}^{\infty} \gamma^k \delta_{t+k}$ [75].

In 2016 the Asynchronous Advantage Actor-Critic (A3C) algorithm was presented [76], introducing asynchronous parallelization. Furthermore, the authors of the A3C algorithm empirically proved that adding the entropy of the policy to the objective function at the actor update step improves learning by leveraging early convergence to incorrect policies [76].

Today, the actor-critic framework is widely used in robotics due to its direct policy approximation and the possibility of continuous integration [77][78][79]. With an understanding of the underlying concepts of this family, we may proceed to the algorithm we will utilize in our work - Soft Actor Critic [27].

2.2.3. Soft Actor Critic

Soft Actor Critic (SAC) is a model-free off-policy Reinforcement Learning algorithm from the Actor Critic family, utilizing soft policy iteration [80] and maximum entropy RL-formulation [27].

SAC quickly became a highly demanded algorithm in robotics [18][81] due to its stability and scalability considerations. In the following section, we will analyze this algorithm and its underlying concepts which make it the perfect candidate for our purposes.

Soft Policy Iteration

We briefly mentioned the Policy Iteration Algorithm in the context of model-based RL in section 2.2.2. The Policy Iteration is a general algorithm which is also applicable for policy convergence in a model-free setting as described in the [80]. Like the original algorithm [56], the Soft Policy Iteration algorithm consecutively executes policy evaluation and policy improvement, but augments the value functions with an entropy term $\mathcal{H}$. Let us analyze these steps further:

- **Soft Policy Evaluation** estimates the value function given the current policy π [56]. The SAC algorithm defines the soft Q-value function as

$$Q_{soft} = r_t + \mathbb{E}_{(s_{t+1},\ldots)\sim\rho_\pi}[\sum_{l=1}^{\infty} \gamma^l (r_{t+l} + \alpha \mathcal{H}(\pi^*(\cdot|s_{t+l})))] \tag{2.32}$$

  where the optimal policy is given by

$$\pi^* = \arg\max_\pi \sum^t \mathbb{E}_{(s_t,a_t)\sim\rho_\pi}[r(s_t, a_t) + \alpha \mathcal{H}(\pi(\cdot|s_t))] \tag{2.33}$$

  The difference from the original Policy Evaluation is the entropy-term $\mathcal{H}$ with a temperature parameter α and trajectory distribution marginals ρ_π with respect to the policy π [80][27]. Authors compute the Q-value by iteratively applying the extended Bellman operator $\mathcal{T}^\pi$ resulting in the update rule

$$\mathcal{T}^\pi Q(s_t, a_t) = r(s_t, a_t) + \gamma \mathbb{E}_{s_{t+1}\sim p}[\mathbb{E}_{a_{t+1}\sim\pi}(Q(s_{t+1}, a_{t+1}) - \log \pi(a_{t+1}|s_{t+1}))] \tag{2.34}$$

We will not go into a detailed decomposition of equations (2.32), (2.33) and (2.34) and refer to the original paper [27] and the explanation of the unmodified Policy Evaluation [56], but provide the intuition behind this augmentation, which can be explained by the additional freedom in policy. This is the reason behind the "soft-" prefix of the algorithm.

- **Soft Policy Improvement** updates the policy towards the exponential of the updated Q-value function from the last step [27]. SAC proposes a probabilistic view of the problem in which the policy π can be chosen from some a set of policies Π. Authors choose to update the policy according to the KL divergence discussed in 2.1.2. The policy update rule can be written as

$$\pi_{new} = \arg\min_{\pi' \in \Pi} \mathbb{KL}(\pi'(\cdot|s_t)||\frac{\exp(Q^{\pi_{old}}(s_t, \cdot))}{Z^{\pi_{old}}(s_t)}) \tag{2.35}$$

  Where $Z^{\pi_{old}}(s_t)$ normalizes the resulting distribution [27].

Analogous to the unmodified algorithm [56], Soft Policy Iteration combines the mentioned steps and ensures the convergence to the optimal policy π^* as was proven in [27].

Entropy-Based Actor Critic

Soft Actor Critic does not have a conventional advantage function described in section 2.2.2 under Advantage Actor Critic. Instead, it utilizes the Soft Policy Improvement algorithm with soft value functions to maximize the augmented objective function

$$J(\pi) = \sum_{t=0}^{T} \mathbb{E}_{(s_t,a_t)\sim\rho_\pi}[r(s_t, a_t) + \alpha\mathcal{H}(\pi(\cdot|s_t))] \tag{2.36}$$

to guide the policy distribution π [27]. Nevertheless, the described architecture is an instance of the Actor-Critic family. SAC uses stochastic gradient descent to optimize:

1. Actor policy Network $\pi_\phi(a_t|s_t)$.
2. Soft Double Q-Network Critic $Q_{\theta_1}(s_t, a_t)$ and $Q_{\theta_2}(s_t, a_t)$.
3. Parameterized Value- and Target-Value Functions $V_\psi(s_t)$ and $V_{\bar{\psi}}(s_t)$.

Soft Actor Critic uses five ANNs instead of the two in A2C mainly for stability of the training which has shown impressive results [27]. We mentioned Double Q-Networks in section 2.2.2, and the soft versions differ by using equation (2.32) for Q_{soft} estimation and trained to minimize residual

$$J_Q(\theta) = \mathbb{E}_{(a_t,s_t)\sim\mathcal{D}}[\frac{1}{2}(Q_\theta(s_t, a_t) - r(s_t, a_t) - \gamma\mathbb{E}_{s_{t+1}\sim p}[V_{\bar{\psi}}(s_{t+1})])^2] \tag{2.37}$$

where $\mathcal{D}$ is a replay buffer and target Value-function $V_{\bar{\psi}}(s_{t+1})$ results in exponential moving average of $V_\psi(s_{t+1})$ (update rule for the target network can be written as $\bar{\psi} \leftarrow \tau\psi + (1-\tau)\bar{\psi}$

for the training hyperparameter τ) [27]. The value function $V_\psi(s_{t+1})$ itself minimizes the squared residual error

$$J_V(\psi) = \mathbb{E}_{s_t \sim \mathcal{D}}[\frac{1}{2}(V_\psi(s_t) - \mathbb{E}_{a_t \sim \pi_\phi}[Q_\theta(s_t, a_t) - \log \pi_\phi(a_t|s_t)])^2] \tag{2.38}$$

To summarize, the Value Function $V_\psi(s_{t+1})$ is necessary for computing the soft Q-function $Q_\theta(s_t, a_t)$, which is optimized by taking a stochastic gradient of equation (2.38). The Target Value Function $V_{\bar{\psi}}(s_{t+1})$ is optimized more slowly than the Value Function $V_\psi(s_{t+1})$ and helps stabilize the learning. The Double Q-Network $Q_\theta(s_t, a_t)$ generates the value estimations analogous to the Critic explained in the previous section, uses soft Q-values [80], takes the minimum of two estimated values for further computations, and calculates the gradient of (2.37) for update. Finally, $\pi_\phi(a_t|s_t)$ minimizes the objective function (2.36) taking into account the encouraging environmental exploration entropy $\mathcal{H}$ [27].

Term included in SAC entropy allows the policy π to be much more flexible while stabilizing training with additional networks. This is why Soft Actor Critic has outperformed many of conventional algorithms as was experimentally proven by the authors of [27]. For our work, it is important that Soft Actor Critic has a probabilistic structure and has a convenient method for policy guidance, irreplaceable in high-level policy approaches we will discuss in the following section.

2.2.4. Hierarchical Reinforcement Learning and Skills

The term "Curse of Dimensionality" was introduced by Richard E. Bellman [82] and describes the problem arising with high-dimensional problems - with a linear increase of degrees of freedoms the number of equally describing data instances grows exponentially. With rising demands on dynamic problems and safety requirements modern problems may therefore require a large amount of parameters to consider on low levels. Introducing hierarchy in the problem formulation helps to divide it into different levels for efficient execution [83].

Hierarchical Reinforcement Learning (HRL) addresses this issue in the Reinforcement Learning field by considering several levels of abstraction, some or all of which may be considered as nested RL-problems themselves [14] (see Figure 2.14). Recalling similar ideas, one can mention semantic parsing [8], Dynamic Movement Primitives [84][85] and Skills [16].

Hierarchical Reinforcement Learning utilizes a terminology to create a hierarchy of subtasks: *high-level* task is the high-level hierarchy task with a bigger level of abstraction, while a *low-level* task is the task operating with atomic actions, states and policies [14]. In robotic RL, a high-level policy is usually a human-interpretable action or complex movement primitive [8][85], which makes Hierarchical Reinforcement Learning in robotics also interesting from a semantic point of view.

In our work, we will concentrate on Skill learning inspired by human subconscious action chains [14]. In contrast to the relative field of Behavioral Cloning, which teaches the robot to mimic demonstrated actions, Skill HRL aims to learn action sequence chains at a high level to later transfer and apply them to different tasks [18]. From another point of view, Skills differ

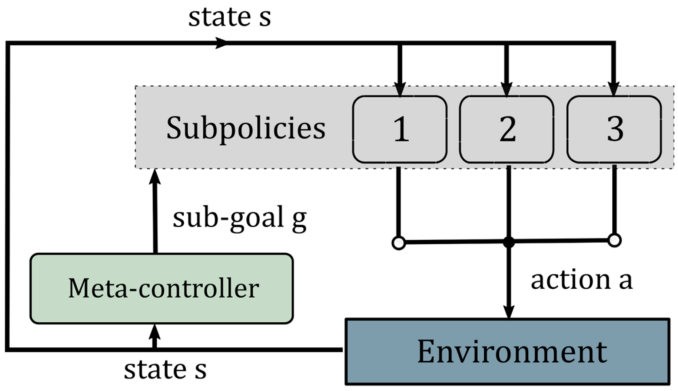


Figure 2.14.: Concept of Hierarchical Reinforcement Learning [83].

from Dynamic Motion Primitives because of their adaptability - Dynamic Motion Primitives are often set to be pre-programmed motion sequences or probabilistic atoms the agent has in its disposal to navigate the task [84], while Skills are inferred from the demonstration itself without any external artificial restrictions.

With that said, Skills in HRL serve as a high level of abstraction for a manipulation sequence while each skill can be decomposed into a sequence of state-action pairs (s_t, a_t) representing the low-level interaction with the environment [18]. Hierarchical Reinforcement Learning is easy to confuse with Meta-RL, which is more of an organizational tool allowing the execution of Meta-Learning by subdividing the task into different levels of abstraction. In the following part, we will analyze this application further.

2.2.5. Meta - Reinforcement Learning

The word "Meta" comes from Greek and means "beyond" [17]. Therefore, Meta-Learning can be explained as "learning to learn" [14]. In this field of study algorithms process meta-data to gain substantial knowledge on one task, which guides the learning of the downstream task based on its basis [18].

To grasp this concept, let us analyze the phenomenon of Meta-Learning with the detached example. If a person is learning to be a cook, they will begin their study with a simple task (for example, preparing a salad) where they will learn basic skills on how to choose the ingredient proportions and chop and cook them. These atomic skills are not only applicable to the salad preparation, but rather to a wide selection of recipes. This person will have to still learn to cook another dish, but their obtained skills will help to guide this learning more efficiently compared to an untrained individual. In this example, preparing a salad is a particular task, inferred abilities to cut vegetables are the skills and learning how to cook can be expressed as a Meta-Learning goal.

Similarly, Meta-Reinforcement Learning is Meta-Learning projected onto the RL field [18]. The main idea of skill transfer between tasks dates back at least to the SKILLS paper from 1994 [16]. In the last section, we stated that Hierarchical Reinforcement Learning differs from

Meta-RL by definition. Now, we can compare the aims of these approaches and strictly denote the difference:

- **Hierarchical Reinforcement Learning** breaks down the complex problem into hierarchical subtasks for efficient agent learning on this particular task.
- **Meta - Reinforcement Learning** aims to optimize the skills gained from and transferred between a variety of tasks so that they are efficient for a meta-goal.

Obviously, HRL provides an abstraction of the executed task that is helpful for Meta-RL, which is often used [86][18]. In our work, we will also utilize a Hierarchical Reinforcement Learning model for Meta-Learning, where obtained Skills will serve as the high-level policy for downstream tasks.

2.3. Clustering Algorithms

Clustering Algorithms have found applications in a wide range of fields, including Computer Vision (CV) [87], intelligent document processing (IDP) [88], facial recognition [89] and much more [90]. The main task of a of clustering algorithm is to allocate similar patterns to corresponding clusters [15]. In this chapter, we will discuss several approaches from legacy algorithms to modern ones to achieve that goal.

2.3.1. Foundations of Clustering Algorithms

As mentioned, each clustering algorithm aims to assign data to one of the clusters based on some metric. First, it is notable to mention two approaches dictated by the task:

- **Supervised Clustering Algorithms** work with labeled data $(x, t) \in (X, T)$. This allows for end-to-end straight loss computation and can be implemented with algorithms discussed in section 2.1.
- **Unsupervised Clustering Algorithms**, on the other hand, try to cluster unlabeled data $x \in X$ based on intrinsic data structure [89]. These algorithms cannot rely on the target estimation error and introduce similarity measures for cluster assignment. For our future work, these algorithms are of interest and we will focus on them.

Furthermore, the clustering algorithms taxonomy also identifies two groups divided by approach [91]:

- **Hierarchical Clustering** splits a dataset (top-down) or merges data (bottom up) in a sequential manner, formulating a hierarchical tree [91]. Such approaches have some advantages, for instance, if the dataset can be well described by only distance metrics between pairs of instances [29].

- **Partitional Clustering**, on the other hand, does not have a hierarchical structure between the clusters. Among others, model-based partitional clustering algorithms stand out by the assumption that the cluster-assigned data is generated by a probability distribution [92][91], making it possible to generate data. This assumption is crucial for our work, therefore we will primarily focus on partitional model-based clustering algorithms.

Next, we will decompose three unsupervised partitional clustering algorithms, K-means, Gaussian Mixture Model and Dirichlet Process Mixture Model, beginning as always from the one of the simplest.

K-means algorithm is a hard assignment model-based unsupervised clustering algorithm [91]. Given an unlabeled dataset $x_1, x_2, ...x_n = X$ of size N, a pre-defined number of clusters K and metric M, it applies the metric to each data point, iteratively assigning the point to the nearest cluster center and moving cluster centers according to newly assigned points with Expectation-Maximization (EM) [15]. The goal of K-means is to minimize the objective function

$$J = \sum_{n=1}^{N} \sum_{k=1}^{K} r_{nk} ||x_n - \mu_k||^2 \tag{2.39}$$

where μ_k is the centroid of cluster k and $r_{nk} \in 0, 1$ is a binary indicator of an instance x_n belonging to cluster k [15]:

$$r_{nk} = \begin{cases} 1 & \text{if } k = \arg\min_j ||x_n - \mu_j||^2 \\ 0 & \text{otherwise} \end{cases} \tag{2.40}$$

$$\mu_k = \frac{\sum_n r_{nk} x_n}{\sum_n r_{nk}} \tag{2.41}$$

Equation 2.40 is called an Expectation or E-step of the EM algorithm [15][29], since at this step the algorithm estimates the expected membership of data points x_n to clusters j. At the Maximization or M-step (2.41), the model adjusts cluster means μ_k to fit the data based on its metric M, which is Euclidean distance in the case of K-means [15] (Figure 2.15):

More generally, the E-step maximizes the negative free energy with respect to the distribution over yet unobserved dataset instances, and the M-steb maximizes negative free energy with respect to the model parameters themselves [93]. Simply put, at the Expectation step model freezes its parameters and tries to capture the dataset's structure, while at the Maximization step model freezes the estimated responsibilities of each data point and tries to optimize their parameters accordingly [15]. The convergence of the EM algorithm to a zero gradient was proven in 1983 [94]. Due to its convergence, the EM alternating algorithm has found wide application in many partitional model-based clustering algorithms, three of which are part of our analysis.

The original K-means [95] is a hard assignment clustering algorithm with a slow computation process, which makes it inefficient in modern applications. For instance, the E-step is relatively slow due to the computation of the whole dataset, hence a batch version of K-means is preferred [15]. Unmodified K-means is sometimes used in robotics applications if instances are clearly linearly separable or highly pre-processed [96]. Addressing hard

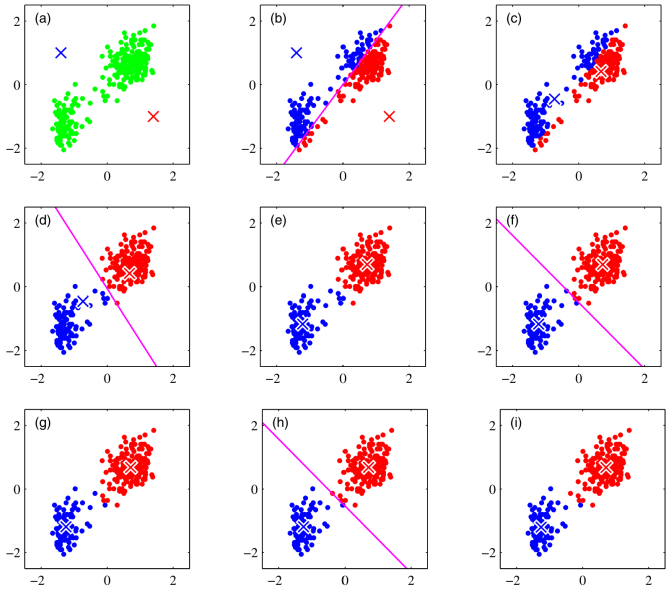


Figure 2.15.: K-means alternating E-/M-steps on Old Faithful data set [15].

assignments, the fuzzy C-means algorithm [97] proposes the idea of a degree of membership, allowing each instance x_n to participate in several clusters simultaneously. Although Fuzzy C-means is empirically effective in robotics and CV applications [98], the soft assignment of Mixture models is integrated into the distribution by definition, allowing them also to generate samples, which is, apart from clustering, an essential property of our future model.

2.3.2. Gaussian Mixture Model

The central concept in the Gaussian Mixture Model (GMM) is the latent variables model [15]. According to the formula for marginal distribution given by the summation of a joint distribution over z

$$p(x) = \sum_{z} p(z)p(x|z) \tag{2.42}$$

where $p(x)$ is the probability of obtaining an instance x, we create a model with a latent parameter z. This latent parameter can be used in cluster modelling, for instance, GMM assumes Gaussian-distributed clusters, with a categorical distribution for prior $p(z)$[15]:

$$p(z_k) = \pi_k \tag{2.43}$$

$$p(x|z_k) = \mathcal{N}(x|\mu_k, \Sigma_k) \tag{2.44}$$

By substituting 2.43 and 2.44 into 2.42, we obtain a marginal distribution of Gaussian Mixtures:

$$p(x) = \sum_{k=1}^{K} \pi_k \mathcal{N}(x|\mu_k, \Sigma_k) \tag{2.45}$$

From this model several properties should be noted:

- Number of clusters K is predefined as a hyperparameter.
- Each cluster is assumed to be Gaussian-distributed with a respective mean μ_k and covariance matrix Σ_k, while the mixing coefficient π_k determines the probability of choosing cluster k.
- Due to its probabilistic nature, the model also has a generative property.

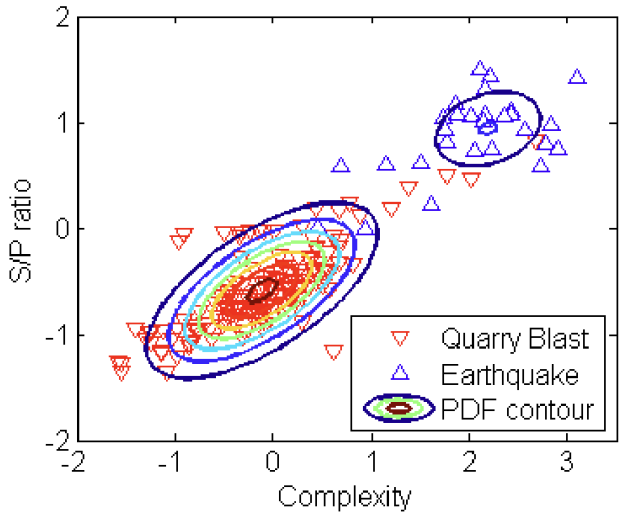


Figure 2.16.: Gaussian Mixture Model applied to the seismic events in Istanbul [99].

Gaussian distribution is a proper choice due to its empirical modelling of natural occurrences [99]. Figure 2.16 demonstrates a fitted GMM on an unlabelled dataset, but how does this model train? The goal of the GMM is to find a maximum likelihood solution [15], in other workds - to fit the model to the data for the its best representation. GMM also uses the Expectation Maximization algorithm mentioned in the previous section, but in contrast to K-means, GMM utilizes a soft assignment defined by responsibilities $\gamma(z_n k)$ at the E-Step:

$$\gamma(z_n k) = p(z_k|x) = \frac{\pi_k \mathcal{N}(x|\mu_k, \Sigma_k)}{\sum_{j=1}^{K} \pi_j \mathcal{N}(x|\mu_j, \Sigma_j)} \tag{2.46}$$

Responsibilities $\gamma(z_n k)$ are derived from Bayes' Theorem and correspond to the concept of the E-step - for each instance x_n the responsibility $\gamma(z_n k) \in 0, 1$ computes the probability of belonging to cluster k. Unlike equation (2.40) in K-means, GMM expectation can assign one data point to several clusters [15]. At the M-step, the model parameters are computed as follows:

$$N_k = \sum_{n=1}^{N} \gamma(z_{nk}) \tag{2.47}$$

$$\mu_k^{new} = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk}) x_n \tag{2.48}$$

$$\Sigma_k^{new} = \frac{1}{N_k} \sum_{n=1}^{N} \gamma(z_{nk})(x_n - \mu_k^{new})(x_n - \mu_k^{new})^T \tag{2.49}$$

$$\pi_k^{new} = \frac{N_k}{N} \tag{2.50}$$

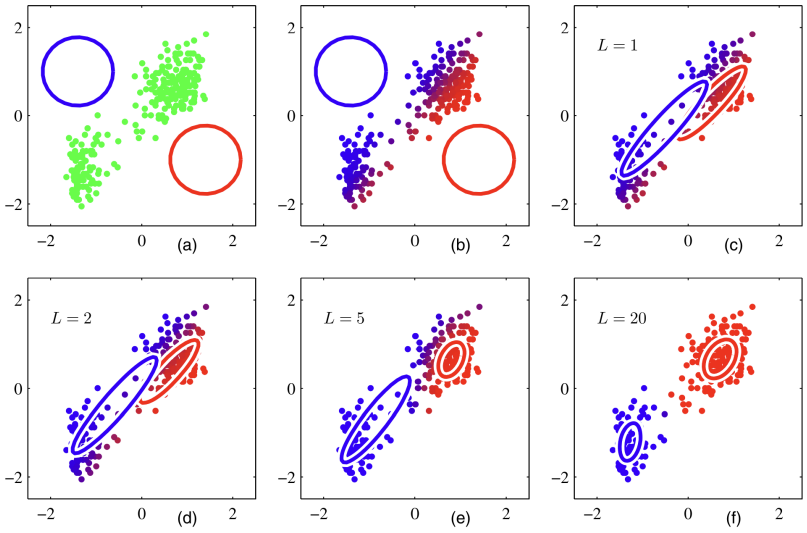


Figure 2.17.: EM algorithm on GMM using the Old Faithful data set [15].

It is clear that K-means is a special case of GMM if covariance shrinkes towards zero [15]. In this case $\gamma(z_n k) \leftarrow r_{nk}$ and GMM will converge to the hard assignment seen in previous section.

Graphical models are frequently utilized in probabilistic modeling to illustrate the dependencies among variables and parameters. In the case of the Gaussian Mixture Model, the mixing coefficient π determines the distribution over the latent variable z_n which in turn dictates the generation of the observed variable x_n. The latent variable z_n selects one of the Gaussian components characterized by its mean μ and covariance matrix Σ, therefore defining the probabilistic structure of x_n [15] (Figure 2.18).

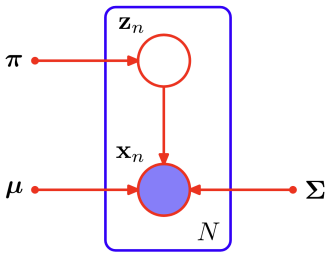


Figure 2.18.: Graph representation of a Gaussian Mixture Model [15].

Let us revisit the properties we derived from the GMM model. The first two properties

along with the generalized approach place GMM on a pedestal as one of the standard and powerful unsupervised partitional clustering algorithms [99][100][101]. The third property, its generative ability, is new compared to the previous algorithm and should be discussed in detail.

Having received a fitted GMM model for given data, one may utilize the marginal distribution 2.45 to generate more samples, for example, for robot movement primitives [102]. Moreover, knowing the Gaussian components, we can utilize this knowledge to generate instances belonging to a specific cluster or even extrapolate the model to improve output accuracy [103].

While the field of GMM applications is vast, one specific property makes it ineffective in terms of the Hierarchical Reinforcement Learning approach tackled in our work. Unfortunately, in GMM, the number of clusters K is a hyperparameter that should be predefined. But what if the task does not specify the number of types of actions used for training? Looking ahead, it is even unclear how many types of skills the model learns from demonstrated data. Defining K means restricts model in its interpretation of the training data. This problem is addressed in our next chapter by introducing a nonparametric Bayesian model - the Dirichlet Process Mixture Model.

2.3.3. Dirichlet Process Mixture Model

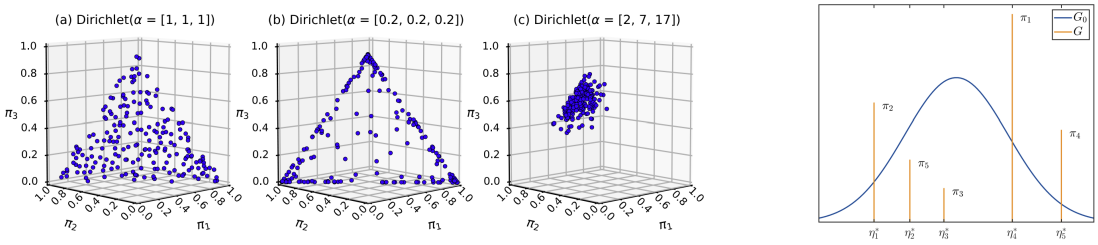
Dirichlet Process Mixture Model (DPMM) are frequently used in nonparametric clustering problems [104], where minimal assumptions about data can be made and the model should show maximum flexibility. This is achieved through a Dirichlet Process prior for infinite mixture distributions [105], allowing the model to determine itself by the data rather than hyperparameters.

From a practical point of view, DPMM with Gaussian-distributed components differs from GMM only by not having a predefined number of clusters, potentially allowing the number of clusters to grow to infinity. However, for the price of flexibility, this makes DPMM very complex mathematically and makes the DPMM posterior distribution intractable [104], which requires additional steps for practical implementation [19]. In this chapter we will sequentially analyze the Dirichlet Process as a prior for DPMM, decompose the transfer to variational inference for the approximation of the posterior and mention a useful extension for stochastic DPMM learning represented by Memoized Online Variational Inference [104].

Dirichlet Process as a prior for DPMM

As mentioned above, adaptability to the data is the main feature of DPMM. In the last section 2.3.2, we have already seen how GMM can generate data from the marginal distribution (2.45) based on intrinsic parameters. Let us see the probability of parameters given data, through the joint posterior distribution obtained with Bayes' Theorem [105]:

$$p(\boldsymbol{\mu}, \boldsymbol{\Sigma}, \mathbf{c}|x) = \frac{p(\mathbf{x}|\boldsymbol{\mu}, \boldsymbol{\Sigma}, \mathbf{c})p(\boldsymbol{\mu}, \boldsymbol{\Sigma}, \mathbf{c})}{p(x)} \propto p(x|\boldsymbol{\mu}, \boldsymbol{\Sigma}, \mathbf{c})p(\boldsymbol{\mu})p(\boldsymbol{\Sigma}) \underbrace{p(\mathbf{c})}_{=\pi} \tag{2.51}$$



(a) (b)

Figure 2.19.: (a) Dirichlet distribution dependency on hyperparameter α using an example of 3-dimensional sampling [105]. (b) Example of a Dirichlet Process for $H = \mathcal{N}(0, \boldsymbol{I})$ [106].

This equation corresponds to the generative GMM model [19] and requires to set $p(\boldsymbol{\mu})$, $p(\boldsymbol{\Sigma})$, $\boldsymbol{\pi}$ to be conjugate priors [105]. Appropriate choices are:

- A Normal Distribution for $\boldsymbol{\mu}$:

$$p(\mu_k) \sim \mathcal{N}(\boldsymbol{\mu}_0, \boldsymbol{\Sigma}_o) \tag{2.52}$$

- An Inverse-Wishart Distribution for $\boldsymbol{\Sigma}$:

$$p(\boldsymbol{\Sigma}_k|\gamma, \beta) \sim \mathcal{W}^{-1}(\gamma, \beta) \tag{2.53}$$

- A Symmetric Dirichlet Distribution for $\boldsymbol{\pi}_k$:

$$p(\boldsymbol{\pi}_1, \boldsymbol{\pi}_2, \ldots \boldsymbol{\pi}_k|\alpha) \sim Dir(\alpha/K, \alpha/K, ...\alpha/K) \tag{2.54}$$

With (2.52), (2.53) and (2.54) we can retrieve the conditional distribution for Gibbs sampling from the respective component [105]. However, if the number of clusters K is not known and can be infinite, we may not sample π_k from a Dirichlet Distribution.

A Dirichlet Process (DP) is a probability distribution over other probability distributions [14]. The Dirichlet process, characterized by a concentration parameter α and a base distribution H, ensures that for any finite partition $\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n$ sampled from a probability distribution $P \sim DP(H, \alpha)$, such that the following holds [105]:

$$(P(\boldsymbol{x}_1), P(\boldsymbol{x}_2), ..., P(\boldsymbol{x}_n)) \sim Dir(\alpha H(\boldsymbol{x}_1), \alpha H(\boldsymbol{x}_2), ..., \alpha H(\boldsymbol{x}_n)) \tag{2.55}$$

Here, the parameter α controls the density of the sampled probability distributions and the base distribution H determines the ground distribution of the Dirichlet Process (see Figure 2.19)

To emphasize the correlation between GMM and DPMM, the marginal probability distribution of a finite mixture model can be described [14] as

$$p(\boldsymbol{x}) = \int p(\boldsymbol{x}|\theta) G(\theta) d\theta \tag{2.56}$$

where θ_k is a parameter defining the mixture component k and G given as

$$G = \sum_{k=1}^{K} \pi_k \delta_{\theta_k} \tag{2.57}$$

In this case, the number of components is finite, and δ_{θ_k} is a Dirac distribution centered at θ, sometimes called an atom [14]. In a Dirichlet Process, the number of components is infinite, resulting in

$$G = \sum_{k=1}^{\infty} \pi_k \delta_{\theta_k} \tag{2.58}$$

In practice, sampling from a Dirichlet Distribution with an infinite number of clusters is impossible [105], which makes the posterior mathematically intractable [104]. In 1973 Thomas S. Ferguson proposed a solution for a Dirichlet Process as a prior for an infinite mixture model [107], in 1985 Aldous introduced the metaphor of the Chinese Restaurant Process to explain the distribution over partitions [14], while in 1994 Sethuraman [108] described the Dirichlet Process as a Stick-Breaking process (see Figure 2.20). These analogies are widely used in describing Bayesian Nonparametric Models [105][19].

Dirichlet Process Analogies

According to [108], if

$$\beta_k \sim Beta(1, \alpha) \tag{2.59}$$

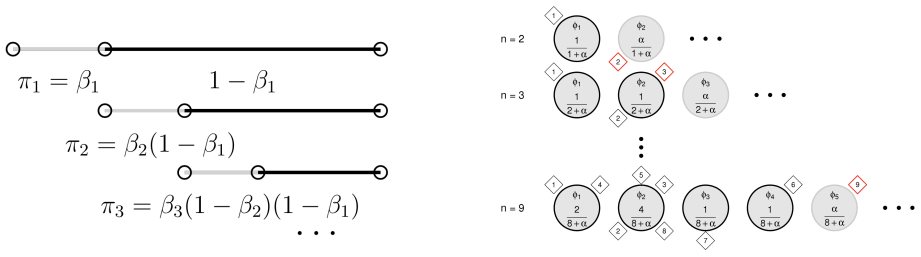
$$\theta_k^* \sim H \tag{2.60}$$

$$\pi_k = \beta_k \prod_{i=1}^{k-1} (1 - \beta_i) \tag{2.61}$$

$$G = \sum_{k=1}^{\infty} \pi_k \delta_{\theta_k^*} \tag{2.62}$$

with $Beta(1, \alpha)$ being Beta-distribution and θ_k^* being parameters of cluster k, then $G \sim DP(\alpha, H)$ [14]. This is called a Stick-Breaking process and demonstrates the intuition behind infinite clusters generated by the Dirichlet Process. A unit length stick is broken into infinite number of segments with length π_k. The proportion of each new segment is determined by $\beta_k \sim Beta(1, \alpha)$ relative to the remaining stick, making the length of each segment equal $\pi_k = \beta_k \prod_{i=1}^{k-1}(1 - \beta_i)$ [108]. The resulting distribution of π is called the Griffiths-Engen-McCloskey (GEM) distribution, which is essentially the infinite component Dirichlet distribution with concentration parameter α being the same for all components [19]. The Stick-Breaking metaphor helps to grasp the concept of infinite clusters and sampling from the Dirichlet Process. But it is still unclear how data gets distributed within those clusters (compare with the responsibilities equation for GMM (2.46)).

The original idea of the Dirichlet process is that the data determines the model itself. Consequently, each instance has a chance to be assigned to an already existing cluster or create a new one, furthermore the order of observations should play no role in the estimation



(a) Stick-Breaking Process metaphor [19].

(b) Chinese Restaurant Process metaphor [105].

Figure 2.20.: (a) Iteratively breaking the proportional to β segments of the stick we obtain $\pi \sim GEM(\alpha)$. (b) Each new sample always has a chance of creating a new cluster - the number of clusters is unbounded.

[105]. Recall that the mixing coefficient is intractable [104], thus for each new sample x_{n+1} the probability to be assigned to one of the already existing clusters c_k is computed as [105]

$$p(c_i = c_k|\alpha) = \frac{n_k}{\alpha + n - 1} \tag{2.63}$$

and the probability of creating a new cluster

$$p(c_i = c_{new}|\alpha) = \frac{\alpha}{\alpha + n - 1} \tag{2.64}$$

where n_k is the number of already assigned to cluster k observations and n is the total number of already assigned instances. Introduced by Aldous and further explained by Gershman & Blei [109], the Chinese Restaurant Process demonstrates the Dirichlet Process cluster allocation with an intuitively understandable metaphor. Imagine a restaurant with an infinite number of tables and friendly customers, all willing to share their dinner with someone. Each newcomer will join the partially occupied table with a probability of $n_k/(\alpha + n - 1)$ and an unoccupied table with a probability of $\alpha/(\alpha + n - 1)$. Notably, component parameters θ_k are not included in the Chinese Restaurant Process analogy but they are part of the Dirichlet Process [109]. This procedure highlights the "rich-gets-richer" nature of clustering, clusters with more instances have a higher probability of assigning a new instance [105].

Dirchlet Process Mixture Model algorithm

In Dirichlet Process Mixture Model, the discrete Dirichlet Process acts as a prior and Mixture Models are often represented using a Gaussian-distributed components [105]. Combined, we have [19]:

$$\theta_k^* \sim H, \quad \pi|\alpha \sim GEM(\alpha), \quad c_i|\pi \sim Cat(\pi), \quad x_j|c_i \sim \mathcal{N}(\theta_{c_i}^*) \tag{2.65}$$

With (2.65) substituted into (2.51) and marginalized over all observations we write the

DPMM joint probability as

$$p(\boldsymbol{x}, \boldsymbol{c}, \boldsymbol{\theta}, \boldsymbol{\beta}) = \prod_{n=1}^{N} \mathcal{N}(x_n|\theta_{c_n}) Cat(c_i|\boldsymbol{\pi}(\boldsymbol{\beta})) \prod_{k=1}^{\infty} Beta(\beta_k|1, \alpha) H(\theta_k|\lambda_0) \tag{2.66}$$

where λ_0 denotes parameters of the Dirichlet Process base distribution H with the assumption that the base distribution also belongs to the exponential family [104]. The goal is to optimize the true posterior $p(\boldsymbol{c}, \boldsymbol{\theta}, \boldsymbol{\beta}|\boldsymbol{x})$, which is impossible analytically due to intractability [19]. Analogous to the EM algorithm, described by the authors in [105], the algorithm inspired by [110] iteratively assigns each instance to a cluster according to the Chinese Restaurant Process described above. This algorithm particularly resembles the GMM E-step (2.46) due to the analogous way of treating cluster responsibilities $p(c_i = k|c_{1...k}, x_i)$ (Figure 2.21, lines (8-11)), as well as the M-step (Figure 2.21, lines (12-15)), where details on means and covariance estimations are omitted (refer to [105]): In simple terms, this algorithm utilizes the Dirichlet Process for initialization and computes the familiar conditional posterior distributions for optimization from section 2.3.2. While this approach is suitable for explaining the transition from GMM to DPMM, variational inference for DPMM has several advantages compared to the conventional method [111][112][104] and will be further explored as the main method for fitting a DPMM.

Variational algorithms approximate an intractable posterior with a factorized variational distribution [111]. A key assumption here is that each latent variable is mutually independent and has its own variational factor (mean-field assumption) [104][19]. With that, the best full-factorized variational distribution q is given by

$$q(\boldsymbol{c}, \beta, \theta) = \prod_{n=1}^{N} q(c_n|\hat{r}_n) \prod_{k=1}^{K} q(\beta_k|\hat{\alpha}_1, \hat{\alpha}_0) q(\theta_k, \hat{\lambda}_k), \tag{2.67}$$

$$q(c_n) = Cat(c_n|\hat{r}_{n1}, ...\hat{r}_{nK}), \quad q(\beta_k) = Beta(\beta_k|\hat{\alpha}_1, \hat{\alpha}_0), \quad q(\theta_k) = H(\theta_k|\hat{\lambda}_k) \tag{2.68}$$

where the parameters denoted by $\hat{*}$ are variational factors of q and a generalized responsibility r_n is used for cluster assignments and entropy computation[104]. This truncated distribution forces the data to be explained only by K clusters despite having Dirichlet Process prior by forcing all atom weights beyond K to be zero

$$G = \begin{cases} G = \sum_{k=1}^{K} \pi_k \delta_{\theta_k} & \text{for } k \in [1, ..., K] \\ 0 & \text{for } k \in [K+1, ..., \infty) \end{cases} \tag{2.69}$$

The best approximation of $p(\boldsymbol{c}, \boldsymbol{\theta}, \boldsymbol{\beta}|\boldsymbol{x})$ is then computed with the help of the ELBO we already mentioned in section 2.1.2. The resulting optimization problem can be expressed as [19]:

$$ELBO(q) = \mathbb{E}[\log p(\boldsymbol{x}|\boldsymbol{c}, \boldsymbol{\theta}, \boldsymbol{\beta})] - \mathbb{KL}(q(\boldsymbol{c}, \boldsymbol{\theta}, \boldsymbol{\beta})||p(\boldsymbol{c}, \boldsymbol{\theta}, \boldsymbol{\beta})) \tag{2.70}$$

The direct optimization of (2.70) is conducted on the full dataset [111], but the work of Hughes and Sudderth [104] introduced a computationally stable stochastic online and memoized online variational inferences utilizing sufficient statistics for dynamic optimization.

Algorithm DPMM

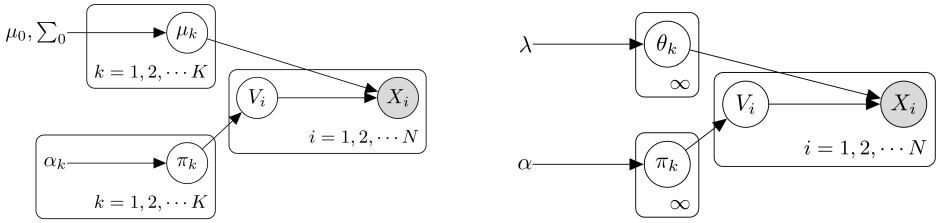
input : $\alpha, \mu_0, \sigma_0^2, \sigma_y^2$ (e.g., $\alpha = 0.01$, $\mu_0 = 0, \sigma_0^2 = I_d \cdot 3^2, \sigma_y^2 = I_d \cdot 1^2$, where I is an identity matrix of $d = 2$), $\tau_0 = 1/\sigma_0^2$, $\tau_y = 1/\sigma_y^2$.

output: a MCMC chain of simulated values of $\boldsymbol{c}$.

Given the concentration parameter α and the state of the Markov chain $\{\mu_k^{(t-1)}, \tau_k^{(t-1)}\}, \boldsymbol{c}^{(t-1)}$, sample a new set of $\{\mu_k^{(t)}, \tau_k^{(t)}\}$ and $\boldsymbol{c}^{(t)}$:

for $t \leftarrow 1$ **to** *maxiter* **do**
  for $i \leftarrow 1$ **to** n **do**
    Remove y_i from cluster c_i because we are going to draw a new sample of c_i for y_i.
    If the previous step causes a cluster to becomes empty, then remove the cluster, its corresponding parameters, and rearrange the order of the clusters into contiguous $1, 2, \ldots, K$.
    Draw $c_i | c_{-i}, y$ from:
    for $k \leftarrow 1$ **to** $K + 1$ **do**
      Calculate the probability of $c_i = k$ using
      $p(c_i = k | c_{-i}, y_i) \propto$
      $$\frac{n_{-i,k}}{n - 1 + \alpha} \mathcal{N}\left(\tilde{y}_i; \frac{\bar{y}_k n_k \tau_k + \mu_0 \tau_0}{n_k \tau_k + \tau_0}, \frac{1}{n_k \tau_k + \tau_0} + \sigma_y^2\right).$$
      Calculate the probability of $c_i = k + 1$ using
      $p(c_i \neq c_k \; \forall j \neq i | c_i, y_i) \propto$
      $$\frac{\alpha}{n - 1 + \alpha} \mathcal{N}(\tilde{y}_i; \mu_0, \sigma_0^2 + \sigma_y^2).$$
      if $c_i = k$ for some $j \neq i$ **then**
        Update $\bar{y}_k, n_k, \tau_k$ according to $c_i = k$.
      else
        $c_i = K + 1$, a new cluster has been created. Append this new cluster to the vector of non-empty clusters.
    end
  end
  Set $\boldsymbol{c}^{(t)} = c_i$.
end
return $\boldsymbol{c}$

Figure 2.21.: Presented in [105] DPMM algorithm. For simplicity, precision $\tau = \Sigma^{-1}$ is used for cluster responsibilities, means and covariance estimation.



(a) Generative graphic model of GMM. (b) Generative graphic model of DPMM.

Figure 2.22.: Comparison of GMM and DPMM [19].

Therefore these details will be discussed in the next section, but for now, we summarize the DPMM framework.

The Dirichlet Process Mixture Model is a powerful tool for clustering that uses a Dirichlet Process prior to operate on an infinite number of clusters [105]. A comparison with a finite mixture model like GMM requires a probabilistic formulation (see Figure 2.22), where the similarities are clearly visible.

Despite this, due to its nature the posterior distribution of a DPMM is mathematically intractable and should be approached from a different angle. We have considered a straightforward approach (2.21) [105] and a variational approach (2.70) [19][104][111], the second of which allows us to perform a stochastic approach and the dynamic merging of clusters, which we will pay attention to now.

Memoized Online Variational Bayes

In 2013 Hughes and Sudderth in their work [104] introduced a Memoized Online Variational Bayes (MemoVB) inference algorithm generalizing the EM [93] for nonparametric Bayesian models. This approach processes batches of the full dataset represented by expected sufficient statistics $s_k(\boldsymbol{x})$ and expected mass $\hat{N}_k$ for each component. Using sufficient statistics is important because operating with batches with similar sufficient statistics will yield similar inferences about intrinsic parameters [113]. This assumption helps authors to derive the memoized algorithm for DPMM ELBO [104][19]:

$$\begin{aligned} ELBO(q) = \sum_{k=1}^{K} [\mathbb{E}_q[\theta_k]^T s_k(x) - \hat{N}_k[\alpha(\theta_k)] + \hat{N}_k[\log \pi_k(\beta)] - \sum_{n=1}^{N} \hat{r}_{nk} \log \hat{r}_{nk} \\ + \mathbb{E}_q[\log \frac{Beta(\beta_k|1,\alpha)}{q(\beta_k|\hat{\alpha}_{k1}, \hat{\alpha}_{k0})}] + \mathbb{E}_q[\log \frac{H(\theta_k|\lambda)}{q(\theta_k|\hat{\lambda}_k)}]] \end{aligned} \tag{2.71}$$

This update is executed iteratively for batches B_t, where first local parameters q_{c_n} for $n \in B_t$ are updated, then global parameters q_{β_k} and q_{θ_k} are estimated with memoized sufficient and full-dataset statistics $S_k^t = [\hat{N}_k(B_t), s_k(B_t)]$ and $S_k^0 = [\hat{N}_k, s_k(x)]$ respectively [104]:

$$\hat{r}_{nk} = \frac{\tilde{r}_{nk}}{\sum_{l=1}^{K} \hat{r}_{nl}}, \quad where \quad \tilde{r}_{nk} = exp(\mathbb{E}_q[\log \pi_k(\beta)] + \mathbb{E}_q[\log p(x_n|\theta_k)]) \tag{2.72}$$

$$\hat{N}_k = \sum_{n=1}^{N} \hat{r}_{nk}, \quad \alpha_{k1} = \alpha_1 + \hat{N}_k, \quad \alpha_{k0} = \alpha_0 + \sum_{l=k+1}^{N} \hat{N}_l,$$
$$s_k(x) = \sum_{n=1}^{N} \hat{r}_{nk} t(x_n), \quad \hat{\lambda}_k = \lambda_0 + s_k(x) \tag{2.73}$$

To comprehend this update rule, we could draw a parallel and call equation 2.72 an Expectation step since it updates parameters based on the observations (with respect to memoized Batches B_t and full statistics) and 2.73 - a Maximization step since it tries to fit the model with respect to the parameters estimated in the E-step (see section 2.1.1 for detailed discussion). However, strictly it is not the case due to the nonparametric nature of DPMM [104] and this statement only serves to acquire an intuition behind this process.

This update accelerates learning on large datasets while still being stable as shown in [104]. However, authors also introduced another trick for better learning. Recall that as DPMM allocates samples to the clusters, the number of clusters monotonically increases. This may result in local optima [104], potentially yielding suboptimal solutions. To avoid this, Hughes and Sudderth introduced Birth and Merge moves to escape local optima (see Figure 2.23)

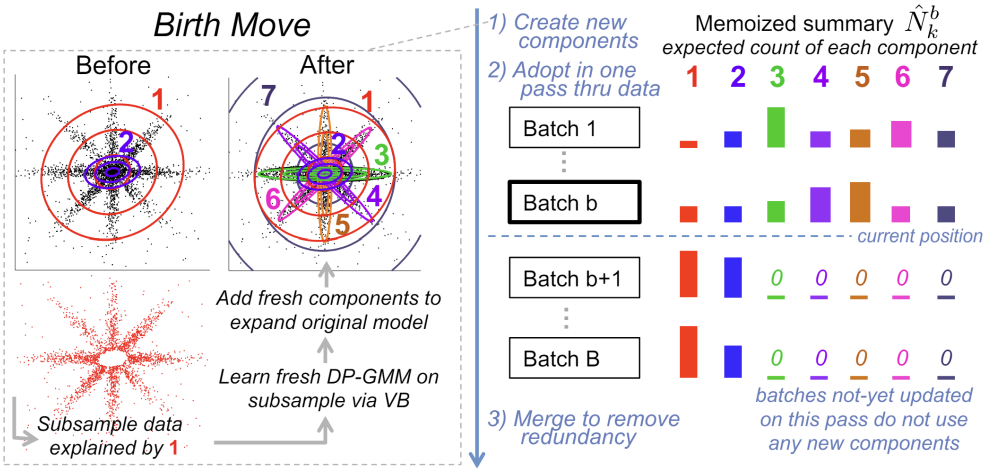


Figure 2.23.: Presented in [104] birth-merge moves (MO-BM) to escape local optima. New components are created, evaluated and accepted or rejected during the Birth phase and later merged to avoid redundancy.

The birth-merge moves (MO-BM) [104] could be described metaphorically as a the beating of a heart:

- **Birth moves phase** consists of creating new components to ensure stable DPMM convergence. The process includes a *collection step*, where subsamples x' are collected from component k', and then new components are created at the *creation step* where DPMM with a limited number of clusters K' is fitted to x'. Finally, at the *adoption step* experimental parameter update on the new model with $K + K'$ clusters determine if the

new component will be adopted or pruned based on an estimation of ELBO [104]. As a result, DPMM can further grow in size.

- **Merge moves phase** ensures that the DPMM size will not diverge, which will result in a computational load. During this phase at the *component selection step*, the algorithm chooses a random component k_a and a suitable candidate for merging k_b with marginal likelihood $p(k_a|k_b) \propto \frac{M(S_{k_a}+S_{k_b})}{M(S_{k_a})M(S_{k_b})}$ with $M(S_k) = exp(\alpha_0(\lambda_0 + s_k(x)))$. Next, at the *merging candidate step* q' is formed with new merged components, and lastly at *acceptance step*, an exact ELBO evaluation determines if this merge is successful [104]. This results in a shrinking number of DPMM clusters.

Hughes and Sudderth empirically proved the efficiency of the augmented DPMM optimization algorithm [104]. Their work has became a useful and stable tool for further research [19]. In our work we will also incorporate the MemoVB stochastic approach for DPMM clustering.

3. Related Work

Having built up our knowledge foundation, we turn to the related work vital for our algorithm. In this chapter we will sequentially decompose two approaches we tailor together in one functional algorithm. In first section 3.1 we analyze the Skill Prior guided reinforcement learning framework proposed in [18]. Second section 3.2 concentrates on novel nonparametric Bayesian latent space clustering model [19]. We use this chapter to discuss these approaches and highlight their relevance to our work.

3.1. Accelerating Reinforcement Learning with Learned Skill Priors

The Reinforcement Learning agents learning by interacting with an environment by estimating action policy π_θ (see section 2.2 for more details). We have also already discussed in section 2.2.4 the possibility of elevating the policy on high level by introducing skills. But in reality without external guidance even high-level policy RL may be inefficient due to nature of the trial-and-error. Other algorithms [114][115] have learned behaviour priors to guide offline RL towards goal.

Authors of "Accelerating Reinforcement Learning with Learned Skill Priors" [18] combine two approaches to form a hierarchical RL algorithm with skill-prior guidance SPiRL. The proposed algorithm can be divided into two steps. First, the unlabelled action-state sequence is used for Skills embedding and Skill Prior derivation. Second, the learned latent Skill representations are used as a action space for Hierarchical Reinforcement Learning with guidance of the Skill Prior. Resulting model outperforms unmodified RL-algorithms such as Soft Actor Critic seen in section 2.2.3 which was the initial intention of this work.

3.1.1. Learning Prior over Skills

In the Figure 3.1 the main architecture of the algorithm is provided. Latent Skill embedding is obtained via skill encoder from the unstructured sequence of state-actions pairs. The Skill Prior maps the beginning environment state over latent skill embedding, guiding towards the most promising skill to explore in the RL-step. Authors utilize the raw unstructured long and complex manipulation sequences from D4RL maze environment [116], block stacking and uses a simulated kitchen environment based on Gupta et al. [117].

In our work we will perform experiments only on kitchen environment, therefore we will focuse on this application with emphasis on kitchen-mixed-v0 and kitchen-complete-v0 datasets [116]. Nevertheless, authors provided several experiments with maze navigation and block stacking - our work may also be extended to these benchmarks. SPiRL framework consists of VAE discussed in section 2.1.2 encoding fixed horizon demonstrated actions

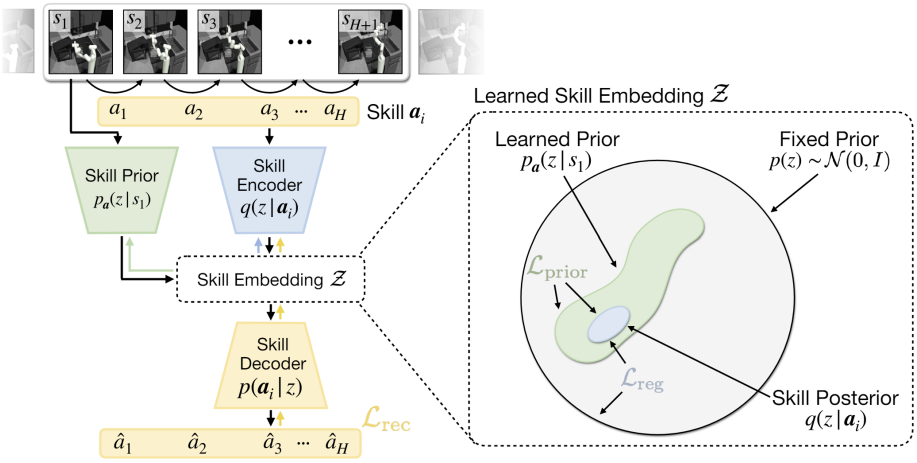


Figure 3.1.: SPiRL Deep model Architecture as proposed in the [18].

$a_i = \{a_t^i, a_{t+1}^i, ..., a_{t+H}^i\}$ into latent space with additional Skill Prior Network learning the same latent representation given s_t.

The intuition behind this approach is that Skill Prior encoding $p_a(z|s_t)$ provides a strong belief of a promising skill to execute based on the current state, while trained decoder $p(a_i'|z)$ reconstructs the actions from the Skill Embedding space.

In chapter 4 we will dive deeply into underlying processes of Skill Encoding and described framework. For now it is sufficient to say that trained at this step model lays a strong foundation for skill-based hierarchical reinforcement learning.

3.1.2. Hierarchical Reinforcement Learning with Skill Priors

In section 2.2.4 we analysed the concepts behind high-level policies in reinforcement Learning. SPiRL [18] utilizes this concept by learning a policy $\pi_\theta(z|s_t)$ over Skill Latent Space and decodes them with pre-trained at the previous step decoder $p(a_i'|z)$. To leverage the advantage gained by Skill Prior, authors modify the policy-based Soft Actor-Critic (section 2.2.3) by replacing the entropy term with the negated KL divergence between the policy and the learned Skill Prior:

$$J(\theta) = \mathbb{E}_\pi[\sum_{t=1}^{T} \tilde{r}(s_t, z_t) - \alpha D_{KL}(\pi(z_t|s_t), p_a(z_t|s_t))] \tag{3.1}$$

This small change is justified, since SAC entropy augmentation [27] is equivalent to the negated KL divergence between the policy and uniform action prior [18]. Modified SAC for Skill-Prior guided RL is presented in the Figure 3.2.

It is worth mentioning second version authors provided in their implementation. As stated in paper [18] resulted for kitchen environment were most likely provided by the SPiRL model with closed-loop low-level skill decoder implemented as extension for described above

approach in March 2021 (according to official github repository). This version significantly improved performance by adding a bypass for state vector directly to the decoder. This way, closed-loop SPiRL model learns to decode from a concatenated vector obtained from the union of current state observations $\boldsymbol{s}_i$ and encoded from corresponding actions latent skill samples $z \sim q(z|\boldsymbol{a}_i)$. Since, practically, this model processes only one latent skill z for a fixed horizon, intermediate observations result from the decision π_θ made by actor network and suffice the requirements of VAE. In our work we inherit this structure for our research and will address to it as a closed-loop model (see Figures 4.1 and 4.2 in corresponding sections).

Algorithm SPiRL: Skill-Prior RL

Inputs: H-step reward function $\tilde{r}(s_t, z_t)$, discount γ, target divergence δ, learning rates $\lambda_\pi, \lambda_Q, \lambda_\alpha$, target update rate τ.
Initialize replay buffer $\mathcal{D}$, high-level policy $\pi_\theta(z_t|s_t)$, critic $Q_\phi(s_t, z_t)$, target network $Q_{\bar{\phi}}(s_t, z_t)$
for each iteration **do**
  for every H environment steps **do**
    $z_t \sim \pi(z_t|s_t)$ ▷ sample skill from policy
    $s_{t'} \sim p(s_{t+H}|s_t, z_t)$ ▷ execute skill in environment
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{s_t, z_t, \tilde{r}(s_t, z_t), s_{t'}\}$ ▷ store transition in replay buffer
  for each gradient step **do**
    $\bar{Q} = \tilde{r}(s_t, z_t) + \gamma\big[Q_{\bar{\phi}}(s_{t'}, \pi_\theta(z_{t'}|s_{t'})) - \alpha D_{\mathrm{KL}}\big(\pi_\theta(z_{t'}|s_{t'}), p_{\boldsymbol{a}}(z_{t'}|s_{t'})\big)\big]$ ▷ compute Q-target
    $\theta \leftarrow \theta - \lambda_\pi \nabla_\theta\big[Q_\phi(s_t, \pi_\theta(z_t|s_t)) - \alpha D_{\mathrm{KL}}(\pi_\theta(z_t|s_t), p_{\boldsymbol{a}}(z_t|s_t))\big]$ ▷ update policy weights
    $\phi \leftarrow \phi - \lambda_Q \nabla_\phi\big[\frac{1}{2}\big(Q_\phi(s_t, z_t) - \bar{Q}\big)^2\big]$ ▷ update critic weights
    $\alpha \leftarrow \alpha - \lambda_\alpha \nabla_\alpha\big[\alpha \cdot (D_{\mathrm{KL}}(\pi_\theta(z_t|s_t), p_{\boldsymbol{a}}(z_t|s_t)) - \delta)\big]$ ▷ update alpha
    $\bar{\phi} \leftarrow \tau\phi + (1-\tau)\bar{\phi}$ ▷ update target network weights
return trained policy $\pi_\theta(z_t|s_t)$

Figure 3.2.: SPiRL SAC modification for leveraging Skill Prior knowledge [18]. Differences to the original algorithm are marked red.

3.1.3. Relevance for our Work

In [18] authors have proved that Skill Prior indeed accelerates the Reinforcement Learning for sparse rewards which the most robotic applications agents are operating with. This fact makes this algorithm a promising candidate for further modification and main framework of our purposes.

We are handling the proposed in [18] architecture as the main framework due to two main reasons:

1. Reliable and suitable Skill encoding framework for our purposes.
2. Build-in Hierarchical Reinforcement Learning Closed-Loop architecture for simulated or real-world agent.

3.2. DIVA: A Dirichlet Process Based Incremental Deep Clustering Algorithm via Variational Auto-Encoder

Deep clustering is a research field that applies deep learning architectures for clustering highly non-linear data that conventional algorithms struggle with [118]. One approach is VAE-based clustering where the data is assumed to be the same form and structure, thus it can be represented in latent space of Variational Autoencoder. On one hand, we have discussed the structure of VAE in section 2.1.2 and seen the probabilistic nature of latent space, allowing Bayesian clustering [119]. On the other hand we've discussed in section 2.3.3 that in some cases models with fixed number of components may not adapt to dynamic and complex features represented in the dataset.

Authors of "DIVA: A Dirichlet Process Based Incremental Deep Clustering Algorithm via Variational Auto-Encoder" [19] framework combine the concept of model-based deep clustering with a Variational Autoencoder and a Dirichlet Process Mixture Model. Proposed algorithm incorporates a DPMM-based prior into the architecture of a Variational Autoencoder, allowing it to classify complex datasets in a nonparametric manner that outperforms many state-of-the-art approaches [19]. Essentially this approach is a nonparametric upgrade of GMVAE mentioned in section 2.1.2 with a potentially infinite number of clusters and flexible assignment.

In Figure 3.3 the main architecture of the algorithm can be seen. Given an unlabelled dataset, the DIVA algorithm simultaneously learns the latent distribution of projected samples, the number of clusters, the parameters of an infinite mixture of Gaussians and the correct reconstruction [19].

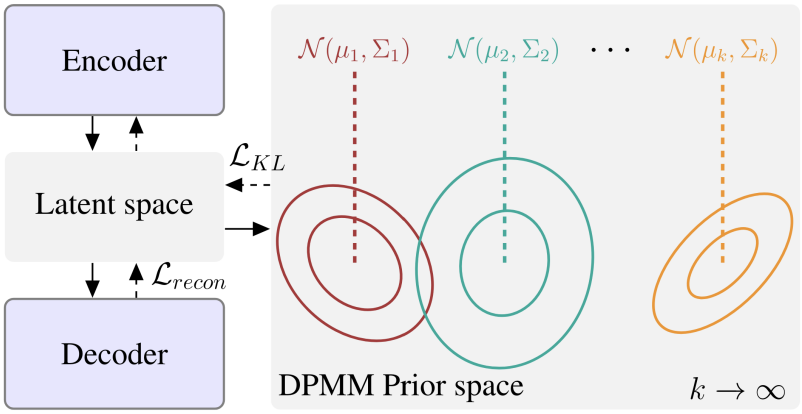


Figure 3.3.: Architecture of DIVA [19]. Variational Autoencoder projects inputs onto a Gaussian-distributed latent space that is later fitted to the DPMM model with an infinite amount of clusters.

The algorithm can be divided into two steps - VAE learning with DPMM prior and DPMM learning based on the allocated data [19].

3.2.1. DPMM-prior based VAE update

Assuming we have a DPMM as a prior, the question is how to fit the latent distribution to the existing model. Authors keep the reconstruction loss $\mathcal{L}_{rec}$ from original the VAE and fit the encoder output distribution to the corresponding mixture component via stochastic assignment $v_i = k$. In this way, for a sampled latent vector $z_i = \mu(\boldsymbol{x}, \boldsymbol{\phi}) + \sigma(\boldsymbol{x}, \boldsymbol{\phi}) \odot \boldsymbol{\epsilon}$ where $\boldsymbol{\phi}$ are encoder parameters and $\boldsymbol{\epsilon} \sim \mathcal{N}(0, \boldsymbol{I})$ is Gaussian noise, corresponding cluster assignments p_{ik} are computed through the existing DPMM model and the resulting Loss can be expressed for each sample i assigned to cluster k as [19]

$$\mathcal{L}_{KL_{ik}} = \frac{1}{2}[\log \frac{|\Sigma_k|}{|\Sigma(x_i, \boldsymbol{\phi})|} - D + tr\{\Sigma_k^{-1}\Sigma(x_i, \boldsymbol{\phi})\} + (\mu_k - \mu(x_i, \boldsymbol{\phi}))^T\Sigma_k^{-1}(\mu_k - \mu(x_i, \boldsymbol{\phi}))] \quad (3.2)$$

Avoiding wrong assignments, authors of [19] introduced soft assignments, where responsibility p_{ik} (similar to the idea we have seen in section 2.3.2) is computed for each sample i and cluster k, resulting in a combined KL divergence

$$\mathcal{L}_{KL_i} = \sum_{k=1}^{K} p_{ik}\mathcal{L}_{KL_{ik}} \quad (3.3)$$

Notably, at the first epoch an untrained VAE should fit to an untrained DPMM, therefore at the first epoch the DPMM is replaced with simple Gaussian distribution [19].

3.2.2. Bayesian nonparametric DPMM for VAE

VAE training is executed batch-wise, while fitting the DPMM is executed at the very end of each epoch. As we have seen in section 2.3.3 that Dirichlet Process Mixture Model requires data to be processed. DIVA fits the DPMM to the obtained latent samples z_i, making this model the prior distribution for VAE [19]

Authors also utilize the method developed by Hughes and Sudderth [104], which we sufficiently described in section 2.3.3. It relates to stochastic variational posterior approximation update with MemoVB and birth-merge moves.

3.2.3. Relevance for our Work

In [19] authors have proved the superiority of the versatile DPMM Prior in unsupervised clustering on MNIST, Fashion-MNIST and Reuters10k dataset compared to modern supervised classification algorithms [19]. This adaptability and promising clustering ability make this algorithm an interesting candidate for the VAE augmentation module in our work. We will utilize the DPMM architecture for VAE proposed in [19] due to the following reasons:

1. Versatile and nonparametric clustering technique for latent features.
2. An opportunity for analysis of self-clustered skill interpretations.

4. Methodology

In this section we present Generalized Adaptive Skill Prior Meta Reinforcement Learning (GASP Meta-RL), a skill prior guided meta-reinforcement learning framework with integrated deep nonparametric Bayesian clustering of latent skills. Our work will be closely aligned with recent research achievements we analyzed in chapter 3, as well as utilize most of the components discussed in chapter 2. Recall that our work focuses on the development and evaluation of a Meta-Reinforcement Learning agent trained on unstructured demonstrations to encode and simultaneously cluster latent skills. Essentially, our approach will be divided into two main parts. In the first part, we will discuss the process of learning a DPMM-distributed skill embedding space, while in the second part, we will deconstruct the hierarchical reinforcement learning based on the acquired knowledge.

4.1. Generalized Adaptive Skill Prior

Figure 4.1 shows the detailed architecture of the generalized skill prior learning framework is provided.

Our Generalized Skill Prior model will simultaneously learn:

- DPMM - distributed **Skill Embedding space** $q_\phi(z|\boldsymbol{a}_i)$.
- Accordingly distributed **Skill Prior** for later Hierarchical RL - guidance $p_{\boldsymbol{a},\psi}(z|s_1)$.
- Action **Reconstruction** from the latent space $q_\theta(\hat{\boldsymbol{a}}_i|z)$.

Below, we will decompose the data and loss backpropagation flow for a full understanding of the underlying processes.

4.1.1. Inference and Backpropagation

At the stage of Generalized Skill Prior learning we utilize the D4RL kitchen library [116] containing recorded demonstrations of robotic manipulations in the Franka Kitchen environment (more on the datasets and simulation frameworks will be covered in sections 5.1 and 5.2). Similar to [18] we are parsing the state-action sequences into **H-length horizon pairs** $\{(s_1, a_1), (s_2, a_2), \dots (s_H, a_H)\}_i$ and forwarding the corresponding actions $\boldsymbol{a}_i$ to the **skill encoder network** $q_\phi(z|\boldsymbol{a}_i)$ parameterized by ϕ and the first state s_1 of the sequence i to the **skill prior encoder** $p_{\boldsymbol{a},\psi}(z|s_1)$ parameterized by ψ (intuition behind this is covered in section 3.1). Since we are working with a sequence of consecutive actions, both networks are equipped with **LSTM layers** for capturing time dependencies as discussed in section 2.1.3. Both encoders provide Gaussian distributions $q_\phi(z|\boldsymbol{a}_i) = \mathcal{N}(\boldsymbol{\mu}_\phi, \boldsymbol{\Sigma}_\phi)$ and $p_{\boldsymbol{a},\psi}(z|s_1) = \mathcal{N}(\boldsymbol{\mu}_\psi, \boldsymbol{\Sigma}_\psi)$.

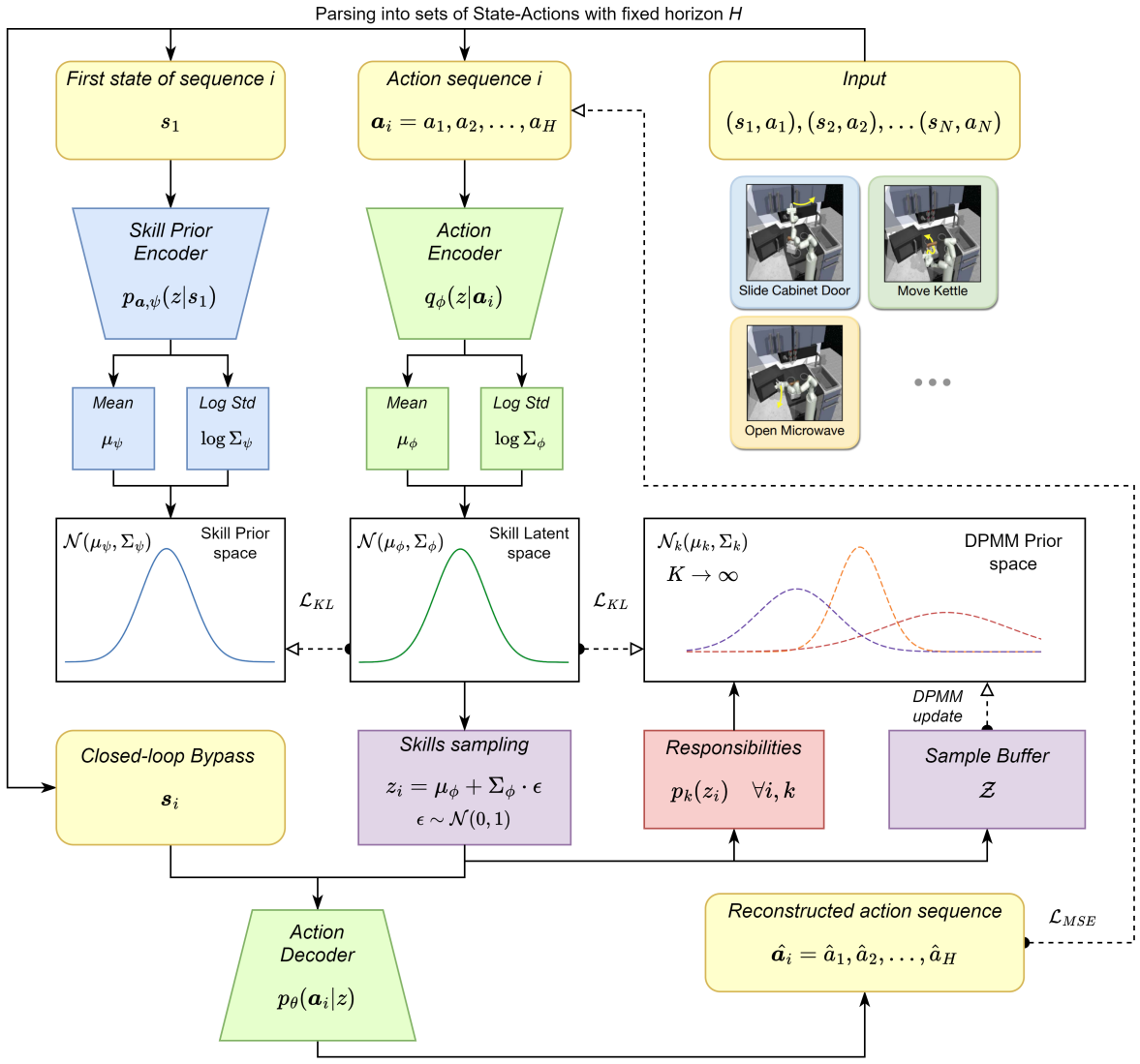


Figure 4.1.: Generalized Adaptive SKill Prior model. The model architecture follows the implementation of the deep latent model from [18] with several augmentations including DPMM. Solid lines represent the data flow, dashed lines represent loss components (see Backpropagation in section 2.1).

For the framework output, the **decoder** ANN $q_\theta(\hat{\boldsymbol{a}}_i|z)$, as a mirrored version of the encoder, reconstructs the sequence of actions parameterized by θ.

4.1.2. Dirichlet Process Mixture Model Prior-Regularized learning

As discussed in section 2.1.2, we will utilize the Gaussian output of the encoder to create the latent space distribution, but analogous to the [19] we will utilize a **nonparametric Bayesian DPMM as a prior distribution**. The probabilistic nature of encoder allows us to utilize Evidence Lower Bound estimation (recall ELBO for probabilistic models in sections 2.1.2 and 2.3.3). Similar to [18], the VAE architecture is augmented with an additional Skill Prior network, but in contrast to the paper, our approach estimates KL divergence $\mathcal{L}_{KL}(q_\phi(z|\boldsymbol{a}_i)||p_\psi(z|s_1))$ instead of NLL-Loss $NLL(p(z|s_1), q(z|\boldsymbol{a}_i))$, ensuring mode-coverage (see Figure 2.4). Notably, for the Kullback-Leibler divergence between the Action Encoder and Skill Prior Encoder we freeze the weights ϕ of the Encoder to ensure Skill Prior training [18]. The reconstruction error is computed as MSE-loss $\mathcal{L}_{MSE} = \sum_{i=1}^{H}(a_i - \hat{a}_i)^2$. We leave this term unweighted since both other terms can be weighted accordingly.

The Dirichlet Process Mixture Model model discussed in section 2.3.3 serves simultaneously as a prior and as a clustering model for latent skills. As mentioned in section 3.2, the first epoch is considered to be a warm-up where the model is not yet trained to create a nonparametric Bayesian DPMM. Instead, during the first epoch we use the **unmodified VAE ELBO estimation** as in (2.7). Therefore, the total training error is backpropagated non-stochastically due to the reparametrization trick described in section 2.1.2, and the total loss of the first epoch can be written as

$$\mathcal{L}_{tot(0)}(\phi, \psi, \theta) = \underbrace{\mathcal{L}_{MSE}(\boldsymbol{a}, \hat{\boldsymbol{a}}_i)}_{\text{Reconstruction Error}} + \beta_1 \underbrace{\mathcal{L}_{KL}(q_\phi(z|\boldsymbol{a}_i)||p_\psi(z|s_1))}_{\text{KL(Encoder}||\text{Skill Prior)}} + \beta_2 \underbrace{\mathcal{L}_{KL}(q_\phi(z|\boldsymbol{a}_i)||\mathcal{N}(0, \boldsymbol{I}))}_{\text{KL(Encoder}||\text{Gaussian Fixed Prior)}} \tag{4.1}$$

where β_i are weighting factors [18]. For clarity, we present the equation (4.1) in a probabilistic form, but rewrite the algorithmic approximation for the DPMM-Prior Loss. As the warm-up epoch concludes, our goal is to enhance the prior space by substituting it with a **DPMM**. At this step we approximate our Dirichlet Process mixture model joint probability p

$$\begin{aligned} p(\boldsymbol{z}, \boldsymbol{c}, \beta, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = &\prod_{i=1}^{N} \mathcal{N}(z_i|\mu_{c_n}, \Sigma_{c_n}) Cat(c_n|\boldsymbol{\pi}(\beta)) \prod_{k=1}^{\infty} \mathcal{B}(\beta_k|1, \alpha) \\ &\prod_{k=1}^{\infty} \mathcal{N}(\mu_k|\mu_0(\lambda\Sigma_k)^{(-1)}) \mathcal{W}(\Sigma_k|\boldsymbol{W}, \nu) \end{aligned} \tag{4.2}$$

by a full-factorized variational distribution q

$$q(\boldsymbol{c}, \beta, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \prod_{i=1}^{N} Cat(c_n|\hat{r}_{n_1}, ..., \hat{r}_{n_k}) \prod_{k=1}^{K} \mathcal{B}(\beta_k|\hat{\alpha}_{k_0}, \hat{\alpha}_{k_1}) \prod_{k=1}^{K} \mathcal{N}(\mu_k|\hat{\mu}_0(\hat{\lambda}\Sigma_k)^{(-1)}) \mathcal{W}(\Sigma_k|\hat{W}, \hat{\nu}) \tag{4.3}$$

To achieve this, we collect all latent vector samples used during the training phase of this epoch from the buffer $\mathcal{Z}$ [19] and utilize the ELBO maximization as discussed in section 2.3.3

in equations (2.71) (2.72) and (2.73) to optimize the DPMM parameters $\hat{\mu}_0, \hat{\lambda}, \hat{\mathcal{W}}, \hat{r}_{n_1:n_k}, \hat{\alpha}_{k_1}, \hat{\alpha}_{k_0}$ and $\hat{\nu}$. Each sample serves as an instance for the nonparametric Bayesian model. We also use batch training and birth-merge moves explained in section 2.3.3 for stable training and to avoid local optima [104].

From the second epoch on, our model will have an **initialized DPMM prior for latent space** reference with stored component parameters $\{(\mu_1, \Sigma_1), (\mu_2, \Sigma_2), ...(\mu_K, \Sigma_K)\}$ for the current cluster size learned at the end of the previous epoch $K \in [1, 2, ..., \infty)$. We freeze the parameters of the DPMM during the training phase for the VAE update kept from (4.3). The computation of KL divergence is performed with soft clustering responsibilities p_{ik} for each sample i and cluster k according to the weighted allocation $\mathcal{L}_{KL_i} = \sum_{k=1}^{K} p_{ik} \mathcal{L}_{KL_{ik}}$ as mentioned in section 3.2. The resulting objective function adjusted for the DPMM-distributed prior is

$$
\begin{aligned}
&\mathcal{L}_{tot}(\phi, \psi, \theta) = \sum_{j=1}^{H} (a_j - \hat{a}_j)^2 \\
&+ \beta_1 [\log \frac{|\Sigma_\psi(s_1)|}{|\Sigma_\phi(\boldsymbol{a}_i)|} + tr\{\Sigma_\psi(s_1)^{-1} \Sigma_\phi(\boldsymbol{a}_i)\} + (\mu_\psi(s_1) - \mu_\phi(\boldsymbol{a}_i))^T \Sigma_\psi(s_1)^{-1} (\mu_\psi(s_1) - \mu_\phi(\boldsymbol{a}_i))] \\
&+ \beta_2 \sum_{k=1}^{K} p_{ik} [\log \frac{|\Sigma_k|}{|\Sigma_\phi(\boldsymbol{a}_i)|} + tr\{\Sigma_k^{-1} \Sigma_\phi(\boldsymbol{a}_i)\} + (\mu_k - \mu_\phi(\boldsymbol{a}_i))^T \Sigma_k^{-1} (\mu_k - \mu_\phi(\boldsymbol{a}_i))]
\end{aligned}
\tag{4.4}
$$

where ϕ are parameters of the Skills encoder $q_\phi(z|\boldsymbol{a}_i)$ and $\mu_\phi(\boldsymbol{a}_i)$ and $\Sigma_\phi(\boldsymbol{a}_i)$ are the outputs of the Skill Encoder Network with input $\boldsymbol{a}_i$ given current parameters ϕ. We also omit the dimensionality term D resulting from KL divergence between two multivariate Gaussian distributions as this term is a hyperparameter that determines the latent space dimensionality and remains constant.

Differing from the original paper we cannot perform validation in a conventional manner since our input is an unlabelled sequence which does not have target values $\boldsymbol{y}_n$. The original VAE performance is evaluated by sampling from the fixed Gaussian prior $p(z) \sim \mathcal{N}(0, \boldsymbol{I})$, but due to the intractability of the posterior distribution of the DPMM-prior discussed in section 2.3.3 we cannot sample from a Dirichlet Process-based mixture model. Therefore we perform a hard assignment $\arg\max(p_{ik})$ of latent skill z_i to the component k and sample from this component using the known parameters estimated at the end of the previous epoch $p_k(z) \sim \mathcal{N}(\mu_k, \Sigma_k)$. Algorithm 1 summarizes the Generalized Skill Prior learning, we mark red all changes we introduced to original model.

Algorithm 1 Generalized Adaptive Skill Prior Learning

Require: Dataset $\mathcal{D}$, Batch size $\mathcal{B}$, DPMM parameters $\mu_0, \lambda, \mathcal{W}, \nu, \alpha$, Samples Buffer $\mathcal{Z}$, Skill Encoder (Q) parameters ϕ, Skill Prior Ensemble (P) parameters ψ, Decoder (D) parameters θ.

Initialize ϕ, ψ, θ
for each epoch **do**
  for each iteration **do**
    $\mu_\phi, \Sigma_\phi \leftarrow Q(\boldsymbol{a}_i)$ ▷ Forward Pass Skill Encoder
    $\mu_\psi, \Sigma_\psi \leftarrow P(s_1)$ ▷ Forward Pass Skill Prior Ensemble
    $z_i = \mu_\phi + \Sigma_\phi \cdot \epsilon$ ▷ Skill Embedding Sampling
    Append z_i to $\mathcal{Z}$ ▷ Saving sample to the buffer for DPMM
    $\hat{\boldsymbol{a}}_i \leftarrow D(z_i)$ ▷ Decoding Sampled Skill
    if 1st epoch **then**
      Compute $\mathcal{L}_{tot}(\phi, \psi, \theta)$ with (4.1) ▷ Estimating Loss with Gaussian Prior
    else
      $p_{ik} \leftarrow z_i, \mu_k, \Sigma_k \forall i, k$ ▷ Computing Responsibilities of stored samples
      Compute $\mathcal{L}_{tot}(\phi, \psi, \theta)$ with (4.4) ▷ Estimating Loss with DPMM - Prior
    end if
    $\phi, \psi, \theta \leftarrow \nabla_{\phi,\psi,\theta} \mathcal{L}_{tot}$ ▷ Updating parameters of VAE
  end for
  $\hat{\mu}_0, \hat{\lambda}, \hat{\mathcal{W}}, \hat{r}_{n_1:n_k}, \hat{\alpha}_{k_1}, \hat{\alpha}_{k_0}, \hat{\nu} \leftarrow \mathcal{Z}$ ▷ Updating parameters of DPMM
  $\mathcal{Z} = \varnothing$ ▷ Resetting buffer
end for

4.2. Prior Guided Hierarchical Reinforcement Learning

Analogous to [18] we use the trained framework to leverage downstream task learning via **Hierarchical Reinforcement Learning**. Figure 4.2 demonstrates the architecture of the **Soft Actor Critic** discussed in section 2.2.3 adapted for Skill Prior Learning. Our implementation closely follows the closed-loop Hierarchical Reinforcement Learning discussed in section 3.1.

We keep the structure and framework from the original paper [18], but explicitly emphasize the difference in behaviour caused by Generalized Adaptive Skill Prior Learning in this section.

In section 2.2.4 we mentioned the Hierarchical Reinforcement Learning and the main concepts of Skill high-level policy. Our model utilizes the Skill Prior Network $p_{\boldsymbol{a}}(z|s_1)$ trained in the previous step with frozen parameters ψ for skill embedding z. According to the process of transfer of the latent skills learned from human demonstrations to the downstream task of a long complex manipulation sequence required to be performed at this step, this procedure is **Meta-RL** by definition, which was explained in section 2.2.5.

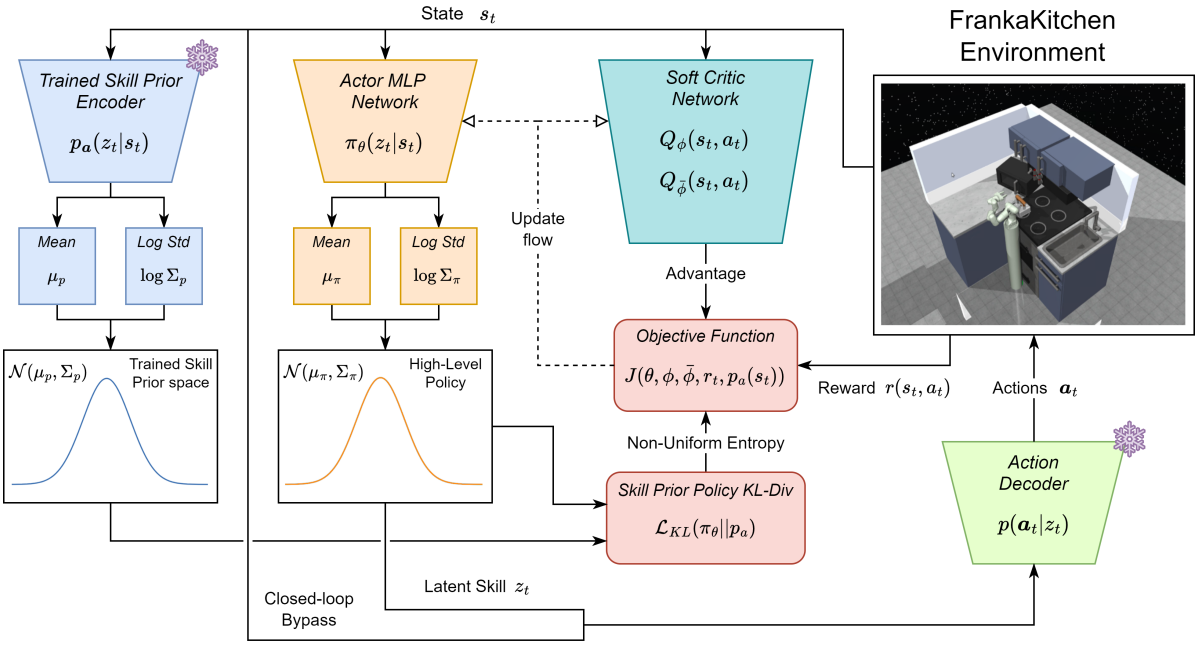


Figure 4.2.: Closed-loop HRL model with Skill Prior. The model architecture closely aligns with the implementation of the deep latent model from [18]. Solid lines represent the data flow, dashed lines represent the gradient update flow.

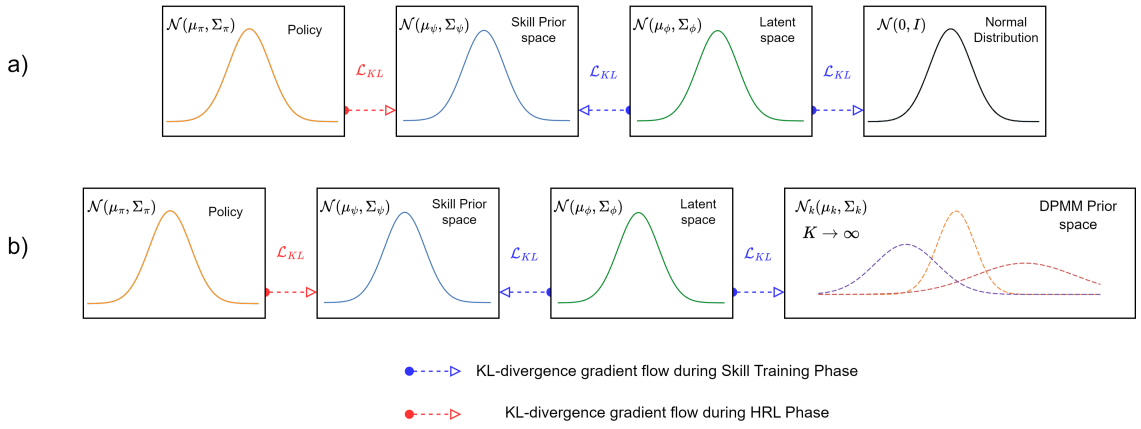


Figure 4.3.: Comparison of a) the original and b) our GASP Meta-RL KL divergence gradient flow during Skill-Prior based HRL. Since all networks try to minimize the respective KL divergence, the resulting high-level policy will behave differently.

Recall that authors of [18] used the SAC described in section 2.2.3 with a rewritten entropy term $\mathcal{H}$ for a non-uniform distribution for regularized via Skill Prior Learning $\mathcal{L}_{KL}(\pi_\theta(z_t|s_t)||p_a(z_t|s_t))$ (see section 3.1). Essentially this restricts the probabilistic Actor

Policy π to follow a Gaussian Distribution aiming to minimize KL divergence with the inferred Skill Prior, which was trained with $\mathcal{L}_{KL}(q_\phi(z|\boldsymbol{a}_i)||p_{\boldsymbol{a},\psi}(z|s_t))$ with respect to the Skill Embedding space $q_\phi(z|\boldsymbol{a}_i)$ which in turn was fitted according to the original ELBO (discussed in section 2.1.2) to the Normal distribution $\mathcal{N}(0, \boldsymbol{I})$. In contrast, our model was trained on a DPMM-distributed Prior space which down the chain makes the Skill Prior accordingly fit. From now on we will say that our Skill Prior or Actor Policy are DPMM-distributed, meaning this chain process. The comparison of the original and our distribution dependencies can be seen in Figure 4.3.

Authors of the Soft Actor Critic Algorithm [27] mentioned that the soft entropy $\mathcal{H}$ can be rewritten to represent any parametric probabilistic distribution. Unfortunately, as mentioned in section 2.3.3, DPMM is an infinite cluster distribution model and thus cannot be used directly. With that in mind, we will not alter the Hierarchical Reinforcement Learning procedure, but the resulting probabilistic policy will still be trained to represent the DPMM-distributed Skill embedding space.

5. Results

Our GASP Meta-RL model was capable of performing similarly to the original framework while simultaneously learning Skill latent space clustering with an embedded Dirichlet Process Mixture Model. In this chapter, we will focus on the practical part of the implementation, discuss the empirical results of our study, and analyze its performance from different points of view.

5.1. Implementation Details

As mentioned in section 3.1, GASP Meta-RL utilizes the framework presented in [18] by adapting the latent space to a nonparametric bayesian DPMM. Below, we will accurately analyze the framework of our model focusing on the practical details.

5.1.1. Libraries and Environments

Authors of the original framework [18] conducted several experiments with different environments. Although our work can be extended to the presented tasks, we will focus on the Kitchen environment, where the task is to learn Skill embeddings from a sequence of demonstrated subtasks.

For the ANN blocks we utilize the commonly used **Pytorch** library [120]. It provides us with various pre-built neural layers and activation functions as discussed in section 2.1.1, derives backpropagation and incorporates the optimizers for ANN-training. For clustering purposes we the use proposed in [104] **bnpy** library. As discussed in section 2.3.3, this library provides a nonparametric Bayesian clustering Dirichlet Process Mixture model with integrated birth-merge moves. For Reinforcement Learning, we utilize the **D4RL** benchmark [116] which supports offline RL suitable for the chosen SAC (refer to section 2.2.3), is based on OpenAI **gym** [121], and requires the physics engine **MuJoCo** [122] for rigid body simulation.

The mentioned D4RL benchmark contains several environments. The one we have chosen for our purposes is FrankaKitchen, which sets up the problem with a manipulator with 9 Degrees of freedom (DoF) and a simulated kitchen with interactive items [117]. D4RL offers unlabelled demonstrations of a robot performing 4 subtasks: *moving the kettle, opening the microwave, turning the burner on* and *sliding the cabinet door*. Notably, D4RL presents three possible data sets:

- **kitchen-complete-v0** provides several demonstration sequences of all 4 subtasks being executed in order.

- **kitchen-partial-v0** demonstrates complex manipulations where at some point all 4 subtasks are completed as a part of a long manipulation sequence.
- **kitchen-mixed-v0** contains mixed demonstrations of each of the 4 subtasks, but never in sequence together.

By default, we use mixed dataset to emphasize the ability to learn on unstructured demonstrations, but we also experiment with the complete dataset in section 5.2.2.

5.1.2. Additional Enhancements

In the following we will delve into specific details of our implementation that are worth mentioning. We noted in the section 2.1.2 that VAE architecture suggests μ and Σ as outputs of the encoder. In practice it is common [19][18] to learn μ and $\log \Sigma$, as shown in Figure 4.1. The reason for this is numerical stability concerning ELBO estimation. For an Encoder represented as an Artificial Neural Network, there is no computational difference between learning $\log \Sigma$ and Σ, but equation (4.4) requires the computation of the quotient between the standard deviations of two distributions, which can be rewritten using logarithmic rules as $\log \Sigma_1 - \log \Sigma_2$, making calculations easier. It is also worth noting that authors of DIVA [19] use $\log \Sigma^2$ to represent variance, while authors of SPiRL [18] utilize $\log \Sigma$. Both approaches are equivalent in computational effort; however, consistency should be maintained during integration.

Evaluating the latent clustering model was another challenge. As briefly mentioned in section 3.2.2, the original DIVA algorithm performed evaluation based on labelled datasets. In our case, D4RL Franca-Kitchen data [116] does not contain labels but only demonstrated action sequences. We will allocate a separate section 5.2.6 to discuss about what exactly our agent classifies into separate clusters. For now, it is important to say that similar to the validation of VAE discussed in 2.1.2 we choose one of the DPMM components $\mathcal{N}_k(\boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k)$ by hard assignment, sample a $\mathcal{N}_k$-distributed latent skill z and decode it as a validation instance. We found that this approach indeed fulfills the requirements of the evaluation phase and is empirically satisfying.

We also found that our model is unstable if we follow the proposed in [19] method of fitting DPMM at the end of each epoch - due to its size, networks train too slowly for dynamic reference DPMM model yield. Therefore, we introduce an less frequent DPMM fitting algorithm with exponentially growing epoch intervals i_n expressed as

$$i_n = \lfloor 2^{(\xi \cdot n / n_{tot})} \rfloor \tag{5.1}$$

where ξ determines the decay rate, n is the current epoch, n_{tot} is the total number of training epochs and $\lfloor x \rfloor$ operand returns the lower integer part. Therefore, during first $n_{tot}/\xi\%$ of training epochs, DPMM are fitted each epoch, then the intervals become larger. This addition not only stabilizes the training, but also reduces computational time.

5.2. Experiments

We conduct several experiments listed in this section. All experiments were performed on the Technical University of Munich LRZ-server, equipped with NVIDIA Tesla V100-PCIE GPU with 16GB graphics RAM. Computational time for training varies from one experiment to another and will be noted in corresponding sections. First, we will compare the general performance of our model with the original SPiRL framework [18], then discuss DPMM-distributed space and finally decode Skills generated by each component for a interpretation of obtained by our model latent space.

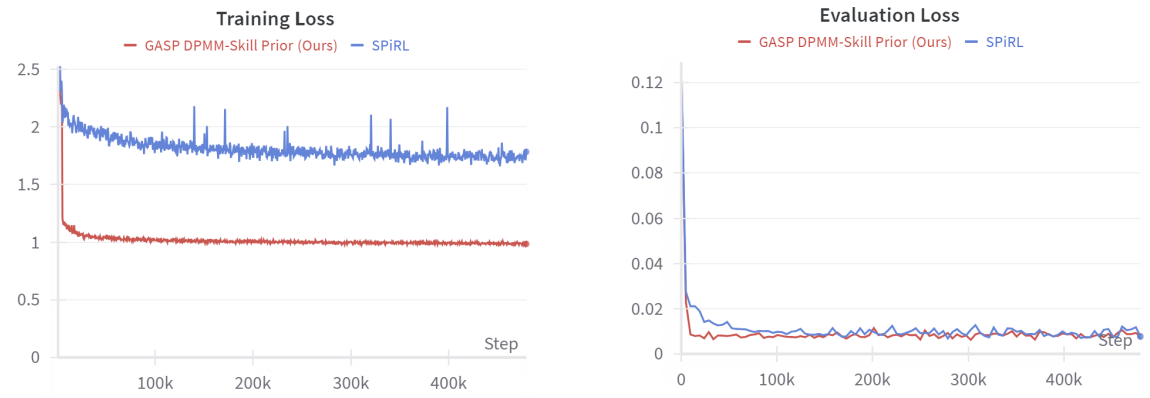
5.2.1. Baseline Performance

Since the original framework also deals with the D4RL library, we found it reasonable to leave most of the data-telated training hyperparameters unchanged. For baseline comparison, we have chosen the hyperparameters stated in Table 5.1 and the mixed FrankaKitchen dataset (kitchen-mixed-v0) from D4RL [116].

Table 5.1.: GASP Meta-RL Hyperparameters

Hyperparameter	Description	Value
dataset	Chosen Data set	kitchen-mixed-v0
optimizer	Stochastic Optimizer	RAdam
adam_beta	β_1 parameter in Adam	0.9
lr	Learning rate	1e-3
batch_size	Batch Size	128
n_lstm_layers	Number of LSTM-Layers	1
n_processing_layers	Number of Layers in Encoder (and Decoder)	5
num_prior_net_layers	Number of Layers in Prior Ensemble	6
nz_enc	Dimensionality of Encoder Input (and Decoder Output)	128
nz_vae	Dimensionality of Latent Space	10
b_minNumAtomsForNewComp	min number of atoms for new components	800
b_minNumAtomsForTargetComp	min number of atoms for target components	960
b_minNumAtomsForRetainComp	min number of atoms for retain components	960
kl_div_prior_weight	β_1 parameter from Loss equation (4.4)	1
kl_div_weight	β_2 parameter from Loss equation (4.4)	5e-4
num_epochs	Number of Epochs (Skill Prior Learning)	100
environment	Simulated environment	KitchenEnv
hl_agent	Hierarchical Reinforcement Learning Agent	SACAgent
n_rollout_steps	Fixed Skill Horizon	10
n_layers	Number of Layers in Actor (and Critic) Networks	5
num_epochs	Number of Epochs (Hierarchical Reinforcement Learning)	200

We evaluated overall performance on the configuration proposed in [18]. For this, we trained our GASP Meta-RL and original SPiRL model on the mentioned LRZ-server. Training took approximate 12 and 15 hours respectively. We also evaluated the training duration of our model, incorporating the Dirichlet Process Mixture Model fitted at each epoch as described in [19], and recorded a total training time of 35 hours. While both versions yielded similar performances, we decided to stick with the quicker model.



(a) (b)

Figure 5.1.: Overall performance of our GASP Meta-RL model (red) compared with SPiRL (blue) [18]. (a) Training loss from Skill Prior training phase. (b) Evaluation loss from Skill Prior training phase.

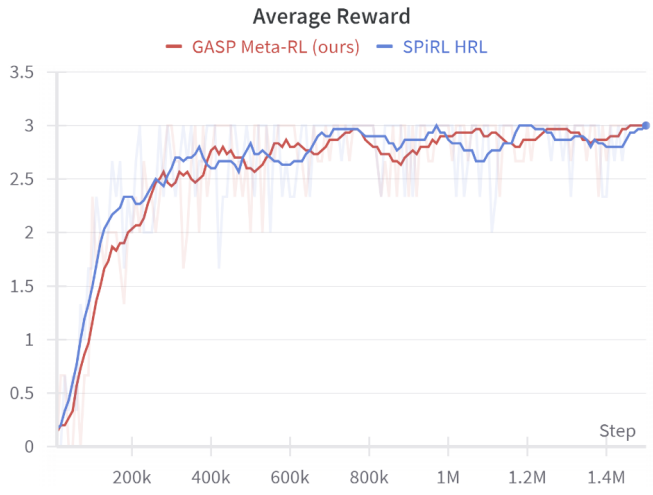


Figure 5.2.: Average number of achieved subtasks throughout the Hierarchical RL phase.

As clearly seen from Figures 5.1 and 5.2, our model performs on average at the same level as the original while also performing nonparametric Bayesian clustering and using this knowledge for guidance through the RL task. We will emphasize that our goal was to incorporate infinite cluster skill allocation into the model; thus, our next experiments are focused on variation of datasets, fine-tuning and DPMM cluster interpretation.

5.2.2. Dataset variations

We also found it interesting to test how our model perceives structured data. As mentioned in section 5.1.1, FrankaKitchen [116] presents another dataset with strurctured demonstrations of all tasks being executed in order. For the purity of the experiment, we trained our model with the same hyperparameters shown in Table 5.1 except the dataset changed to kitchen-complete-v0.

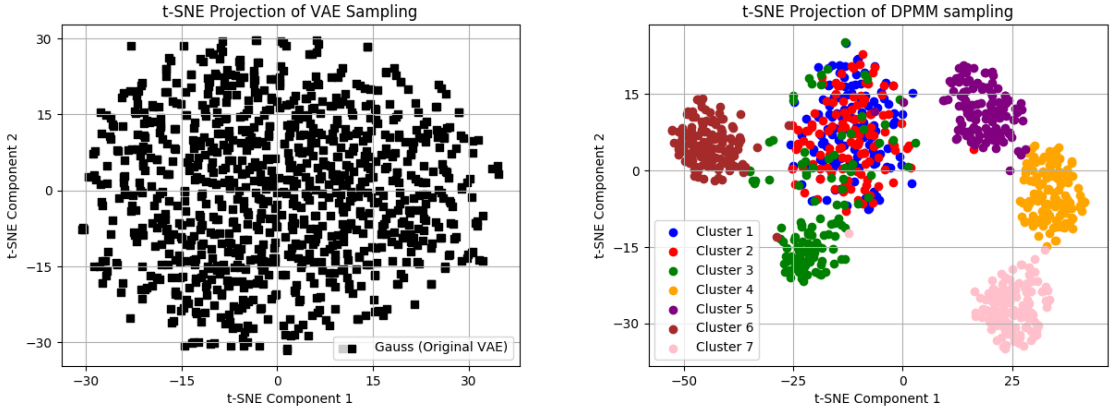
Through empirical evaluation, we concluded that our model did not bring significant improvement to overall performance. Moreover, due to DPMM-Prior clustering, our model lagged behind unmodified closed-loop model with respect to computational speed (90 minutes against 15 minutes). Compared to the experiment with the unstructured dataset where our model performed on par with and at some point better than original approach, we deduce that our model is too complicated for kitchen-complete-v0 structured demonstration cloning task, comparable with Imitation Learning approaches [10] and simply overcomplicates simple requirement. One noticeable advantage of GASP Meta-RL, however, lies in its ability to quicker adaptation to the task as shown in Figure 5.3.

Figure 5.3.: GASP trained on kitchen-complete-v0 dataset exceeds in stability, but it is comparably slow with respect to original model trained on structured demonstrations.

5.2.3. Skills Embedding Clustering

For latent Skills Embedding clusters visualization, we are using t-distributed stochastic neighbor embedding (TSNE) [123] for dimensionality reduction from *nz_vae* dimensions to human-interpretable 2D. We plot several $\mathcal{N}_k$-distributed samples for each trained DPMM component k and provide a visualization of the original Gaussian-distributed latent space of VAE for comparison with one of our GASP experiments in Figure 5.4.

The TSNE projection may be misleading due to high dimensionality; therefore, we provide comprehensive comparison of clusters with Euclidean distance between cluster means and Bhattacharyya distance in Table (5.2). At this point we utilize the frequently used in statistical analysis symmetric Bhattacharyya distance [124] instead of KL divergence since the latter describes the information loss between one probabilistic distribution approximated with another and is not symmetric by its nature (see section 2.1.2). Diagonal entries by definition are not meaningful (both Bhattacharyya and Euclidean distances will yield 0 for two similar



(a) Original unmodified Gaussian-distributed latent space (SPiRL [18])

(b) DPMM-distributed latent space (Generalized Adaptive Skill Prior)

Figure 5.4.: GASP clusters skill embeddings according to nonparametric Bayesian model. Our algorithm decides upon the number of clusters sufficient to describe the latent space by itself.

distributions), therefore we replace them with comparisons to the Normal distribution $\mathcal{N}(0, \boldsymbol{I})$, essentially computing the distance between generated by our model clusters and the fixed prior for unmodified VAE. For readability, we highlight values describing most dissimilar pair of DPMM clusters and underline the clusters values which deviate most from Normal distribution.

Table 5.2.: GASP DPMM Cluster analysis

	Euclidean Distance (Between cluster Means)							Bhattacharyya Distance						
Clusters	1	2	3	4	5	6	7	1	2	3	4	5	6	7
1	0.6012	0.0398	0.6979	1.1230	0.8254	0.9786	1.3778	7.0303	0.1463	1.1930	9.8035	3.9940	5.0629	8.9924
2	0.0398	0.5816	0.6691	1.1496	0.8585	0.9476	1.3809	0.1463	6.9189	1.1903	9.8633	3.8113	4.4289	8.4757
3	0.6979	0.6691	0.4834	1.4057	1.4074	0.8873	1.1793	1.1930	1.1903	5.4144	7.2338	3.5548	2.6714	4.8377
4	1.1230	1.1496	1.4057	1.3744	0.7148	2.0589	0.8889	9.8035	9.8633	7.2338	6.1969	4.5094	20.7263	3.6602
5	0.8254	0.8585	1.4074	0.7148	1.2870	1.7815	1.4367	3.9940	3.8113	3.5548	4.5094	5.7333	12.6616	8.8287
6	0.9786	0.9476	0.8873	2.0589	1.7815	0.8481	2.0450	5.0629	4.4289	2.6714	20.7263	12.6616	5.3451	15.8629
7	1.3778	1.3809	1.1793	0.8889	1.4367	2.0450	1.3365	8.9924	8.4757	4.8377	3.6602	8.8287	15.8629	5.2640

[1]Diagonal Entries correspond to applying the respective metric between the current cluster and $\mathcal{N}(0, \boldsymbol{I})$.

[2] Highlighted values describe the most dissimilar pair of DPMM clusters.

[3]Underlined values describe the most deviating DPMM cluster from $\mathcal{N}(0, \boldsymbol{I})$.

As discussed in sections 2.3.3 and 4.1, our model dynamically changes the number of clusters during training and utilizes instances generated during inference for cluster allocation. Therefore, the DPMM with MemoVB from the work of Hughes [104] may behave slightly differently in different runs. In Figure 5.5 we present the an example of the evolution of DPMM cluster distribution over a hundred training epochs.

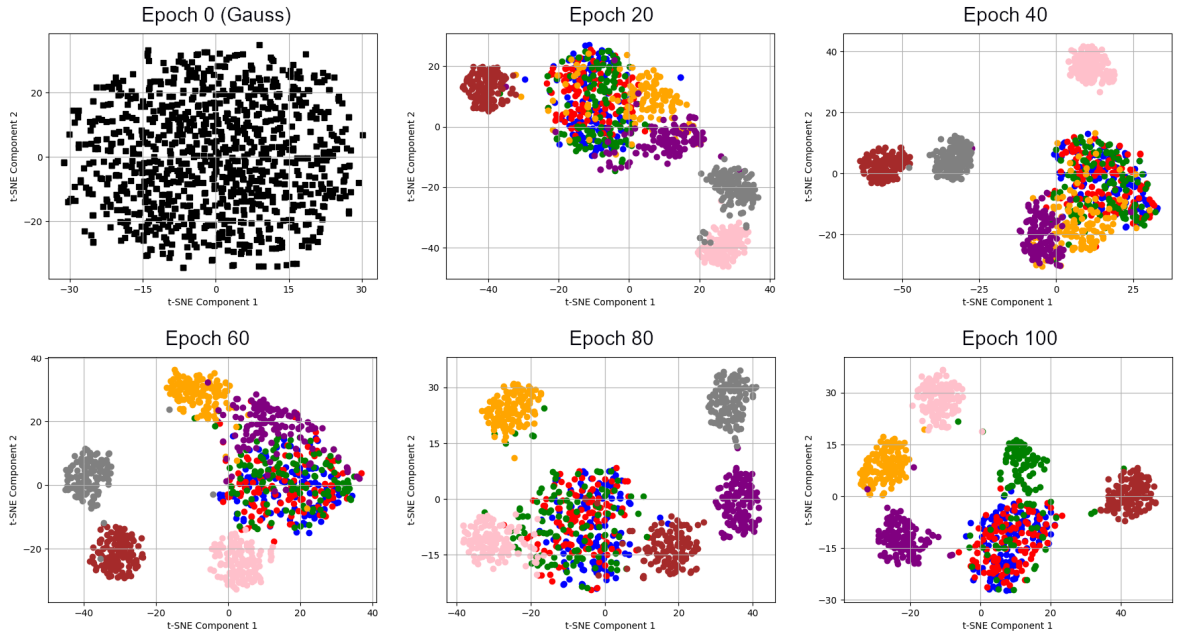


Figure 5.5.: An example of cluster evolution of DPMM-distributed prior throughout training. Note that GASP always initializes with a standard normal distribution at the first epoch.

5.2.4. Dirichlet Process Mixture Fine-Tuning

In this experiment, all models had the same parameters up to the minimum numbers required for new, target and retain components (denoted later by atomic numbers in our work for consistency) featured in work of Hughes [104] which was analysed in section 2.3.3. We empirically demonstrate that the atomic number of components plays a crucial role in forming clusters. Utilizing adaptive fitting intervals from equation (5.1) with $n_{tot} = 100$ and $\xi = 4$ we obtained 40 out of 44 DPMM fitting procedures at epoch 65; therefore, we early stopped the skill prior learning to visualize the progress. With larger values of atomic numbers, merging occurs slower and more components are created during training. On the other hand, low atomic numbers allow chaotic assignment, and clusters becomes denser. Figure 5.6 demonstrates the results of the conducted experiment.

According to our analysis as stated in table 5.1, we found by trial that 800 and 960 (also 960 for retain components) are the best atomic numbers choices for FrankaKitchen [116] tasks. It is worth mentioning that the end result remains consistent and converges to 9 ± 2 clusters and all trained models showed comparable performance at the HRL stage of training.

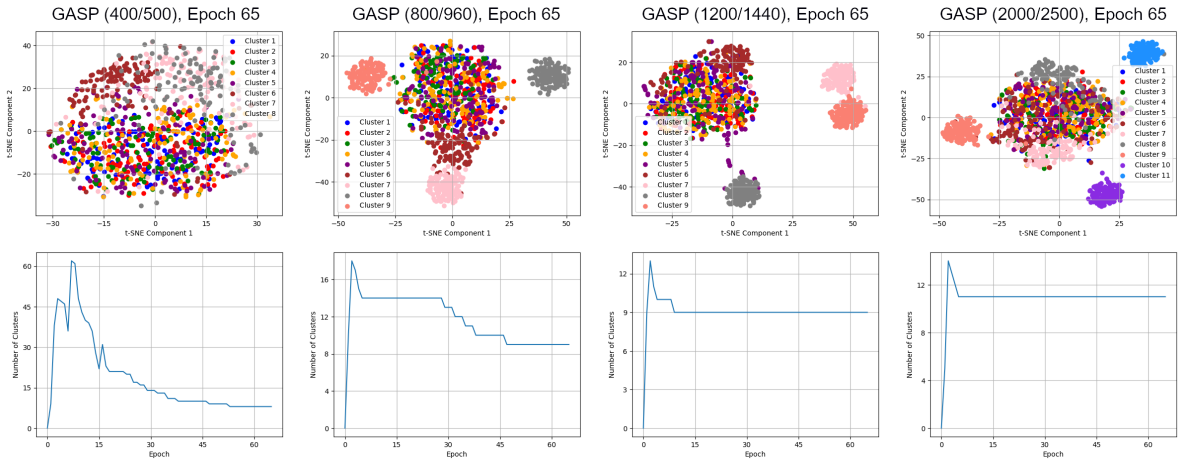


Figure 5.6.: GASP models initialized with different values of Hughes atomic numbers [104] adapt differently to the dataset (first value in brackets corresponds to b_minNumAtomsForNewComp and second to b_minNumAtomsForTargetComp and b_minNumAtomsForRetainComp).

5.2.5. Rotation of Kullback–Leibler Divergence Direction

In section 4.2 we mentioned that in the original paper [18], which was taken as the main framework of our model, Skill Embedding inverse KL divergence is computed towards a fixed Prior $\mathcal{L}_{KL}(q_\phi(z|\boldsymbol{a}_i)||\mathcal{N}(0,\boldsymbol{I}))$ and Skill Prior space $\mathcal{L}_{KL}(q_\phi(z|\boldsymbol{a}_i)||p_{\boldsymbol{a},\psi}(z|s_t))$. While the prior-conditioned loss is caused by ELBO derivation (see sections 2.1.2, 2.3.3, 4.1), the Skill Prior loss direction appeared to us as an unintuitive choice.

By the definition of Kullback–Leibler divergence we discussed in 2.1.2, we understand that if $Q = \mathcal{N}(\mu_\psi, \Sigma_\psi)$ and $P = \mathcal{N}(\mu_\phi, \Sigma_\phi)$, $\mathcal{L}_{KL}(Q||P)$ and $\mathcal{L}_{KL}(P||Q)$ will theoretically converge to the same result. For this experiment, we switched the direction of KL divergence from $\mathcal{L}_{KL}(q_\phi(z|\boldsymbol{a}_i)||p_{\boldsymbol{a},\psi}(z|s_t))$ to $\mathcal{L}_{KL}(p_{\boldsymbol{a},\psi}(z|s_t)||q_\phi(z|\boldsymbol{a}_i))$. With that, the train of thought "Policy π strives towards Skill Prior p which strives towards Latent Skill distribution q which strives towards DPMM-distribution p_{dpmm}" is complete (see Figure 4.3). As was expected, the experiment did not show any difference between the original and flipped KL divergence. Therefore this change can be denoted as purely optional and aimed towards approach consistency.

5.2.6. Latent Skills Interpretation

In the following, we examine how exactly our model utilizes the DPMM-distributed prior for the reinforcement learning task.

Since in our work we are utilizing a closed-loop architecture as mentioned in section 3.1, the decoder receives a merged vector of current states and latent skills generated by the current policy $\pi_\theta(z|\boldsymbol{s}_t)$. This property makes this particular version not capable of a convenient generative process - while Latent Skills may be generated from different DPMM components,

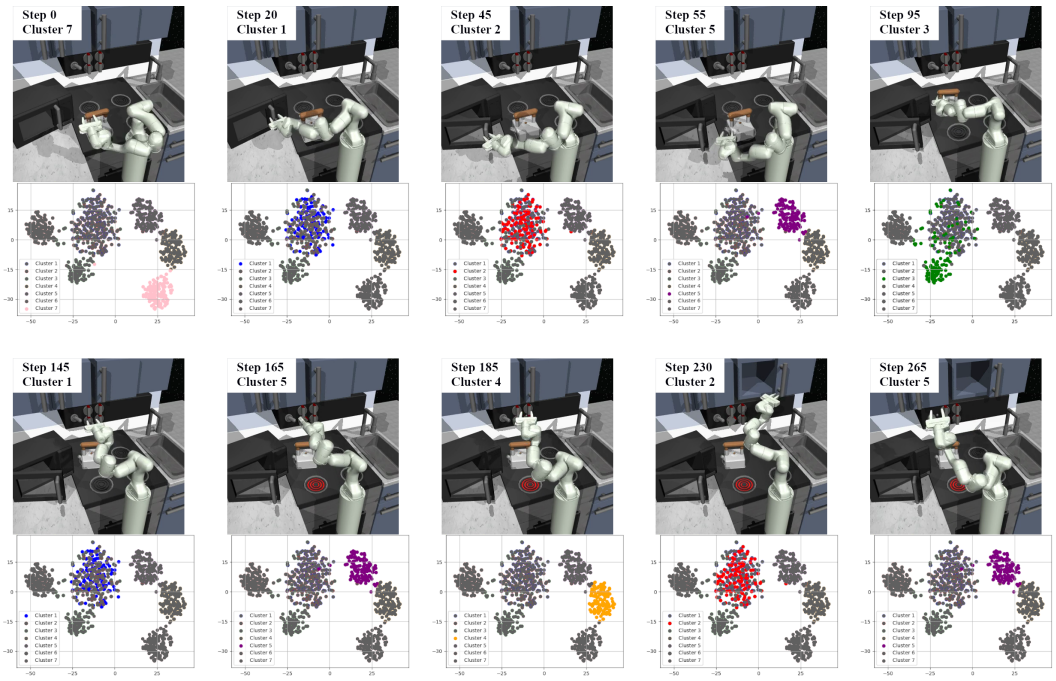


Figure 5.7.: Resulting performance of the GASP Meta-RL model with executed High-Level Skills originating from highlighted clusters.

states paired with them are unknown. To overcome this issue, we consider different approach to understand the underlying decision processes of our model.

Recalling the approach with KL divergence between policy $\pi_\theta(z_t|s_t)$ and the learned skill prior $p_a(z_t|s_t)$ discussed in section 3.1, we now may hard-assign generated skills from current policy to the corresponding component of the inferred Dirichlet Process Mixture prior. According to Figure 4.3 (b) this is equivalent to clustering of HRL high-level policy π_θ. We then plot the active clusters to track the originated DPMM components responsible for executed skills.

This experiment opens up the possibility to correlate latent processes with agent performance. In section 4.1.1 we discussed the action horizon length H the agent operates with and specified that our model trains on $H = 10$, meaning that each skill induced action sequence lasts at least 10 time steps. Consequently, the latent skills the GASP Meta-RL model infer are abstract movement primitives rather than full subtask-oriented complex actions. Figure 5.7 demonstrates how actions generated by the learned DPMM components allow the model to perform all 4 subtasks. For this experiment we analyzed the frames to interpret latent skills z indirectly sampled from each DPMM cluster, summarized everything in Table 5.3 and supported our analysis with Figure 5.8 for visualization.

Table 5.3.: GASP Meta-RL Skill Interpretation experiment

Cluster	Percentage of activity during the whole experiment	Skill Interpretation
Cluster 1	13.8 %	Linear precise positioning
Cluster 2	51.8 %	Sweeping movement
Cluster 3	6.9 %	Appears only when moving the kettle
Cluster 4	3.4 %	Shoulder lifting
Cluster 5	13.8 %	Twisting movement
Cluster 6	3.4 %	Grabber closing
Cluster 7	6.9 %	Rotational positioning

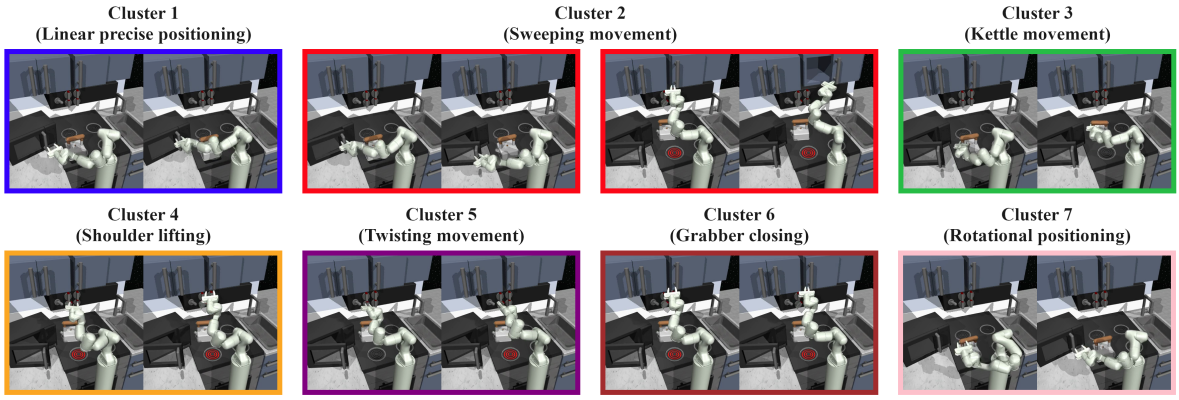


Figure 5.8.: Possible interpretations of the skills inferred by GASP Meta-RL.

For future work, we may consider enabling a flexible skill horizon H and experiment with our model's ability to extend simple sweeping motions over longer horizons and shorten skills that require precision by decoding them into separate clusters.

5.3. Discussion

We have conducted several experiments that demonstrate the effectiveness of our model. The Generalized Adaptive Skill Prior (GASP) Meta-RL framework primarily focuses on three key areas of interest, where it can be applied partially or enhanced for further needs. In the following sections, we will discuss the application of our model in modern robotics, outline the benefits it brings to the study of artificial intelligence, and conclude with a psychological comparison to human decision-making processes.

5.3.1. Robotics

Modern robotics research is primarily targeted toward industrial applications and focused on the digitalization and data-driven control [125]. Essentially robots represent the physical embodiment of a control agent for real-world interactions regardless of whether this is

an automated guided vehicle or a manipulator. Since our work focused on the simulated environment, the next possible step would be to replicate this experiment in the real world. This process is called Sim-to-Real [126] transfer and requires additional steps, such as CV-aided scene representation and error propagation. Our model was trained with the default configuration of the D4RL dataset [116] with a 9-DoF Franka robot, but naturally our approach can be extended to other environments with different task specifications.

Modern research in this topic aims at making robots more versatile and flexible [125]. With recorded demonstration sequences and potentially unlimited interpretation possibilities, our model may be applied to any task requiring non-algorithmic approach or dynamic adaptation such as, for example, human-machine collaboration.

5.3.2. Artificial Agent

On the one hand, the idea of utilizing Skill Embedding for Reinforcement Learning is not new [16][43]. On the other, studies in unsupervised clustering have gone a long way from K-means [15] to nonparametric Bayesian models like DPMM with memoized Bayes [104].

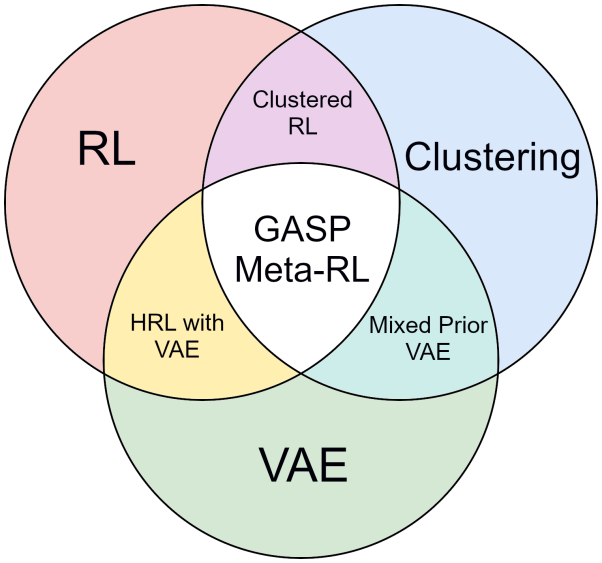


Figure 5.9.: GASP Meta-RL combines several fields of study in Artificial Intelligence allowing our agent to benefit from advanced aspects of each of them.

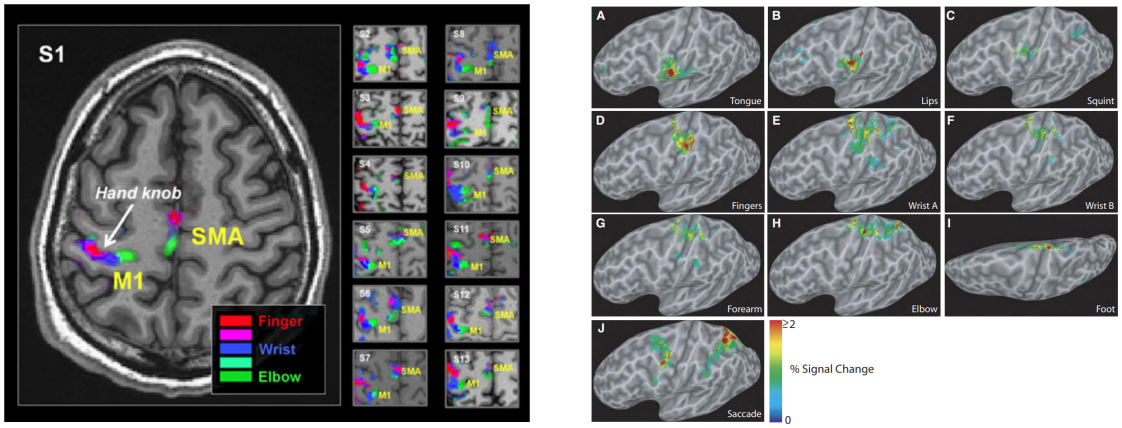
We have discussed in section 2.2.4 how Hierarchical Reinforcement Learning may utilize high-level policies for efficient decision making and highlighted some attempts at making latent space non-Gaussian distributed (original SPiRL paper [18] contains the implementation of GMVAE, but the authors consider it for improving the efficiency of their model and thus restrain from using it in further research), but to our knowledge we are the first to propose an artificial agent for HRL based on a DPMM-distributed VAE. Our approach successfully combines algorithms from several fields of study (Figure 5.9). By going away from a fixed number of clusters, we essentially give the agent full freedom of interpretation. Recalling

equation (4.4), we constrain unsupervised learning only in terms of reconstruction and ELBO. The agent chooses the number of clusters fully autonomously and confirms the consistency of its choice in the HRL step of learning. Next, we will compare this behaviour with the biological learning characteristic for sentient beings.

5.3.3. Neuroscience

Despite remarkable scientific breakthroughs, humanity still cannot fully comprehend the enigma of sentience. Consequently, it is crucial to investigate emerging concurrences across different fields, drawing inspiration from biology while also using our observations of machine behavior to gain insights into our own cognitive processes.

Neuroscience is the field of science studying the function and structure of the nervous system, including the brain [127]. Different researches have pointed out the dependencies between human actions and specific regions of the cerebral cortex [128][129][130]. For our purposes, we will focus on the motor cortex responsible, among other things, for body part movements [127]. The study of neuroimaging, which is responsible for the visualisation of brain functions and the mapping of brain regions to specific actions, commonly uses functional magnetic resonance imaging (fMRI) for real-time visualization [129]. Two selected studies [130][128] have shown the correspondence between different human hand manipulations and the respective regions of the motor cortex, as shown in Figure 5.10.



(a) fMRI gradients for index finger, wrist and elbow movements for 13 subjects [128].

(b) Brain regions activated by performing different movement primitives [130].

Figure 5.10.: Neuroimaging of patients performing simple manipulations.

Compared with Figure 5.7, we can show that our algorithm can be viewed as a biologically inspired neurological model similarly utilizing inferred DPMM components for different skills that are later decoded into action sequences. WIt is important to underline that the activated brain regions possess general patterns of clusters, but their shape, position, and number are inconsistent (see Figure 5.10 (a)) and thus cannot be specified mathematically, which makes the choice of a nonparametric Bayesian Dirichlet Process Mixture superior.

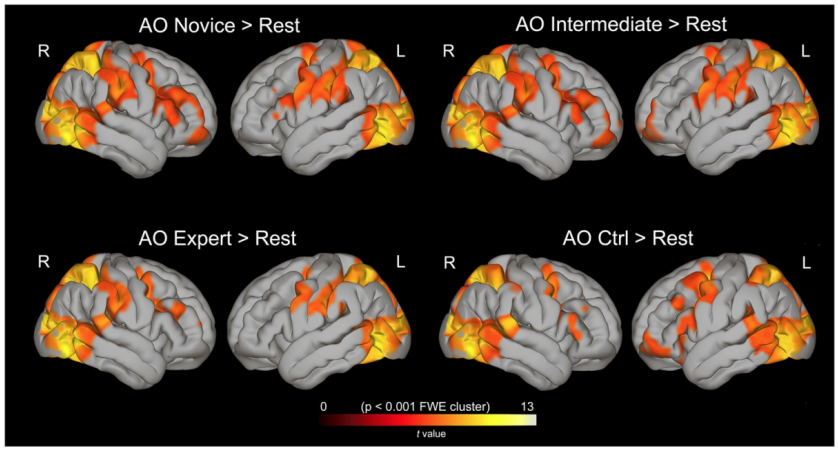


Figure 5.11.: Difference between rest condition and complex manipulation induced brain activations observed in novice, intermediate, expert, and control (imaginary object manipulation) performers [129].

Throughout their lives humans sharpen their skills. There is a debate on whether this process is conscious or subconscious, but the authors of [129] visualize how mastering a skill affects the brain activation regions (Figure 5.11).

Authors of the original paper recorded fMRI scans of a control group of eighteen individuals observing complex hand manipulation. Figure 5.11 shows how the quality of observed action affects the cortex activation. It is clear that activated brain area by the demonstration of expert movement is more structured and concise than that performed by a novice [129].

These results can be explained by the entropy of actions - redundant, novice skills cause dispersed brain activation, while masterful is structured more efficiently in the cerebral cortex. This observation has two main implications. Firstly, recalling Figure 5.5, we can identify similar convergence in our model, where understanding of performed actions is structured into compact clusters, which further supports the importance of latent skill clustering. Secondly, the quality of demonstrated actions may lead to less structured order in the DPMM-distributed prior. Unfortunately, we did not have a dataset comparable to [116] that includes demonstrations of varying quality, particularly examples of poor or non-masterful performance. We consider this an interesting area for future experimentation.

6. Conclusion

In this work, we have presented Generalized Adaptive Skill Prior (GASP) Meta-Reinforcement Learning, a novel approach for nonparametric Bayesian clustering of the Skill Embedding space for Hierarchical Reinforcement Learning. Our research integrates several Artificial Intelligence studies focusing on Robotics, including Deep Artificial Neural Networks, Machine Learning Clustering Algorithms, and Reinforcement Learning.

We analyzed different frameworks and structured our research from fundamental to state-of-the-art. We proposed a content breakdown of Deep Learning Architectures, describing the generative Variational Autoencoder model for Skill Embeddings and emphasizing the importance of sequence processing by Long Short-Term Memory modules for the perception of long demonstrated action sequences. We then explored Reinforcement Learning, proposing the use of the value and policy-based model-free off-policy Soft Actor Critic algorithm for Hierarchical Reinforcement Learning, supported by a careful analysis of modern applications in Robotics. For our model, we suggested using a Dirichlet Process Mixture Model-based Prior for adaptive and generalized Skill Prior clustering, supported by a meaningful analysis of comparable clustering algorithms.

Our solution is a complex framework combining modern developments with nonparametric adaptive Skill Prior learning for entropy-based guidance through Hierarchical Reinforcement Learning. Our experiments demonstrated several key findings. We evaluated the baseline performance of the GASP Meta-RL model against the original SPiRL framework. Both models were trained on the same dataset (kitchen-mixed-v0) using similar hyperparameters. Our model showed competitive performance with the original framework while incorporating nonparametric Bayesian clustering. However, it was overly complicated for structured datasets (kitchen-complete-v0), showing a stable yet slow learning process. This indicates that while GASP Meta-RL excels in unstructured scenarios, it might be more complex than necessary for simpler, structured tasks.

Experiments with different atomic numbers and visualization of cluster evolution demonstrated the adaptive learning process, bringing new insights into skill definitions. GASP autonomously determines the quantity and form of clusters for optimal task adaptation. Hyperparameters set the framework, but the task itself fills it with patterns. We also analyzed how our model uses the DPMM-distributed prior for reinforcement learning tasks. The hard assignment of latent skills to DPMM components allowed us to interpret the underlying decision processes. This experiment highlighted the model's capability to generate meaningful movement primitives and perform complex manipulation tasks by leveraging clustered skills. Therefore, we emphasize the agility of our solution and its comparability with subconscious processes in the human cerebral cortex concerning learning and performing long and complex manipulation action sequences.

A. Appendix

We have chosen to avoid including code snippets in the main body of our work, recognizing that practical modeling and experiments are integral to our research. In this Appendix, we will carefully describe all the modifications and standalone modules developed for this study. To be short, we only write code snippets important for understanding, our full framework's public repository is accessible at `https://github.com/Eibozhenko-Mikhail/SPIRL_DPMM.git`. Due to the large size of the experimental weights for the models proposed in this work, they are not included in the repository. Those who are interested can request access to these weights by contacting the author, Mikhail Eibozhenko, at `mikhailerox@gmail.com`.

`SPIRL_DPMM/spirl/configs`

This folder contains all configuration files for skill prior learning and HRL.

`SPIRL_DPMM/spirl/configs/skill_prior_learning/kitchen`

All configurations of main skill prior learning experiments conducted for our research are listed here. In every configuration file a commentary about this experiment version can be found. For example, `spirl_DPMM_h_cl_v_11_06_complete` contains following commentary:

```
########################## Experiment version ########################
#
# This version was created on 11.06
#
# Differencies from original:
# - DPMM
# - Correct evaluation
# - Adaptive DPMM fitting
# - b_minNumAtomsForNewComp=1200.0,
# - b_minNumAtomsForTargetComp=1440.0,
# - b_minNumAtomsForRetainComp=1440.0,
# - Training on complete kitchen dataset
```

meaning that this version contains working Dirichlet Process Mixture Model Prior, correct evaluation and adaptive fitting (see section 5.1.2), was trained with corresponding atomic numbers and on kitchen-complete-v0 dataset (see section 5.2.2).

`SPIRL_DPMM/spirl/configs/hrl/kitchen`

This folder contains configurations for Hierarchical Reinforcement Learning stage of GASP

Meta-RL. Similarly to skill prior configuration files, each file contains experiment version specification.

SPIRL_DPMM/spirl/models/CL_SPIRL_DPMM_mdl.py

This file is the main framework of Dirichlet Process Mixture Model based Skill Prior. As mentioned in section 3.1.2, this model is inherited from original SPiRL model `SkillPriorMdl` and comes as augmented version of closed loop `ClSPiRLMdl`. First we adjust default initialization enabling DPMM creation:

```
def __init__(self, params, logger=None):
    super().__init__(params, logger=logger)

    # Hughes hyperparameters
    self.dpmm_param = dict(
        sF=0.1,
        b_minNumAtomsForNewComp=800.0,
        b_minNumAtomsForTargetComp=960.0,
        b_minNumAtomsForRetainComp=960.0,
    )

    # DPMM parameters
    pwd = os.getcwd()
    self.bnp_root = pwd + '/save/bn_model/'
    self.bnp_iterator = cycle(range(2))
    self.bnp_model = None
    self.bnp_info_dict = None
    self.comp_var = None
    self.comp_mu = None
    self.num_clusters = 0
    self.cluster_logging = []
```

Obviously, we need to adjust forward pass with distinction for validation (see equations (4.4) and (4.1)). Gradient flow is turned off simultaniously with setting `self._sample_prior` flag to *True*:

```
if self._sample_prior: # if validation
    if not use_learned_prior and self.bnp_model:
        print("Validation based on DPMM...")
        z = output.q.sample()
        _, hard_assignment = self.cluster_assignments(z) # [batch_size]
        zs_sampled = []
        for i in range(len(hard_assignment)):
            k = hard_assignment[i]
            z_sampled = torch.distributions.MultivariateNormal(
                loc=self.comp_mu[k].to(z.device),
                covariance_matrix=torch.diag_embed(self.comp_var[k],
            ).to(z.device)).sample()
            zs_sampled.append(z_sampled)
        z_component = torch.stack(zs_sampled, dim=0)
```

```
        output.z = z_component
        output.z_q = z # for loss computation
    else:
        print("Validation based on Gauss...")
        output.z = output.p.sample()
        output.z_q = output.q.sample() # for loss computation
else: # if training/inference
    output.z = output.q.sample()
    output.z_q = output.z.clone() # for loss computation
```

As discussed in sections 3.2 and 4.1.2, we rework loss computation for SPiRL framework by adding weighted sum of KL-divergence between components of DPMM and output of encoder (class `DivaKLDivLoss` will be discussed later):

```
def loss(self, model_output, inputs):
    """Loss computation of the SPIRL_DPMM model.
    :arg model_output: output of SPIRL_DPMM model forward pass
    :arg inputs: dict with 'states', 'actions', 'images' keys from data loader
    """

    # Rewriting the method of skill_prior model for DIVA functionality
    losses = AttrDict()

    # reconstruction loss, assume unit variance model output Gaussian

    losses.rec_mse = NLL(self._hp.reconstruction_mse_weight) \
        (Gaussian(model_output.reconstruction, torch.zeros_like(model_output.reconstruction)),
         self._regression_targets(inputs))

    # KL loss distinction (At epoch 0 - initializing DPMM, standart Gauss-based computation:)
    ########____________________ Final Version: _________________########

    if not self.bnp_model:
        # Gauss based Loss (Default VAE)
        losses.kl_loss = KLDivLoss(self.beta)(model_output.q, model_output.p)
    else:
        # DPMM based Loss (DPMM Prior)
        z = model_output.z_q.detach()
        comp_mu = self.comp_mu
        comp_var = self.comp_var
        prob_comps, hard_assignment = self.cluster_assignments(z) # prob_comps --> resp, comps
            --> Z[n]
        _, self.num_clusters = prob_comps.shape
        losses.kl_loss = DivaKLDivLoss(self.beta)(model_output.q.mu, model_output.q.log_sigma,
            prob_comps, comp_mu, comp_var)

    # learned skill prior net loss
    losses.q_hat_loss = self._compute_learned_prior_loss(model_output)

    # Optionally update beta
    if self.training and self._hp.target_kl is not None:
        self._update_bedta(losses.kl_loss.value)

```

```
    losses.total = self._compute_total_loss(losses)
```

Lastly, we introduce `fit_dpmm()` method required for adapting DPMM to the latent samples generated throughout learning as well as two auxiliary functions analogous to [19]:

```
def fit_dpmm(self, z):
    z = XData(z.detach().cpu().numpy())
    if not self.bnp_model:
      print("************************************************************")
      print("_________________ Initialing DPMM model ... _________________")
      self.bnp_model, self.bnp_info_dict = bnpy.run(z, 'DPMixtureModel', 'DiagGauss', 'memoVB',
                                            output_path = self.bnp_root+str(next(self.
                                                bnp_iterator)),
                                            initname='randexamples',
                                            K=1, gamma0 = 5.0, sF=0.1,
                                            ECovMat='eye',
                                            b_Kfresh=5, b_startLap=0, m_startLap=2,
                                            # moves='birth,merge,shuffle',
                                            moves='birth,delete,merge,shuffle',
                                            nLap=2,
                                            b_minNumAtomsForNewComp=self.dpmm_param['
                                                b_minNumAtomsForNewComp'],
                                            b_minNumAtomsForTargetComp=self.dpmm_param['
                                                b_minNumAtomsForTargetComp'],
                                            b_minNumAtomsForRetainComp=self.dpmm_param['
                                                b_minNumAtomsForRetainComp'],
                                           )
    else:
      print("************************************************************")
      print("__________________ Fitting DPMM model ... __________________")
      self.bnp_model, self.bnp_info_dict = bnpy.run(z, 'DPMixtureModel', 'DiagGauss', 'memoVB',
                                            output_path = self.bnp_root+str(next(self.
                                                bnp_iterator)),
                                            initname=self.bnp_info_dict['task_output_path'],
                                            K=self.bnp_info_dict['K_history'][-1],
                                            gamma0=5.0,
                                            sF=self.dpmm_param['sF'],
                                            b_Kfresh=5, b_startLap=1, m_startLap=2,
                                            # moves='birth,merge,shuffle',
                                            moves='birth,delete,merge,shuffle',
                                            nLap=2,
                                            b_minNumAtomsForNewComp=self.dpmm_param['
                                                b_minNumAtomsForNewComp'],
                                            b_minNumAtomsForTargetComp=self.dpmm_param['
                                                b_minNumAtomsForTargetComp'],
                                            b_minNumAtomsForRetainComp=self.dpmm_param['
                                                b_minNumAtomsForRetainComp'],
                                           )
    self.calc_cluster_component_params()
    print("_____________________ End of DPMM Phase _____________________")
    print("************************************************************")

def cluster_assignments(self, z):
```

```
    z = XData(z.detach().cpu().numpy())
    LP = self.bnp_model.calc_local_params(z)
    # Here, resp is a 2D array of size N x K. here N is batch size, K active clusters
    # Each entry resp[n, k] gives the probability that data atom n is assigned to cluster k
        under
    # the posterior.
    resp = LP['resp']
    # To convert to hard assignments
    # Here, Z is a 1D array of size N, where entry Z[n] is an integer in the set {0, 1, 2, ...
        K-1, K}.
    # Z represents for each atom n (in total N), which cluster it should belongs to accroding
        to the probability
    Z = resp.argmax(axis=1)
    return resp, Z

def calc_cluster_component_params(self):
    self.comp_mu = [torch.Tensor(self.bnp_model.obsModel.get_mean_for_comp(i)) for i in np.
        arange(0, self.bnp_model.obsModel.K)]
    self.comp_var = [torch.Tensor(np.sum(self.bnp_model.obsModel.get_covar_mat_for_comp(i),
        axis=0)) for i in np.arange(0, self.bnp_model.obsModel.K)]
```

This class is the core of our solution, implementing non-parametric Bayesian model into original Variational Autoencoder. Nevertheless, another tweaks to the whole framework are required for final adaptation of this model.

SPIRL_DPMM/spirl/modules/losses.py

In the original code of [18], the losses are implemented as classes. This approach required significant adaptation work based on the methodologies of [19] and [104]. Our class `DivaKLDivLoss` corresponds to method proposed in [19] with respect to differences to SPiRL framework listed in section 5.1.2:

```
class DivaKLDivLoss(Loss):
    # Actual DIVA KL Divergence part
    def compute(self, mu, log_sigma, prob_comps, comp_mu, comp_var):
        """
        :arg inputs: mu, log_sigma of encoder, probabilistic assignments, current DPMM
            parameters mu and var
        """
        # We consider only probabilistic assignments, not hard ones
        # get a distribution of the latent variables
        var = torch.exp(2*log_sigma)
        # batch_shape [batch_size], event_shape [latent_dim]

        # Computing the Multivariate distributions:
        dist = torch.distributions.MultivariateNormal(loc=mu,
                                                      covariance_matrix=torch.diag_embed(var))
        # get a distribution for each cluster
        B, K = prob_comps.shape # batch_shape, number of active clusters
        kld = torch.zeros(B).to(mu.device)
```

```
        for k in range(K):
            # batch_shape [], event_shape [latent_dim]
            prob_k = prob_comps[:, k]
            dist_k = torch.distributions.MultivariateNormal(loc=comp_mu[k].to(mu.device),
                                                            covariance_matrix=torch.diag_embed(comp_var[
                                                                k]).to(mu.device))
            # batch_shape [batch_size], event_shape [latent_dim]
            expanded_dist_k = dist_k.expand(dist.batch_shape)

            kld_k = torch.distributions.kl_divergence(dist, expanded_dist_k) # shape [
                batch_shape, ]
            kld += torch.from_numpy(prob_k).to(mu.device) * kld_k

        kl_divergence = torch.mean(kld)
        return kl_divergence
```

SPIRL_DPMM/spirl/train.py

This file is responsible for data preparation and model training. Our changes here mainly concern adding conditioned fitting of Dirichlet Process Mixture Model and rewriting checkpoint parameters for inferred components saving.

```
for self.batch_idx, sample_batched in enumerate(self.train_loader):
    data_load_time.update(time.time() - end)
    inputs = AttrDict(map_dict(lambda x: x.to(self.device), sample_batched))
    with self.training_context():
        self.optimizer.zero_grad()
        output = self.model(inputs)
        output_list.append(output.z) # Combining all latent outputs together for DPMM
        losses = self.model.loss(output, inputs)
        losses.total.value.backward()
        self.call_hooks(inputs, output, losses, epoch)

        if self.global_step < self._hp.init_grad_clip_step:
            # clip gradients in initial steps to avoid NaN gradients
            torch.nn.utils.clip_grad_norm_(self.model.parameters(), self._hp.init_grad_clip)
        self.optimizer.step()
        self.model.step()

    if self.args.train_loop_pdb:
        import pdb; pdb.set_trace()

    upto_log_time.update(time.time() - end)
    if self.log_outputs_now and not self.args.dont_save:
        self.model.log_outputs(output, inputs, losses, self.global_step,
                               log_images=self.log_images_now, phase='train', **self.
                                   _logging_kwargs)
    batch_time.update(time.time() - end)
    end = time.time()

```

```
    if self.log_outputs_now:
        print('GPU {}: {}'.format(os.environ["CUDA_VISIBLE_DEVICES"] if self.use_cuda else '
            none',
                                  self._hp.exp_path))
        print(('itr: {} Train Epoch: {} [{}/{} ({:.0f}%)]\tLoss: {:.6f}'.format(
               self.global_step, epoch, self.batch_idx, len(self.train_loader),
               100. * self.batch_idx / len(self.train_loader), losses.total.value.item())))

        print('avg time for loading: {:.2f}s, logs: {:.2f}s, compute: {:.2f}s, total: {:.2f}s'
              .format(data_load_time.avg,
                      batch_time.avg - upto_log_time.avg,
                      upto_log_time.avg - data_load_time.avg,
                      batch_time.avg))
        togo_train_time = batch_time.avg * (self._hp.num_epochs - epoch) * epoch_len / 3600.
        print('ETA: {:.2f}h'.format(togo_train_time))

    del output, losses
    self.global_step = self.global_step + 1

### FIT DPMM at the end of the epoch
outputs = torch.stack(output_list)

fit_dpmm = False

# Adaptive Fitting
if epoch == self.next_DPMM_fitting_epoch:
    fit_dpmm = True
    interval = int(np.exp(4*epoch/self._hp.num_epochs)//1)
    self.next_DPMM_fitting_epoch += interval
```

SPIRL_DPMM/spirl/rl

This folder contains all necessary files to set up and run Skill Prior based Reinforcement Learning. As we mentioned in section 4.2, trained Skill Prior network already follows the DPMM distribution, therefore no significant changes should be made. Nevertheless we introduced some modifications purely for analytical experiments from our work.

SPIRL_DPMM/spirl/rl/train.py

This file runs the training of Hierarchical Reinforcement Learning agent. At the setup stage we also download the DPMM parameters from the checkpoint:

```
###################### - - - Updating DPMM parameters for HRL experiments - - -


weights_file = CheckpointHandler.get_resume_ckpt_file(ckpt, path)
# TODO(karl): check whether that actually loads the optimizer too
self.global_step, start_epoch, _ = \
    CheckpointHandler.load_weights(weights_file, self.agent,
```

```
                                load_step=True, strict=self.args.strict_weight_loading)
self.agent.load_state(self._hp.exp_path)
if hasattr(self.agent, "bnp_model"):
        print("This agent uses bnp_model, updating weights...")
        checkpoint = torch.load(weights_file, map_location=self.agent.device)
        self.agent.bnp_model = checkpoint['DPMM_bnp_model']
        self.agent.bnp_info_dict = checkpoint['DPMM_bnp_info_dict']
        self.agent.comp_mu = checkpoint['DPMM_comp_mu']
        self.agent.comp_var = checkpoint['DPMM_comp_var']
        self.agent.cluster_logging = checkpoint['DPMM_logging_clusters']
        print("DPMM Components updated!")
self.agent.to(self.device)

##############################################################################
```

If render flag is *True*, algorithm will generate rollouts with episode recordings, allowing to spectate the simulation of agent training.

`SPIRL_DPMM/spirl/rl/components/agent.py`

This algorithm contains class of Hierarchical Agent which is collective definition of high- and low-level policies discussed in section 2.2.4. As high-level policy generates an action corresponding to latent Skill vector z, we may hard-assign this vector with low-level agent policy network to one of DPMM clusters (for intuition refer to section 5.2.6).

```
# Tracking the original component responsible for this action
_, hard_assignment = self.ll_agent.policy.net.cluster_assighnments(torch.from_numpy(self.
    _last_hl_output.action))
output.dpmm_hard_assignment = hard_assignment[0]
```

`SPIRL_DPMM/visualisations`

This folder contains latent cluster visualisation algorithm `DPMM_vis.py` and debugging tool for DPMM-based inference `SD_inference.py`. These algorithms served for work-related analysis, t-SNE decomposition and images creation. For full code refer to `https://github.com/Eibozhenko-Mikhail/SPIRL_DPMM.git`.

Acronyms

A2C Advantage Actor-Critic. 23, 25

AI Artificial Intelligence. 1, 2

ANN Artificial Neural Network. 3–6, 20, 22, 23, 25, 49, 54, 78

CV Computer Vision. 28, 30, 64

DIVA Dirichlet Process Based Incremental Deep Clustering Algorithm via Variational Auto-Encoder. 2, 45, 46, 55, 79

DoF Degrees of freedom. 54, 64

DPMM Dirichlet Process Mixture Model. 33, 34, 37–41, 45–51, 53–62, 64–67, 78, 79, 81

DQN Deep Q-Network. 20, 21, 78

ELBO Evidence Lower Bound. 9, 37, 39, 41, 49, 53, 55, 61, 65

EM Expectation-Maximization. 29, 32, 37, 39, 78

FFNN Feed-forward Neural Network. 4–6, 10

GASP Generalized Adaptive Skill Prior. 2, 47, 52, 54, 56–64, 67, 79–81

GMM Gaussian Mixture Model. 30–35, 37, 39, 78

GMVAE Gaussian Mixture Variational Autoencoder. 10, 45, 64

HRL Hierarchical Reinforcement Learning. 26–28, 52, 60, 62, 64, 65, 79

KL Kullback–Leibler. 8, 25, 43, 46, 49, 50, 52, 53, 58, 61, 62, 78, 79

LSTM Long Short-Term Memory. 11, 12, 47, 78

MDP Markov Decision Process. 13–15, 78

MemoVB Memoized Online Variational Bayes. 39, 41, 46, 59

POMDP Partially observable Markov Decision Process. 14, 15, 20, 78

RL Reinforcement Learning. 1, 13, 15–18, 20, 21, 24, 26–28, 42, 43, 47, 51, 52, 54, 56–58, 62–64, 67, 79–81

SAC Soft Actor Critic. 24–26, 43, 44, 52, 54, 79

SPiRL Skill-Prior Reinforcement Learning. 2, 42–44, 55–57, 59, 64, 67, 79

VAE Variational Autoencoder. 3, 8–10, 42, 44–46, 49–51, 55, 58, 59, 64, 78

List of Figures

List of Tables

Bibliography

[1] G. Clark. "Industrial Revolution". In: *Economic Growth*. Springer, 2010, pp. 148–160.

[2] R. Forrester. "The Invention of the Steam Engine". In: *SocArXiv, preprint, Oct* (2019).

[3] J. Mokyr and R. H. Strotz. "The second industrial revolution, 1870-1914". In: *Storia dell'economia Mondiale* 21945.1 (1998).

[4] S. Loughlin. "Industry 3.0 to Industry 4.0: Exploring the transition". In: *New Trends in Industrial Automation*. IntechOpen, 2018.

[5] Y. N. Harari. *Sapiens: A brief history of humankind*. Random House, 2014.

[6] J. Barata and I. Kayser. "Industry 5.0–Past, present, and near future". In: *Procedia Computer Science* 219 (2023), pp. 778–788.

[7] M. Wooldridge. *A brief history of artificial intelligence: what it is, where we are, and where we are going*. Flatiron Books, 2021.

[8] E. E. Aksoy, A. Orhan, and F. Wörgötter. "Semantic decomposition and recognition of long and complex manipulation action sequences". In: *International Journal of Computer Vision* 122 (2017), pp. 84–115.

[9] A. K. Shakya, G. Pillai, and S. Chakrabarty. "Reinforcement learning algorithms: A brief survey". In: *Expert Systems with Applications* (2023), p. 120495.

[10] A. Hussein, M. M. Gaber, E. Elyan, and C. Jayne. "Imitation learning: A survey of learning methods". In: *ACM Computing Surveys (CSUR)* 50.2 (2017), pp. 1–35.

[11] R. Szeliski. *Computer vision: algorithms and applications*. Springer Nature, 2022.

[12] M. Treviso, J.-U. Lee, T. Ji, B. v. Aken, Q. Cao, M. R. Ciosici, M. Hassid, K. Heafield, S. Hooker, C. Raffel, et al. "Efficient methods for natural language processing: A survey". In: *Transactions of the Association for Computational Linguistics* 11 (2023), pp. 826–860.

[13] S. Arora and P. Doshi. "A survey of inverse reinforcement learning: Challenges, methods and progress". In: *Artificial Intelligence* 297 (2021), p. 103500.

[14] C. Sammut and G. I. Webb. *Encyclopedia of machine learning*. Springer Science & Business Media, 2011.

[15] C. M. Bishop and N. M. Nasrabadi. *Pattern recognition and machine learning*. Vol. 4. 4. Springer, 2006.

[16] S. Thrun and A. Schwartz. "Finding structure in reinforcement learning". In: *Advances in neural information processing systems* 7 (1994).

[17] P. Brazdil, C. G. Carrier, C. Soares, and R. Vilalta. *Metalearning: Applications to data mining*. Springer Science & Business Media, 2008.

[18] K. Pertsch, Y. Lee, and J. Lim. "Accelerating reinforcement learning with learned skill priors". In: *Conference on robot learning*. PMLR. 2021, pp. 188–204.

[19] V. AUTO-ENCODER. "DIVA: ADirichlet PROCESS MIXTURES BASED INCREMENTAL DEEP CLUSTERING ALGORITHM VIA VARIATIONAL AUTO-ENCODER". In: ().

[20] W. S. McCulloch and W. Pitts. "A logical calculus of the ideas immanent in nervous activity". In: *The bulletin of mathematical biophysics* 5 (1943), pp. 115–133.

[21] F. Rosenblatt. "The perceptron: a probabilistic model for information storage and organization in the brain." In: *Psychological review* 65.6 (1958), p. 386.

[22] H. Wang and B. Raj. "On the origin of deep learning". In: *arXiv preprint arXiv:1702.07800* (2017).

[23] A. Voulodimos, N. Doulamis, A. Doulamis, and E. Protopapadakis. "Deep learning for computer vision: A brief review". In: *Computational intelligence and neuroscience* 2018 (2018).

[24] S. Grigorescu, B. Trasnea, T. Cocias, and G. Macesanu. "A survey of deep learning techniques for autonomous driving". In: *Journal of field robotics* 37.3 (2020), pp. 362–386.

[25] K. Hornik, M. Stinchcombe, and H. White. "Multilayer feedforward networks are universal approximators". In: *Neural networks* 2.5 (1989), pp. 359–366.

[26] I. N. Da Silva, D. Hernane Spatti, R. Andrade Flauzino, L. H. B. Liboni, S. F. dos Reis Alves, I. N. da Silva, D. Hernane Spatti, R. Andrade Flauzino, L. H. B. Liboni, and S. F. dos Reis Alves. *Artificial neural network architectures and training processes*. Springer, 2017.

[27] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor". In: *International conference on machine learning*. PMLR. 2018, pp. 1861–1870.

[28] S. Dasari, F. Ebert, S. Tian, S. Nair, B. Bucher, K. Schmeckpeper, S. Singh, S. Levine, and C. Finn. "Robonet: Large-scale multi-robot learning". In: *arXiv preprint arXiv:1910.11215* (2019).

[29] E. Alpaydin. *Introduction to machine learning*. MIT press, 2020.

[30] X. Glorot and Y. Bengio. "Understanding the difficulty of training deep feedforward neural networks". In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. JMLR Workshop and Conference Proceedings. 2010, pp. 249–256.

[31] Y. Xu and R. Goodacre. "On splitting training and validation set: a comparative study of cross-validation, bootstrap and systematic sampling for estimating the generalization performance of supervised learning". In: *Journal of analysis and testing* 2.3 (2018), pp. 249–262.

[32] L. Perez and J. Wang. "The effectiveness of data augmentation in image classification using deep learning". In: *arXiv preprint arXiv:1712.04621* (2017).

[33] S.-H. Lim, S. R. Young, and R. M. Patton. "An analysis of image storage systems for scalable training of deep neural networks". In: *system* 5.7 (2016), p. 11.

[34] L. Van Der Maaten, E. O. Postma, H. J. van den Herik, et al. "Dimensionality reduction: A comparative review". In: *Journal of Machine Learning Research* 10.66-71 (2009), p. 13.

[35] H. Hotelling. "Analysis of a complex of statistical variables into principal components." In: *Journal of educational psychology* 24.6 (1933), p. 417.

[36] L. Maaten. "Visualizing data using t-SNE". In: *Journal of machine learning research* 9.Nov (2008), p. 2579.

[37] G. E. Hinton and R. R. Salakhutdinov. "Reducing the dimensionality of data with neural networks". In: *science* 313.5786 (2006), pp. 504–507.

[38] M. Petrov and T. Wortmann. *Latent fitness landscapes-exploring performance within the latent space of post-optimization results*. 2021.

[39] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol. "Extracting and composing robust features with denoising autoencoders". In: *Proceedings of the 25th international conference on Machine learning*. 2008, pp. 1096–1103.

[40] D. P. Kingma and M. Welling. "Auto-encoding variational bayes". In: *arXiv preprint arXiv:1312.6114* (2013).

[41] L. Weng. "From autoencoder to beta-vae". In: *lilianweng. github. io/lil-log* (2018).

[42] E. Jang. "A beginner's guide to variational methods: Mean-field approximation, 2016". In: *URL https://blog. evjang. com/2016/08/variational-bayes. html. Accessed* 1.02 (2018).

[43] G. Sejnova, M. Vavrecka, and K. Stepanova. "Bridging Language, Vision and Action: Multimodal VAEs in Robotic Manipulation Tasks". In: *arXiv preprint arXiv:2404.01932* (2024).

[44] S. Rezaei-Shoshtari, D. Meger, and I. Sharf. "Learning the latent space of robot dynamics for cutting interaction inference". In: *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2020, pp. 5627–5632.

[45] N. Dilokthanakul, P. A. Mediano, M. Garnelo, M. C. Lee, H. Salimbeni, K. Arulkumaran, and M. Shanahan. "Deep unsupervised clustering with gaussian mixture variational autoencoders". In: *arXiv preprint arXiv:1611.02648* (2016).

[46] J. L. Elman. "Finding structure in time". In: *Cognitive science* 14.2 (1990), pp. 179–211.

[47] S. Hochreiter and J. Schmidhuber. "Long short-term memory". In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[48] C. Olah et al. "Understanding lstm networks". In: (2015).

[49] L. Graesser and W. L. Keng. *Foundations of deep reinforcement learning*. Addison-Wesley Professional, 2019.

[50] H. Kurniawati. "Partially observable markov decision processes (pomdps) and robotics". In: *arXiv preprint arXiv:2107.07599* (2021).

[51] W. Shi, S. Guo, X. Cong, W. Sheng, J. Yan, and J. Chen. "Frequency Agile Anti-Interference Technology Based on Reinforcement Learning Using Long Short-Term Memory and Multi-Layer Historical Information Observation". In: *Remote Sensing* 15.23 (2023), p. 5467.

[52] H. Nguyen, S. Katt, Y. Xiao, and C. Amato. "On-Robot Bayesian Reinforcement Learning for POMDPs". In: *2023 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2023, pp. 9480–9487.

[53] Z. Qiao, K. Muelling, J. Dolan, P. Palanisamy, and P. Mudalige. "Pomdp and hierarchical options mdp with continuous actions for autonomous driving at intersections". In: *2018 21st International Conference on Intelligent Transportation Systems (ITSC)*. IEEE. 2018, pp. 2377–2382.

[54] X. Li, W. Shang, and S. Cong. "Model-based reinforcement learning for robot control". In: *2020 5th International Conference on Advanced Robotics and Mechatronics (ICARM)*. IEEE. 2020, pp. 300–305.

[55] J. Kober, J. A. Bagnell, and J. Peters. "Reinforcement learning in robotics: A survey". In: *The International Journal of Robotics Research* 32.11 (2013), pp. 1238–1274.

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

[57] N. Vlassis, M. Toussaint, G. Kontes, and S. Piperidis. "Learning model-free robot control by a Monte Carlo EM algorithm". In: *Autonomous Robots* 27 (2009), pp. 123–130.

[58] T. Dam. "Sample Efficient Monte Carlo Tree Search for Robotics". In: (2023).

[59] M. Safeea and P. Neto. "A Q-learning approach to the continuous control problem of robot inverted pendulum balancing". In: *Intelligent Systems with Applications* 21 (2024), p. 200313.

[60] R. J. Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine learning* 8 (1992), pp. 229–256.

[61] D. Kalashnikov, A. Irpan, P. Pastor, J. Ibarz, A. Herzog, E. Jang, D. Quillen, E. Holly, M. Kalakrishnan, V. Vanhoucke, et al. "Scalable deep reinforcement learning for vision-based robotic manipulation". In: *Conference on robot learning*. PMLR. 2018, pp. 651–673.

[62] L. Zhong. "Comparison of Q-learning and SARSA Reinforcement Learning Models on Cliff Walking Problem". In: *2023 International Conference on Data Science, Advanced Algorithm and Intelligent Computing (DAI 2023)*. Atlantis Press. 2024, pp. 207–213.

[63] C. J. C. H. Watkins. "Learning from delayed rewards". In: (1989).

[64] C. J. Watkins and P. Dayan. "Q-learning". In: *Machine learning* 8 (1992), pp. 279–292.

[65] M. Imtiaz, Y. Qiao, and B. Lee. "Comparison of Two Reinforcement Learning Algorithms for Robotic Pick and Place with Non-Visual Sensing". In: *International Journal of Mechanical Engineering and Robotics Research* 10.10 (2021), pp. 526–535.

[66] G. A. Rummery and M. Niranjan. *On-line Q-learning using connectionist systems*. Vol. 37. University of Cambridge, Department of Engineering Cambridge, UK, 1994.

[67] D. Ramachandran and R. Gupta. "Smoothed sarsa: reinforcement learning for robot delivery tasks". In: *2009 IEEE International Conference on Robotics and Automation*. IEEE. 2009, pp. 2125–2132.

[68] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. "Playing atari with deep reinforcement learning". In: *arXiv preprint arXiv:1312.5602* (2013).

[69] M. Sumanas, A. Petronis, V. Bucinskas, A. Dzedzickis, D. Virzonis, and I. Morkvenaite-Vilkonciene. "Deep Q-learning in robotics: Improvement of accuracy and repeatability". In: *Sensors* 22.10 (2022), p. 3911.

[70] N. Khlif, K. Nahla, and B. Safya. "Reinforcement learning with modified exploration strategy for mobile robot path planning". In: *Robotica* 41.9 (2023), pp. 2688–2702.

[71] H. Hasselt. "Double Q-learning". In: *Advances in neural information processing systems* 23 (2010).

[72] H. Van Hasselt, A. Guez, and D. Silver. "Deep reinforcement learning with double q-learning". In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 30. 1. 2016.

[73] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. "Prioritized experience replay". In: *arXiv preprint arXiv:1511.05952* (2015).

[74] Y. Wang, Y. Fang, P. Lou, J. Yan, and N. Liu. "Deep reinforcement learning based path planning for mobile robot in unknown environment". In: *Journal of Physics: Conference Series*. Vol. 1576. 1. IOP Publishing. 2020, p. 012009.

[75] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel. "High-dimensional continuous control using generalized advantage estimation". In: *arXiv preprint arXiv:1506.02438* (2015).

[76] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. "Asynchronous methods for deep reinforcement learning". In: *International conference on machine learning*. PMLR. 2016, pp. 1928–1937.

[77] Y. Sasaki, S. Matsuo, A. Kanezaki, and H. Takemura. "A3C based motion learning for an autonomous mobile robot in crowds". In: *2019 IEEE International Conference on Systems, Man and Cybernetics (SMC)*. IEEE. 2019, pp. 1036–1042.

[78] C. Zhou, B. Huang, and P. Fränti. "An advantage actor-critic algorithm for robotic motion planning in dense and dynamic scenarios". In: *arXiv preprint arXiv:2102.03138* (2021).

[79] G. Grandesso, E. Alboni, G. P. R. Papini, P. M. Wensing, and A. Del Prete. "CACTO: Continuous actor-critic with trajectory optimization—towards global optimality". In: *IEEE Robotics and Automation Letters* (2023).

[80] T. Haarnoja, H. Tang, P. Abbeel, and S. Levine. "Reinforcement learning with deep energy-based policies". In: *International conference on machine learning*. PMLR. 2017, pp. 1352–1361.

[81] H. Ji and C. Yin. "Application of Soft Actor-Critic Reinforcement Learning to a Search and Rescue Task for Humanoid Robots". In: *2022 China Automation Congress (CAC)*. IEEE. 2022, pp. 3954–3960.

[82] R. Bellman. "The theory of dynamic programming". In: *Bulletin of the American Mathematical Society* 60.6 (1954), pp. 503–515.

[83] N. Bougie and R. Ichise. "Hierarchical learning from human preferences and curiosity". In: *Applied Intelligence* 52.7 (2022), pp. 7459–7479.

[84] M. Saveriano, F. J. Abu-Dakka, A. Kramberger, and L. Peternel. "Dynamic movement primitives in robotics: A tutorial survey". In: *The International Journal of Robotics Research* 42.13 (2023), pp. 1133–1184.

[85] Y. Zhou, J. Gao, and T. Asfour. "Learning via-point movement primitives with inter- and extrapolation capabilities". In: *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE. 2019, pp. 4301–4308.

[86] A. Nagabandi, I. Clavera, S. Liu, R. S. Fearing, P. Abbeel, S. Levine, and C. Finn. "Learning to adapt in dynamic, real-world environments through meta-reinforcement learning". In: *arXiv preprint arXiv:1803.11347* (2018).

[87] J. Liu, D. Wang, S. Yu, X. Li, Z. Han, and Y. Tang. "A survey of image clustering: Taxonomy and recent methods". In: *2021 IEEE International Conference on Real-time Computing and Robotics (RCAR)*. IEEE. 2021, pp. 375–380.

[88] Q. Bsoul, J. Salim, and L. Q. Zakaria. "An intelligent document clustering approach to detect crime patterns". In: *Procedia Technology* 11 (2013), pp. 1181–1187.

[89] M. Wang and W. Deng. "Deep face recognition with clustering based domain adaptation". In: *Neurocomputing* 393 (2020), pp. 1–14.

[90] A. Ghosal, A. Nandy, A. K. Das, S. Goswami, and M. Panday. "A short review on different clustering techniques and their applications". In: *Emerging Technology in Modelling and Graphics: Proceedings of IEM Graph 2018* (2020), pp. 69–83.

[91] A. E. Ezugwu, A. M. Ikotun, O. O. Oyelade, L. Abualigah, J. O. Agushaka, C. I. Eke, and A. A. Akinyelu. "A comprehensive survey of clustering algorithms: State-of-the-art machine learning applications, taxonomy, challenges, and future research prospects". In: *Engineering Applications of Artificial Intelligence* 110 (2022), p. 104743.

[92] C. Fraley and A. E. Raftery. "How many clusters? Which clustering method? Answers via model-based cluster analysis". In: *The computer journal* 41.8 (1998), pp. 578–588.

[93] R. M. Neal and G. E. Hinton. "A view of the EM algorithm that justifies incremental, sparse, and other variants". In: *Learning in graphical models*. Springer, 1998, pp. 355–368.

[94] C. J. Wu. "On the convergence properties of the EM algorithm". In: *The Annals of statistics* (1983), pp. 95–103.

[95] S. Lloyd. "Least squares quantization in PCM". In: *IEEE transactions on information theory* 28.2 (1982), pp. 129–137.

[96] X. Guo and Y. Zhai. "K-means clustering based reinforcement learning algorithm for automatic control in robots". In: *Int. J. Simul. Syst. Sci. Technol* 17 (2016), p. 24.

[97] J. C. Dunn. "A fuzzy relative of the ISODATA process and its use in detecting compact well-separated clusters". In: (1973).

[98] R. J. Moreno and D. J. Lopez. "Trajectory planning for a robotic mobile using fuzzy c-means and machine vision". In: *Symposium of Signals, Images and Artificial Vision-2013: STSIVA-2013*. IEEE. 2013, pp. 1–4.

[99] H. Kuyuk, E. Yildirim, E. Dogan, and G. Horasan. "Application of k-means and Gaussian mixture model for classification of seismic activities in Istanbul". In: *Nonlinear Processes in Geophysics* 19.4 (2012), pp. 411–419.

[100] B. Ertl, J. Meyer, A. Streit, and M. Schneider. "Application of Mixtures of Gaussians for Tracking Clusters in Spatio-temporal Data." In: *KDIR*. 2019, pp. 45–54.

[101] E. Patel and D. S. Kushwaha. "Clustering cloud workloads: K-means vs gaussian mixture model". In: *Procedia computer science* 171 (2020), pp. 158–167.

[102] Y. Zhou, J. Gao, and T. Asfour. "Movement primitive learning and generalization: Using mixture density networks". In: *IEEE Robotics & Automation Magazine* 27.2 (2020), pp. 22–32.

[103] C. E. Reiley, E. Plaku, and G. D. Hager. "Motion generation of robotic surgical tasks: Learning from expert demonstrations". In: *2010 Annual international conference of the IEEE engineering in medicine and biology*. IEEE. 2010, pp. 967–970.

[104] M. C. Hughes and E. Sudderth. "Memoized online variational inference for Dirichlet process mixture models". In: *Advances in neural information processing systems* 26 (2013).

[105] Y. Li, E. Schofield, and M. Gönen. "A tutorial on Dirichlet process mixture modeling". In: *Journal of mathematical psychology* 91 (2019), pp. 128–144.

[106] M. Moustapha and B. Sudret. "Learning non-stationary and discontinuous functions using clustering, classification and Gaussian process modelling". In: *Computers & Structures* 281 (2023), p. 107035.

[107] T. S. Ferguson. "A Bayesian analysis of some nonparametric problems". In: *The annals of statistics* (1973), pp. 209–230.

[108] J. Sethuraman. "A constructive definition of Dirichlet priors". In: *Statistica sinica* (1994), pp. 639–650.

[109] S. J. Gershman and D. M. Blei. "A tutorial on Bayesian nonparametric models". In: *Journal of Mathematical Psychology* 56.1 (2012), pp. 1–12.

[110] R. M. Neal. "Markov chain sampling methods for Dirichlet process mixture models". In: *Journal of computational and graphical statistics* 9.2 (2000), pp. 249–265.

[111] D. M. Blei and M. I. Jordan. "Variational inference for Dirichlet process mixtures". In: (2006).

[112] Y. Teh, K. Kurihara, and M. Welling. "Collapsed variational inference for HDP". In: *Advances in neural information processing systems* 20 (2007).

[113] Y. Lin and R. Jiang. "Basic Statistics for Bioinformatics". In: *Basics of Bioinformatics: Lecture Notes of the Graduate Summer School on Bioinformatics of China*. Springer, 2013, pp. 27–68.

[114] S. Fujimoto, D. Meger, and D. Precup. "Off-policy deep reinforcement learning without exploration". In: *International conference on machine learning*. PMLR. 2019, pp. 2052–2062.

[115] Y. Wu, G. Tucker, and O. Nachum. "Behavior regularized offline reinforcement learning". In: *arXiv preprint arXiv:1911.11361* (2019).

[116] J. Fu, A. Kumar, O. Nachum, G. Tucker, and S. Levine. "D4rl: Datasets for deep data-driven reinforcement learning". In: *arXiv preprint arXiv:2004.07219* (2020).

[117] A. Gupta, V. Kumar, C. Lynch, S. Levine, and K. Hausman. "Relay policy learning: Solving long-horizon tasks via imitation and reinforcement learning". In: *arXiv preprint arXiv:1910.11956* (2019).

[118] Y. Ren, J. Pu, Z. Yang, J. Xu, G. Li, X. Pu, P. S. Yu, and L. He. "Deep clustering: A comprehensive survey". In: *arXiv preprint arXiv:2210.04142* (2022).

[119] C. Grazian. "A review on Bayesian model-based clustering". In: *arXiv preprint arXiv: 2303.17182* (2023).

[120] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. "Automatic differentiation in pytorch". In: (2017).

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

[122] E. Todorov, T. Erez, and Y. Tassa. "Mujoco: A physics engine for model-based control". In: *2012 IEEE/RSJ international conference on intelligent robots and systems*. IEEE. 2012, pp. 5026–5033.

[123] G. E. Hinton and S. Roweis. "Stochastic neighbor embedding". In: *Advances in neural information processing systems* 15 (2002).

[124] K. Fukunaga. *Introduction to statistical pattern recognition*. Elsevier, 2013.

[125] C. Yang, H. Ma, and M. Fu. *Advanced technologies in modern robotic applications*. Springer, 2016.

[126] W. Zhu, X. Guo, D. Owaki, K. Kutsuzawa, and M. Hayashibe. "A survey of sim-to-real transfer techniques applied to reinforcement learning for bioinspired robots". In: *IEEE Transactions on Neural Networks and Learning Systems* 34.7 (2021), pp. 3444–3459.

[127] M. D. Binder, N. Hirokawa, U. Windhorst, et al. *Encyclopedia of neuroscience*. Vol. 3166. Springer Berlin, Germany, 2009.

[128] L. Strother, W. P. Medendorp, A. M. Coros, and T. Vilis. "Double representation of the wrist and elbow in human motor cortex". In: *European Journal of Neuroscience* 36.9 (2012), pp. 3291–3298.

[129] A. Errante and L. Fogassi. "Parieto-frontal mechanisms underlying observation of complex hand-object manipulation". In: *Scientific reports* 9.1 (2019), p. 348.

[130] J. D. Meier, T. N. Aflalo, S. Kastner, and M. S. Graziano. "Complex organization of human primary motor cortex: a high-resolution fMRI study". In: *Journal of neurophysiology* 100.4 (2008), pp. 1800–1812.