Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

FAKULTÄT
FÜR MATHEMATIK, INFORMATIK
UND NATURWISSENSCHAFTEN

Master Thesis

Decentralized Information Gathering with Continuous and very large Discrete Observation Spaces

Caus Danu

danu.caus@studium.uni-hamburg.de
Study Program: Intelligent Adaptive Systems
Matr.-Nr.: 7014833

Primary Supervisor: Dr. Mikko Lauri
Secondary Supervisor: Prof. Dr. Simone Frintrop

Submission: June 2020

Abstract

Joint, end-to-end training of perception and planning systems is a challenging, but promising small step in the direction of Artificial General Intelligence (AGI). Scaling to multiple cooperating agents is an extra step in this direction and addresses problems that require team work to be solved efficiently, such as rescue missions for instance. This work introduces COGNet-DICE, a new algorithm designed to integrate perception of discrete and continuous signals with planning in a decentralized multi-agent setting. The planning is based on the Graph-based Direct Cross-Entropy method for policy search, also known as the G-DICE evolutionary algorithm, while perception is ensured by neural nets embedded into the graph nodes. The new algorithm scales well to very high-dimensional inputs and multiple agents with the help of Variational Autoencoders, that provide input compression, as well as training dataset augmentation and robustness to noise. We show that the algorithm reaches state of the art performance on signal source localization problems.

Zusammenfassung

Das Ende-zu-Ende Verbundtraining von Wahrnehmungs- und Planungssystemen ist ein ebenso herausfordernder wie vielversprechender Schritt in Richtung allgemeine künstliche Intelligenz. Die Skalierung auf mehrere kooperierende Agenten ist ein weiterer Schritt in diese Richtung, wobei hierdurch Probleme adressiert werden, die auf Team-Work basieren. Hierzu gehören unter anderem Rettungsmissionen. In dieser Arbeit wird COGNet-DICE vorgestellt. Dies ist ein neuartiger Algorithmus, der die Wahrnehmung von diskreten und kontinuierlichen Signalen in einem dezentralisiertem Multi-Agenten Aufbau mit Planung integriert. Die Planung basiert auf der Graph-based Direct Cross-Entropy Methode für Strategiesuche, auch bekannt als G-DICE evolutionärer Algorithmus, wobei der Wahrnehmungsaspekt durch neuronale Netze übernommen wird, welche in den einzelnen Knoten eingebettet sind. Der neuartige Algorithmus skaliert sehr gut auf hochdimensionale Eingangsgrößen und mehrere Agenten durch die Verwendung von Variations-Autoencodern, welche Eingangskompression bieten, sowie Trainingsdatensatz Vergrößerung und eine hohe Rauschtoleranz. Wir zeigen, dass der Algorithmus Spitzenleistung nach dem aktuellen Stand der Technik bei Signalquellenlokalisierung erreicht.

Contents

List of Figures

List of Tables

List of Algorithms

1 Introduction

Planning is the ability to come up with a sequence of well thought actions in order to achieve an end goal. Perception is the ability to interpret the environment using different sensory information: visual, auditory, touch, such that these inform us of the state we are in towards the goal we strive to achieve. Everything we people do seems to involve both. In the process of planning and perceiving we learn a new task by building a behaviour we technically call a policy, or action policy. We plan what to do and then we receive some perceptual feedback. Based on the feedback we update our plan and may act differently next time, when we may perceive better outcomes with different visual, auditory and haptic cues. We even plan how to perceive better, not only what happened as a result of an action.

The state of the art in artificial intelligence (AI) today is such that we treat different systemic blocks separately: there is an architecture for vision problems, a separate architecture for language problems, separate treatment of planning, motor skills etc. Real life evidence suggests that they are in fact very closely tied together with ambiguous boundaries. For example, there is a close connection between vision and planning as suggested by [Law and Gold, 2009]. Therefore, more effort needs to be invested in integrating these different modalities together in order to emulate in an artificial agent the complex activities that we people do with ease. An added layer of complexity that evolution created through many iterations is the fact that different organisms that possess a very good integration of all these modalities to begin with, effectively work together in teams and cooperate, resulting in a compounded outcome.

This work will shed more light on the integration between the perception of continuous signals and planning of actions based on these perceptions. Moreover, this thesis will scale the proposed solution to vision input, which is arguably the most important and informative sense out of all. We will assume a multi-agent setting all throughout the thesis, which is an important step towards the hierarchical complexity that the natural world exhibits.

1.1 Integrated continuous perception and planning with high dimensions

The problem addressed in this thesis consists in finding a new way to perceive and plan at the same time. At first, it may seem to be a control theory problem in the sense that we need to come up with a sequence of actions to control the outcome of a task. However, there is a small but important nuance: in control theory the problems are treated as if the equation governing a certain process is already known and the task is to find good solutions that satisfy the equation. Since we also introduce the aspect of creating the model equation ourselves, we enter the reinforcement learning (RL) realm. The emphasis falls therefore on perception as well, along with learning how to act. In the perception phase the agent learns a model from the sensory data it receives as a result of the actions it takes. The idea is not only to be given a model and learn to act based on it, but rather to come up with the model itself about the physical world and based on it do the planning and find sequences of actions that result in a high performance score.

Up until today there was not much progress made in adapting perception to real world robotics applications. Our world is intrinsically continuous and therefore the perceptual block should be able to work with continuous real numbers, and not just discretized and quantized values. Secondly, even if it is convenient to work with large grids of data, like it is the case of computer vision, it is necessary to find a principled approach to compress them without much loss and solve the scaling issues when many agents have to analyze very high-dimensional discrete data.

In summary, the problem treated in this work falls at the intersection between control theory, reinforcement learning, computer vision and additionally game theory, because of the multiple agents

involved. The aim is to make artificial agents intelligently classify a continuous signal and efficiently act based on it. A second goal is to have the same algorithm also work with discrete data, but very-high dimensional and for a multi-agent setting. Another important aspect is to have a generic a framework as possible, agnostic to the type of task the agent might encounter, hence using a neural network that can adapt to the actions that need to be performed, which may be different for different tasks, and the changing continuous input associated with performing these actions.

1.2 Technical Applications

One of the most prominent domains in which planning algorithms with continuous visual inputs are useful is the field of rescue missions. In particular, we can mention Unmanned Air Vehicles, or UAVs and Unmanned Underwater Vehicles, or UUVs. With the advent of cheaper technology and manufacturing materials, these types of agents can be mass-produced and used to cooperate in all sorts of missions that require prompt and efficient scanning of large areas.

As a concrete example can serve the crashing of various planes either on land, or especially in the ocean waters. In many cases it took a long time to localize the accident sites and some of them were not found at all because of too much wasted time in the first critical moments after the crash, which led to the debris being carried away intractably by the water currents. Surprisingly, even in relatively small areas, such as a mountain range, it is hard to conduct search missions like these.

Another example can be nuclear accidents or other accidents resulting in toxic chemical substances being released and making surveillance missions by people very hard or even impossible. Such situations make the search task even harder because of the added noise. A relevant example at the moment of writing this thesis is the Fukushima Daiichi Nuclear Power Plant disaster. Up to this date, there is no concrete localization of the melted core waste, which poses a serious ecological threat to the underground water supply amongst other things. The mission is extremely hard not only because of the high radiation level which limits the time robots can get exposed to it, but also because of the associated noise this injects into the stream of data perceived by the robot sensors. If many agents were working together, such as drones analyzing the heatmaps and radiation levels in the accident vicinity, the problem can be at least partially mitigated. More agents means less exposure time per agent, and different quality of individual observations for different agents, which can potentially result in a better quality merged outcome with less noise overall.

Another important and technically feasible application is the cleansing of ocean water supplies using automatic patrol agents that swipe large areas and detect polluting waste, such as petrol spills for example, or solid debris. One might easily consider in this case various heterogeneous UUVs working together: some smaller agents quickly scanning and finding waste, and others with a larger storage capacity loading and carrying it to a collection point.

From an archaeological point of view, underwater patrolling can also be very useful, since it definitely increases the chance of finding valuable artifacts, tools and treasures, whose existence is suggested by historical documents, but were never physically found. Closely related to this is also the exploration of wild life and discovery of new organisms leaving at very high depth, where light is scarce and pressure is very high.

1.3 Biological Motivation

Artificial Intelligence is a broad field with many aspects. This mainly stems from the fact that intelligence itself is a generic term that implies actually many different "types of intelligence". The main canonical types can be considered to be: **Logic**, performed by neurons located in the frontal lobe of the human brain; **Vision**, with dedicated neurons located in the Occipital lobe of the human brain; **Language**, mainly tied with the temporal lobe, and including regions such as: "Broca's area" for lexical intelligence and "Wernicke's area" for grammatical and phonological intelligence. Their physical location is illustrated in Figure 1.1

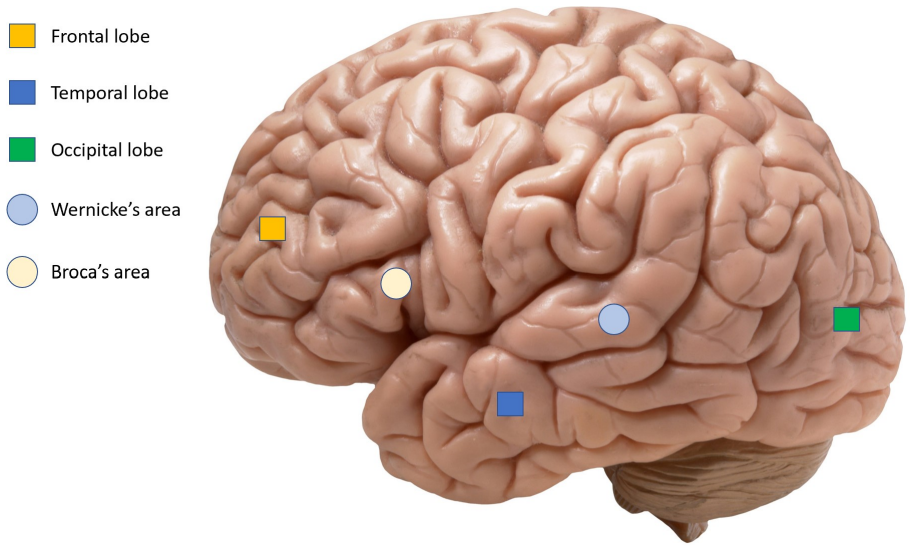


Figure 1.1: Brain centers for different types of intelligence. Notice how planning (Frontal lobe) and vision (Occipital lobe) occupy opposing regions of the brain. Nevertheless, we humans use both intricately combined in order to perform the simplest of tasks.

When referring to logic, it is an ambiguous notion in itself, having connotations with other terms like: mathematical reasoning, planning, decision making etc. In this work we will tackle decision making and think of it as the ability to plan a sequence of future actions in order to optimize an end result. We will also embed vision data in the planning process and emulate an end-to-end training procedure mixing both aspects together.

In the biological realm, people rely very much on planning and vision in doing the simplest of things, like hand-eye coordination for example. In complex activities like various sports for instance, the interactions between the two is vital. The same can therefore be said for the case of artificial agents, especially when talking about rescue agents such as Unmanned Air Vehicles (UAVs) and Unmanned Underwater Vehicles (UUVs), where optimal actions and reliable observations are very important for a fast and efficient handling of complicated, time-critical missions.

One relevant and interesting example that ties both vision and planning literally together is the phenomenon of saccades. A saccade is a rapid eye movement through which humans sample information from the environment. In other words, the brain has certain algorithms that plan eye movements in order to capture vision data that will subsequently be used to plan further actions. For example, the reader is encouraged to move her eyes from left to right a few times without fixating gaze at any object in her field of vision, and then try to track with the eyes her own finger moving from left to right. The reader will hopefully notice how the eyes move much more smoothly while trying to track the finger, and much more abruptly without an object to track. This is a nice example of what goes under the name of **Active Vision**, in other words: planning on where to look in order to get the most information from the environment. Interestingly, the example shows how the brain evolved to use different circuits and algorithms in order to plan eye movements and it speaks about the importance of planning where to look in order to acquire vision data. Incidentally, this example illustrates what goes in the literature under the name of **bottom-up attention**, a kind of involuntary attention where the eyes are attracted to certain predefined cues like color-contrast, luminosity-contrast, movement (which is the main cue in this example) etc.

Afterwards, how to use vision data to guide other actions, such as a limb movement for instance, is a different, higher-level planning problem. For example, how exactly to move a hand in order to grasp a cup involves the mechanism of the so called **top-down attention**, where the brain guides the eyes where to look in order to see both the cup and the hand, and gives correction feedback to the hand such that it is being correctly positioned before attempting a grasp. All of this is a complex planning experiment with vision feedback, and there are many other examples related to such top-down attention mechanisms.

The above examples are meant to persuade the reader of the intricate relation between planning and

computer vision in an artificial agent, much like in a biological setting. In the following chapters we will illustrate how this can be achieved from a mathematical point of view. First we will put emphasis on planning. Then, by scaling our algorithm and framework to deal with very high-dimensional data, we will be able to consider vision data as input to our planning algorithm.

1.4 Core concepts made easy

This section will provide a high-level, gentle introduction into the essential concepts necessary for further sections. In order to understand how planning is done, Decentralized Partially Observable Markov Decision Processes are emphasized as the core framework. To be able to address continuous signals instead of merely discrete ones, the concept of learning is introduced in relation to information theory and the notion of entropy. Compression schemes from deep learning will also prove to be useful for scaling the solution to very high-dimensional problems with many agents and hence, a separate section will be dedicated to Autoencoders and Variational Autoencoders.

1.4.1 Graphical planning models

One of the ways in which planning can be reasoned about is with the help of graphical models. There are 2 major directions when it comes to these models: Bayesian networks and Markov blankets.

Bayesian networks are used in close connection with Bayesian statistics. They consist of nodes connected by edges, where each node will represent the computation of a conditional or unconditional probability distribution that will subsequently be multiplied by other probabilities from other connected nodes in order to come up with a new metric. An illustration is provided in Figure 1.2. As we can see, this network can be conceptualized as a decision tree. We can use it in many ways: to judge the utility of following different branches, to prune some of the branches all together if they seem not likely to lead to good outcomes, to add new leafs and explore new policies, etc.

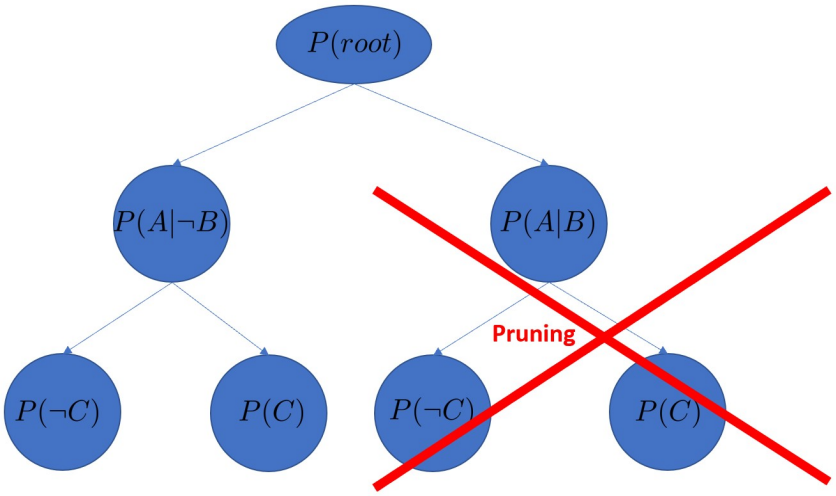


Figure 1.2: Bayesian Network representation. By following the right conditions $Cond_i$ and/or negation of the conditions i.e. $\neg Cond_i$ we will build a decision branch consisting of a sequence of decisions. We can design the tree in such a way that each visited node is associated with a probability metric which will help in creating good decision branches to explore or exploit and ignore/prune sequences that are unlikely to result in useful outcomes.

The second graphical model: Markov blankets is closely related to what we call Markov Decision Processes, or MDPs. A Markov decision process is a framework for modeling decision making very popular in robotics. It consists of states of an agent represented by graph nodes and control actions an agent can issue in order to act on the surrounding environment. Once the agent acts, the environment

reacts with a response in the form of a reward R. Figure 1.3 represents the setting we assume, where the agent can act upon the world with its actuators and then receive a reward back.

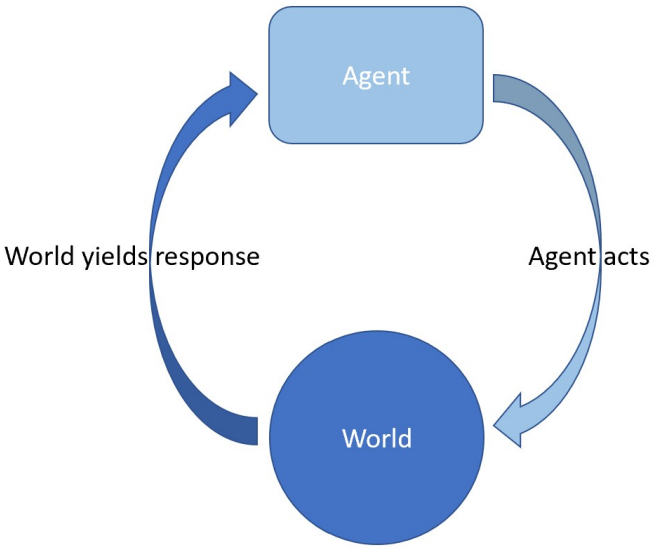


Figure 1.3: Agent interaction with the world. Agent has partial control by acting on the world. The world on the other hand reacts and the agent takes note of this response through the reward it receives.

What is peculiar for MDPs is that the states are uniquely identifiable, i.e. there is no uncertainty about the state of the decision process. A natural question might arise, namely: What happens when the state is not uniquely identifiable ? For example: an agent might move around and see many doors in a corridor that look the same. How will it know exactly where it is localized along this corridor ? In such situations, a partially observable Markov Decision Process or POMDP is particularly useful. A POMDP complicates and enhances the MDP by assuming the observations received by the agent are noisy and uncertain. This effectively means that we can no longer observe the state, but rather have a belief about what this state might be with a certain probability. In other words, we have a statistical distribution over the potential states of an agent, which we call a "belief". Figure 1.4 illustrates how an agent perceives, acts and builds a belief about what state the environment might be in. As one can see, this model is particularly useful for representing an artificial agent that has actuators for acting in the world, sensors for measuring observations and a belief over potential internal states S at different points in time.

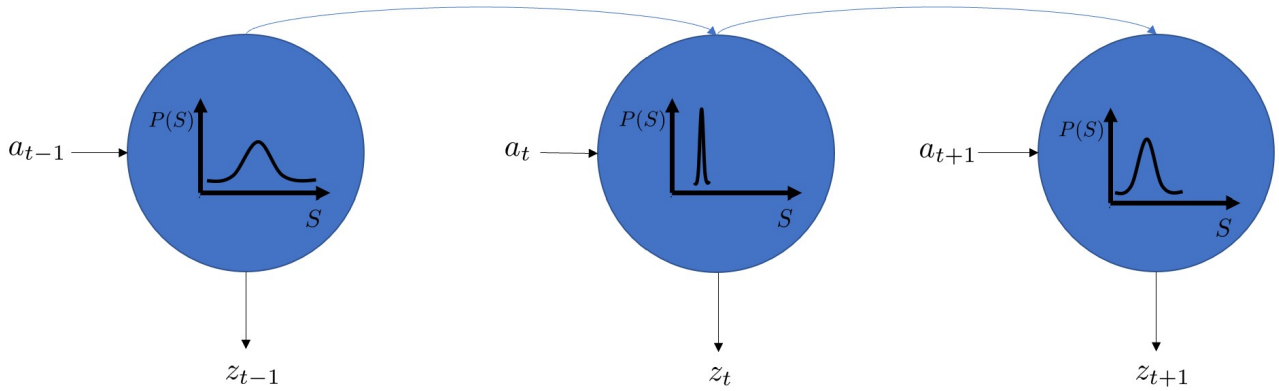


Figure 1.4: Graphical representation of the development of states, measurements and controls. A circle represents the belief about what the internal state of the world is, namely $P(S)$. The agent receives observations z_t through its sensors and acts on the environment through control actions a_t in order to transition to the next internal state.

Going one step further to consider more than one agent, we end up with a new framework: "Decentralized Partially Observable Markov Decision Process", or Dec-POMDP. The aim of the multiple agents is to work together in order to achieve a global goal. It is important to emphasize that all the agents receive the same joint reward. This aspect is to be contrasted with the broader field of Game Theory, where each agent has its own individual reward. Because of the individual rewards, we can say that agents are self-centered, trying to maximize their own score. In the case of Dec-POMDPs however, the agents are cooperating, since the actions they take individually will affect the common reward they all receive in the end. To reiterate and summarize, Decentralized Partially Observable Markov Decision Processes represent a way of reasoning about multiple agents planning and cooperating together, when there is uncertainty in the observations they receive, as well as in the outcome of the actions they take.

Our framework of reasoning in this thesis will be the Dec-POMDP and therefore a more formal mathematical description is required and will be given in the next chapter.

1.4.2 Frequentist vs Bayesian approach to learning

In Machine Learning there are two major approaches to learning that stand out: the Frequentist approach and the Bayesian approach. In the Frequentist approach we assume that the model we are trying to learn follows a unique underlying distribution, parametrized by a true θ_0. This is to be contrasted with the Bayesian approach to learning, where there is no true underlying and unique θ_0 parameter, but rather a distribution over many potential θ's, and therefore many potential hypotheses that can explain the same training data. In a Bayesian setting we try to compute a so called posterior distribution by applying the Bayesian formula of inverting probabilities:

$$P(\theta|Data) = \frac{P(Data|\theta) * P(\theta)}{P(Data)} \tag{1.1}$$

This formula says that in order to compute the most likely model parameter θ given the seen $Data$, we need a prior distribution over the model parameter θ, i.e. $P(\theta)$ given by previous estimations and experiments, and a likelihood of that $Data$ given the model parameter θ, i.e. $P(Data|\theta)$. Additionally, the probability of the entire data under all possible model parameters θ needs to be computed, i.e. $P(Data)$ and used as a normalization factor.

The last step in the Bayesian setting, namely calculating the normalization factor, is a big computational inconvenience. In this work however, we will make use of a procedure called Maximum Likelihood Estimation (short MLE), which is a tool for finding the model parameter $\hat{\theta}$ in the Frequentist approach to learning. Mathematically, the maximum likelihood estimate of a model parameter θ is obtained by maximizing the log-likelihood of seeing the training data.

1.4.3 How to measure learning

To be able to quantify learning there has to be a metric of some kind that describes the acquisition of "learning units". Information can be thought of as a metric directly proportional to the amount of these learning units, which are referred to as bits. Entropy on the other hand would be something inversely proportional to this metric. Semantically, it is useful to associate the notion of entropy with uncertainty. In other words, entropy is the opposite of information. Gaining information is basically minimizing uncertainty, and therefore minimizing the entropy within a system. The notion of entropy originally appeared in physics, e.g. in thermodynamics or quantum mechanics, and was used to measure how chaotic a system is; for example how chaotic is the movement of atoms when temperature changes. In this sense, it is also a measure of uncertainty or lack of predictability.

In summary, when a system is predictable we say there is a great amount of information we learned, measured in bits, that allows us to predict its behaviour. To the contrary, when something is chaotic, we say that learning did not occur and that the entropy is high. Consequently, we can judge how well a system learns something by observing how these metrics, namely information and entropy change over time.

1.4.4 Compressing perception

To be able to scale the solution to the problem to higher dimensions there needs to be a way of approximately solving the addressed task, rather than in an exact manner with raw inputs and brute force. One step in approximating the solution is accomplished in the planning phase, when only a small part of the best policies are kept for the policy improvement step and the rest are temporarily pruned from the tree of potential branches to follow (see Figure 1.2 for intuition about pruning). Another approximating step is executed in the perception phase. Here the task is to perform a compression of the input signal. There are many compression algorithms that can do the job, either lossy or lossless, such as Principal Component Analysis for example, or PCA. However, one of the most state of the art ways in deep learning is to use an Autoencoder (see Figure 1.5). An Autoencoder is a neural network consisting of two other blocks: an encoder and a decoder. The encoder tries to bring the input into a lower-dimensional space, whereas the decoder tries to reconstruct the input using the lower-dimensional variable created by the encoder. In the training process, the compressed variable which will be denoted as L becomes more and more information dense, such that the reconstructed output gets a closer resemblance to the input with each new training step. An Autoencoder, unlike other compression schemes, has the advantage that it can decide on its own what features are important for different datasets.

In the algorithm proposed by this thesis, the perception block will leverage neural networks for interpreting continuous inputs. It is a well known fact that neural nets in general, and hence the nets we will design as building blocks of the new algorithm, are very much data driven, requiring a lot of training items to get good performance. For this reason, instead of an Autoencoder, it is better to use a so called Variational Autoencoder that can not only compress, but also augment the training dataset. A Variational Autoencoder has the added feature of a prior, typically in the form of a Gaussian distribution, that allows the decoder to stochastically recreate the input with the same sampled latent variable L. In other words, many small variations of the input are created in the process of reconstructing the input. This is very useful in augmenting training data and as a result: coping with noisy environments, when the agents take real world measurements that do not look exactly the same as the inputs used in the training simulator. In case of transfer learning from a controlled and constrained simulation environment to real moving robots, this type of variance in what the agent could measure in practice is very useful.

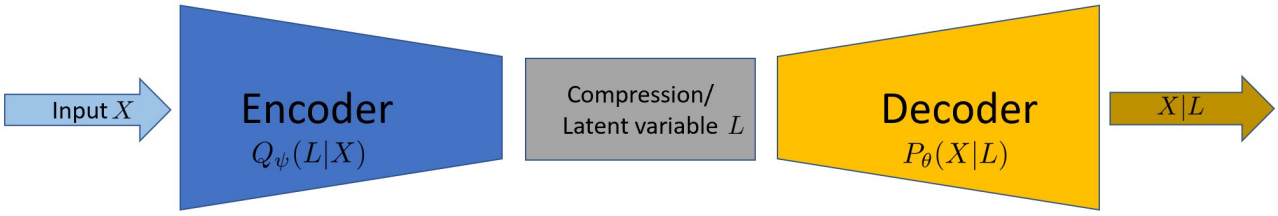


Figure 1.5: (Variational) Autoencoder structure. Two neural networks: an encoder that compresses the input, and a decoder that decompresses it, are trained together in order to create a useful latent representation of the input.

1.5 Overview of previous solutions

The problem of perceiving and planning involves two aspects: how to interpret the observation space, and how to use these observations in planning good actions to take over a sequence of time steps.

Related to observations it is worth mentioning that most of the prior work addresses discrete spaces rather than continuous ones. In many past works the observations were binned/discretized into uniform grids with quantized values. One problem is that both binning and quantizing can throw away important information. Also, if the resolution of the observation grid is too high, the policy search algorithm can easily blow out of proportion, having too many cells to consider.

Various analytical techniques made it possible to tackle continuous spaces as well. Machine learning classifiers such as radial basis functions and beta distributions were used to cluster similar continuous observations into non-uniform grids. Such clustering results in a more compact semantic understanding of the observation space, and it is helpful during online planning for the policy search algorithm to deal with a smaller group of clustered items rather than having to iterate exhaustively through many cells in high resolution grids.

On the planning side the problem boils down to searching efficiently in a huge action space. Various computer science algorithms, accompanied by scalable data structures are key to managing the high number of policy combinations to choose from. They can be divided into exact algorithms that guarantee finding an optimal solution, and approximate algorithms that may find very good, but sub-optimal solutions. To find optimal solutions in a planning problem there's a need for a good heuristic. It was proven that, provided a good heuristic for a single agent scenario, such an optimal algorithm is A*. A counterpart for the multi-agent case was also developed and naturally entitled Multiagent A*.

Another important approach developed in the past was related to Game Theory and consisted in finding such a policy for an agent that satisfies a so called Pareto optimal outcome, i.e. there doesn't exist any other more advantageous action for one agent without making the situation worse for another agent. Reaching this type of equilibrium is an important step in achieving overall efficiency, because all agents do not act in the detriment of one another.

Immediately related to the work of this thesis is the cross-entropy method for policy search. It consists in sampling and evaluating many action-observation pairs and keeping only a small subset of the best performing ones for policy improvements. The cross-entropy approach can have different implementations. The one we are considering in this thesis is the graph-based implementation. It is illustrated in Figure 1.6. The idea is to have a graph for each agent with nodes such as the one presented below. Each node has two main components embedded in it: an action distribution and an observation classifier. They have to be learned through various machine learning techniques. Other works have done this through machine learning methods available prior to the deep learning era. In this thesis however we will use deep learning techniques, i.e. we will embed neural networks in each node of the graphs, as well as have a global Variational Autoencoder that each node can use for augmenting its training dataset.

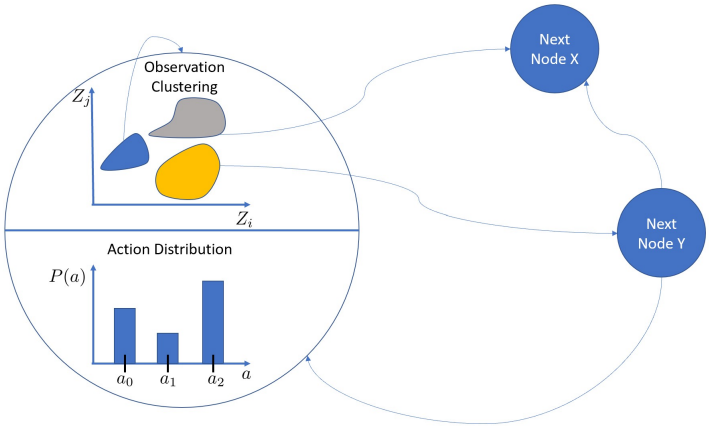


Figure 1.6: Graph implementation of the Cross-Entropy method with continuous observations: a particular node in the graph is responsible for learning an action distribution, as well as learning how to cluster continuous observations and how to transition to next nodes.

1.6 New research questions

What other works in the same field have been doing is to pick a generic model for input clustering and find the best parameters of it through maximum likelihood estimation. For example [Clark-Turner and Amato, 2017] has researched a beta distribution for observation classification, while [Omidshafiei et al., 2017] investigated the radial basis function. In this work we will departure from the model

based approach, and assume we have a generic neural network capable of modelling any function. Theoretically any non-linear function can be modeled by an infinite sum of polynomials as it is the case with Taylor and Maclaurin Series. Fourier Series picture a similar outcome with the help of sinusoids instead of polynomials. More generally, a neural network should be able to learn a linear relation of non-linear basis functions and there should be no need to make assumptions about any a priori models. If however there is still such a need, there is always the possibility to define a prior layer within the network.

Neural nets offer several advantages in the context of our problem. In a discrete version of the problem, the input is simply discretized into bins, which is very inefficient, potentially leading to important information being lost. A neural net on the other hand is capable of extracting the important features and ignore the irrelevant ones. Assumptions about what model the input classifier should follow (ex: radial basis function, beta function, Dirichlet etc.) can also lead to insufficiently correct clustering of the observations. A neural network however, is a universal input classifier that can approximate any underlying model. A third convenience is that a neural net has the added benefit of directly coupling between the learned input features and what action to take.

Another thread of thought is that neural networks can also serve as a way of adaptation to high-dimensional problems. They can specifically be modeled as a function approximator in a lower-dimensional space, process known as compression. Even if the compression is lossy, i.e. results in information being leaked, we can consider this a feature: it helps the system ignore redundant information and not get distracted by noise. Variational Autoencoders are a type of such compressing neural nets. They encode the input into a latent proxy that offers better robustness to noise. In other words, we can abstract the information that has been leaked in the process of compression and still make the system operate reliably within a margin of error.

Inspired by the above points and thoughts, the following questions are posed in this thesis:

1. Can a graph-based direct cross-entropy algorithm be combined with neural networks and successfully trained to directly couple the tasks of perception and planning ?

2. If yes, how does such an algorithm perform in the case of continuous observations and states ? In particular: is such an algorithm useful for source localization problems with continuous observations and states ?

3. And lastly, does this algorithm scale to very high-dimensional discrete inputs, such as images, when we use a lossy compression scheme derived with the help of a Variational Autoencoder ?

All these questions will be answered in the sections to come using appropriate experiments.

1.7 Main contributions

The key research goal for us will be to analyze the possibility of a more intimate integration between perception and planning as previously accomplished. The approach by which integrated perception and planning is done in this work involves a graph data structure with neural net classifiers embedded in each node. The neural net accepts the input observation and generates a transition probability distribution over the next nodes. Each node on the other hand yields an action distribution. The neural network serves therefore as a coupling structure, tying observations and actions more closely together. The main idea is presented in Figure 1.7.

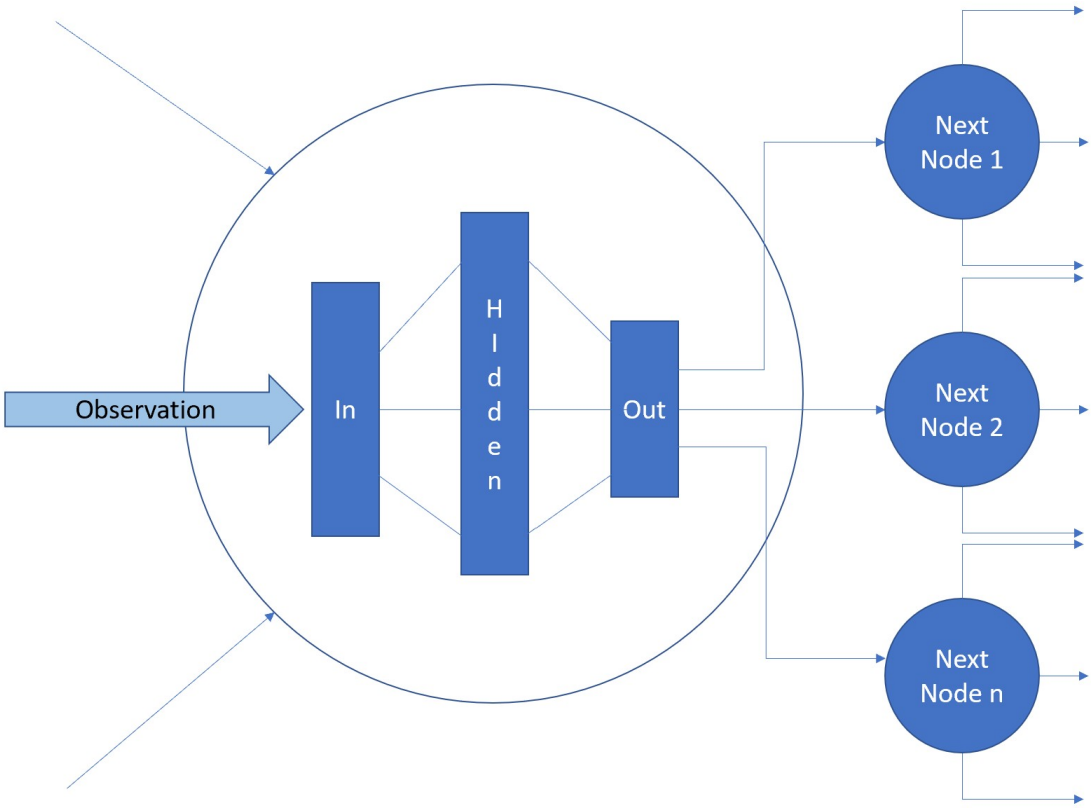


Figure 1.7: Main approach for integrating perception and planning. A neural network following the multi-layer perceptron architecture is embedded into the node of a graph. The network accepts a sample of observation from a continuous space and creates a probability distribution over next nodes. Each next node itself outputs a distribution over actions from which one concrete action will be sampled and carried out.

Such an approach can be used in problems with continuous observations, as well as in problems with very high-dimensional discrete observations. The scaling to very high-dimensional inputs is done with the help of encoders that compress the input into a lower-dimensional latent space.

A new algorithm is proposed: Continuous Observation Graph Neural Net-based Direct Cross-Entropy policy search, short **COGNet-DICE**. It is built using ideas from direct cross-entropy policy search methods combined with neural network classifiers. The algorithm is applied on a moving source localization problem with continuous observations to investigate its properties and compare it to other state of the art solutions. Subsequently it is shown how the new algorithm scales well to very high-dimensional discrete visual inputs with the help of Variational Autoencoders.

1.8 Reasons and intuition

We presented some of the ideas behind the new approach without justifying them from a deeper, more visionary perspective. Some still standing questions are: Why have such an approach to begin with ? What is the reason for having graphs ? Why embed neural nets in their nodes ? Why have action distributions also embedded in their nodes ? To this there can be multiple explanations from different points of view.

Lets begin with an intuition from a biological standpoint. We can imagine that the biological brain is a conglomerate of very dense neurons packed in a small volume we call skull. They are from birth allocated to already predefined centers of the brain. There exist multiple dedicated centers responsible for different types of skills. However, these centers still have to be tuned by training from an early age. Children try and fail at many tasks in order to fine-tune the simplest of skills, such as walking for example. When an action is successful, those neurons that were active during its execution strengthen the connections between their synapses, such that the behaviour that led to success is reinforced. As

suggested by [Lake et al., 2017] the brain has a modular structure and there exist components such as attention and perceptual blocks that serve as dispatchers, deciding what brain centers to activate and orchestrate. Once a certain center is activated, it performs the action it was fine-tuned to do, and passes the control to other centers based on the new feedback. A high-level impression about how the brain has a modular structure, with each component having submodular components within it is presented by Figure 1.8.

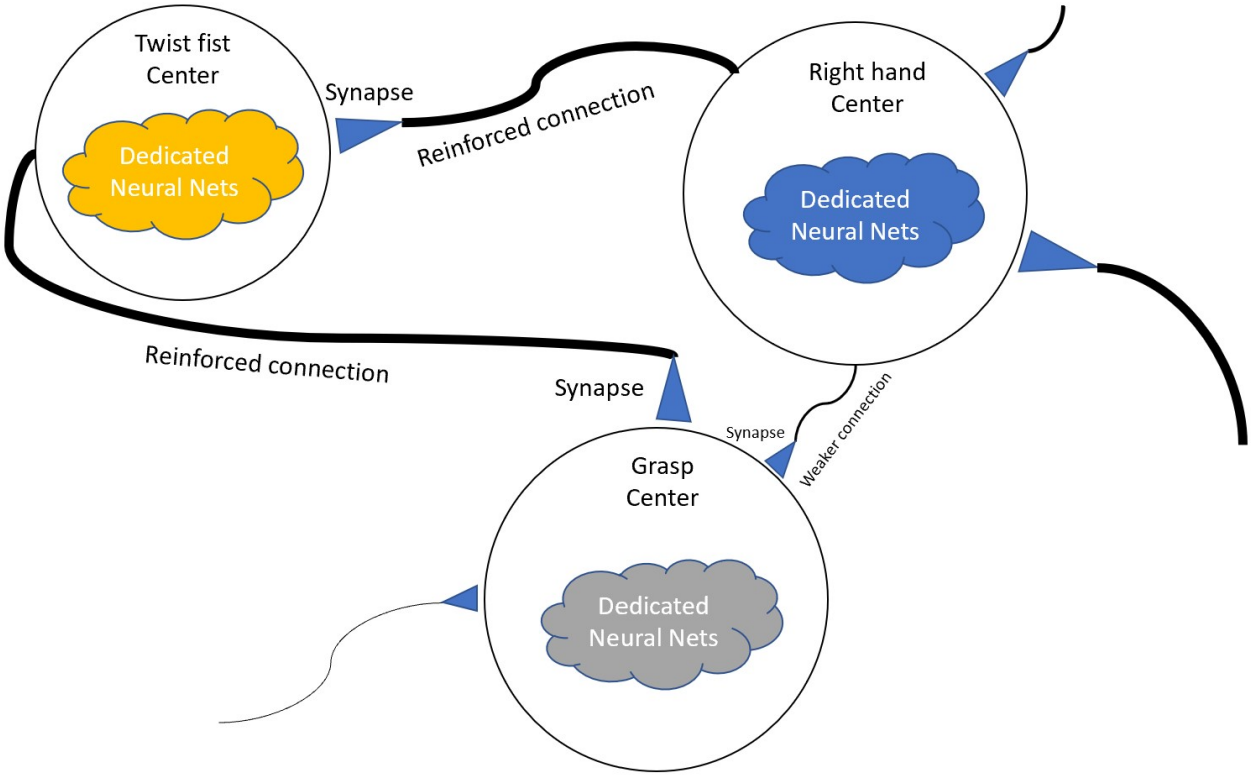


Figure 1.8: Brain centers dedicated for different tasks reinforce connections between themselves. Each component is a black box abstraction that has more lower level circuitry inside.

From a mathematical standpoint, the synapses inside brain centers are emulated by the weights of the neural nets inside the nodes; and the edges of the graph can be thought of as longer neurons that connect different brain centers together. Each observation has to be allocated a pipeline that will process it. The pipeline is the entire graph of nodes and each node is a specialized center in the pipeline. A node will take the observation and bring it to a higher-dimensional hyperplane represented by the hidden layer of the neural net inside the node. The hyperplane is more useful to classify the observation because data is more easily separable in higher dimensions (see Figure 1.9). This idea comes mainly from Support Vector Machines (SVMs) and is called the "kernel trick". Another aspect that deserves mentioning is that a hidden layer with many neurons is very powerful in the sense that each neuron acts like a basis function. It performs linear as well as nonlinear calculations on the inputs it receives as presented in Figure 1.10. Many such neurons stacked together are capable of learning very complex nonlinear models. We can either have more neurons in one hidden layer, or less neurons but on multiple layers. Multiple layers is essentially better because it means we apply the abstraction principle, in other words connecting building blocks together, where each block builds higher learning representations on top of what the other block has learned. On the other hand, one of the main theorems in artificial intelligence says that it is possible to represent any function with just one hidden layer containing an infinite number of neurons. We will pick for simplicity one hidden layer with 10 times as many neurons as the length of the input sample used. Once the observation is classified, the last output layer of the perceptron neural net serves as a dispatcher, or more technically: a multi-class classifier that decides what next node in the pipeline is best for handling the next observation further. Therefore each node acts as a specialist in the overall pipeline represented by the graph: at first the node acts on the current observation and then it decides what node is responsible next for the new

observation.

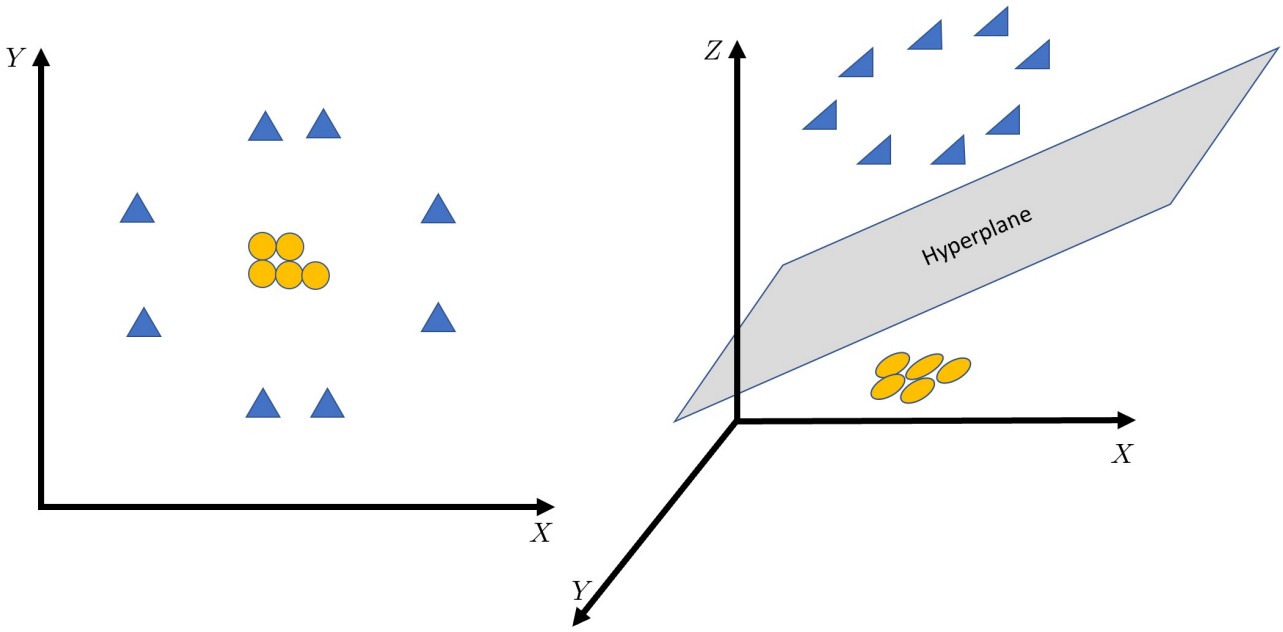


Figure 1.9: Nonlinear split in a lower-dimensional space vs linear split in higher-dimensional space. A circular model would be necessary in the 2D cartesian coordinate system, whereas in 3D a simple linear plane will suffice.

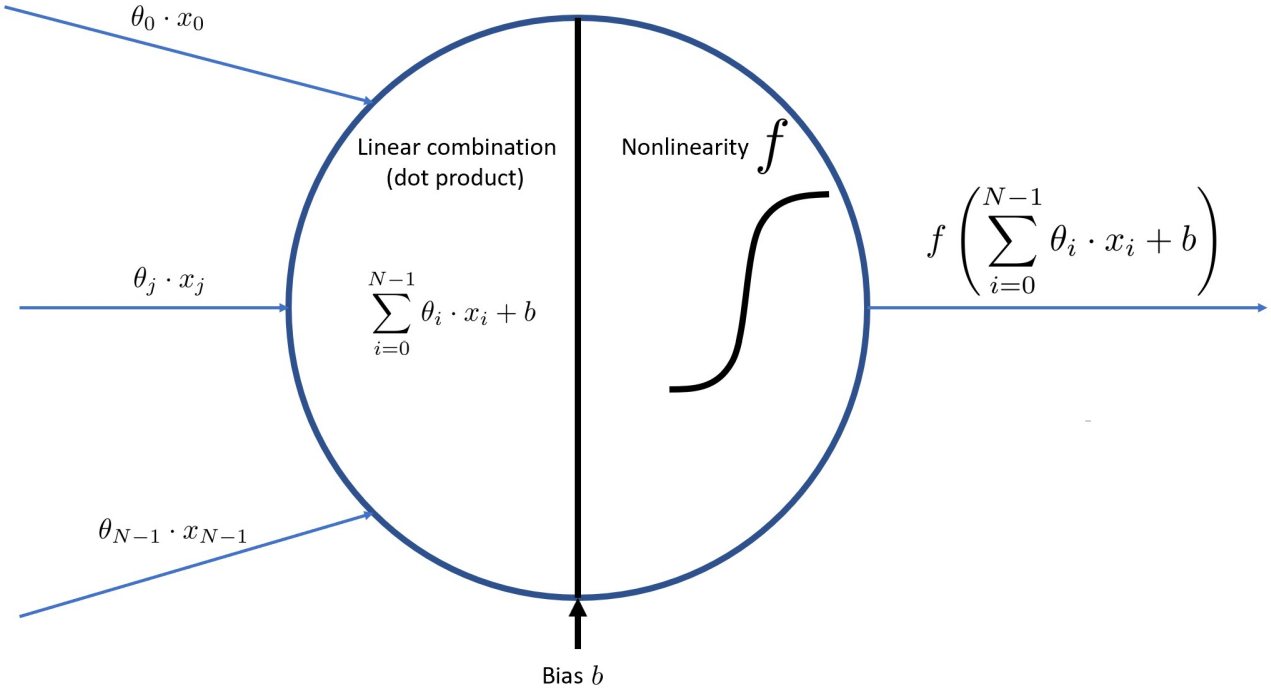


Figure 1.10: Nonlinearity applied to a linear combination of weighted inputs leads to complex and powerful function approximations. The dot product can be thought off as the simplest kernel there is. Other popular kernels from Support Vector Machines theory are: the radial basis function (RBF), polynomial kernel, etc. The nonlinearity side can be accomplished by many functions as well, such as: the sigmoid function, tanh, leaky ReLU, etc.

2 Background Theory

This chapter will first give an in-depth explanation of the important theoretical topics used in this work, namely: Decentralized Partially Observable Markov Decision Processes, Cross-Entropy Optimization and Variational Autoencoders.

An extra implementation topic will also be presented, specifically the main ideas of the Graph-based Direct Cross-Entropy algorithm for policy search, or short G-DICE. This algorithm serves as a baseline for further development and adaptation to a deep learning context.

2.1 Decentralized-POMDPs

The Decentralized Partially Observable Markov Decision Process is a principled framework for reasoning about multiple agents cooperating at a task when there is uncertainty in the state of the world they observe and act upon. For more insight into the topic of Dec-POMDPs refer to [Oliehoek et al., 2016]. The section below will however also list the most important information to have a conceptual understanding of Dec-POMDPs. The Dec-POMDP will be formally defined and then its complexity will be mathematically analyzed, followed by a discussion about different types of inter-agent communication.

More formally, a Dec-POMDP is a tuple of the form $\langle Ag, S, A, T, R, Z, \mathcal{O} \rangle$, where:

- $Ag = \{1, ..., n\}$ is the set of agents
- S is a finite set of world states
- $A = \times_i A_i$ is the set of joint actions, each agent i performing an action from set A_i
- T is the transition function defined as $T : S \times A \to P(S)$, mapping state-action pairs to probability distributions over next states
- R is the reward function defined as $R : S \times A \to \mathbb{R}$, mapping state-action pairs to a real number that quantifies their utility
- $Z = \times_i Z_i$ is the set of joint observations, each agent i receiving an observation from set Z_i
- $\mathcal{O}$ is the observation function defined as: $\mathcal{O} : S \times A \to P(Z)$, mapping state-action pairs to probabilities over joint observations
- $b \in P(S)$ is the belief over states. $b_0(s_0) \triangleq P(s_0)$ is the initial state distribution at time t=0
- h is the horizon of the problem, i.e. the number of time steps the agents will interact with the world

In a Dec-POMDP joint actions are taken by the agents $a_t = \langle a_{1,t}, \ldots, a_{n,t} \rangle$ at each time step t. The actions taken cause a state transition of the environment according to a function T. The environment then responds with joint observations $z_t = \langle z_{1,t}, \ldots, z_{n,t} \rangle$ according to some observation function $\mathcal{O}$. Each agent knows only about its own action $a_{i,t}$ it took and its own observation $z_{i,t}$ that came as a result of the state transition. However all agents receive the same reward $r_t = R(s_t, a_t)$.

There are two important concepts that accompany the Dec-POMDP definition: histories and policies.

Definition 2.1.1. A local action-observation history, or AOH for an agent i is a sequence of actions taken and observations received by that agent at all time steps $t = 0, \ldots, h$

Mathematically we will denote it as:

$$\vec{p}_{i,t} = (a_{i,0}, z_{i,1}, \ldots, a_{i,t}, z_{i,t}) \tag{2.1}$$

The AOH can also be thaught of as a concatenation/tuple of two separate histories, namely the observation history:

$$\vec{z}_{i,t} = (z_{i,1}, \ldots, z_{i,t}) \tag{2.2}$$

and the action history:

$$\vec{a}_{i,t} = (a_{i,0}, \ldots, a_{i,t}) \tag{2.3}$$

All the information an agent might need to learn is carried within an action and observation history, and each agent has to come up with its own local policy based on these.

Definition 2.1.2. A local policy π_i for an agent i is a mapping from local observation histories to actions.

$$\pi_i : \vec{Z}_i \to A_i \tag{2.4}$$

All agents act together according to their respective local policy, which leads to the emergence of an overall joint policy.

Definition 2.1.3. A joint policy π is a tuple of the form $\langle \pi_1, \ldots, \pi_n \rangle$, such that the local policy π_i of each agent i maps local observation sequences to the next local action.

$$\pi : \vec{Z} \to A \tag{2.5}$$

The overall goal is to find an optimal joint policy π^* out of all joint policies π that maximizes the expected sum of discounted rewards over time, which represents the value of the joint policy:

$$V(\pi) = \mathbb{E}\left[\sum_{t=0}^{h-1} \gamma^t R(s_t, a_t) | b_0, \pi\right] \tag{2.6}$$

The expectation is with respect to the sequences of states and executed joint actions. Since we are dealing with a partially observable system, where the state is not uniquely identifiable, we use the term of belief over many states. Hence the variable b_0 in the above formula represents the initial belief distribution from which the state s_0 is drawn.

Previous and future notation that will be used throughout this thesis is consolidated in Table 2.1

Table 2.1: Notation summary.

Notation	Definition	Meaning
$z_{i,t}$		Local observation of agent i at time t
$\vec{z}_{i,t}$	$(z_{i,1}, z_{i,2}, \ldots, z_{i,t})$	A length-t local observation sequence of agent i
z_t	$\langle z_{1,t}, z_{2,t}, \ldots, z_{n,t} \rangle$	Joint observation at time t; tuple of local observations
$\vec{z}_t$	$\langle \vec{z}_{1,t}, \vec{z}_{2,t}, \ldots, \vec{z}_{n,t} \rangle$	A length-t joint observation sequence; tuple of local sequences
$a_{i,t}$		Local action of agent i at time t
$\vec{a}_{i,t}$	$(a_{i,0}, a_{i,1}, \ldots, a_{i,t})$	A length-$(t+1)$ local action sequence of agent i
a_t	$\langle a_{1,t}, a_{2,t}, \ldots, a_{n,t} \rangle$	Joint action at time t; tuple of local actions
$\vec{a}_t$	$\langle \vec{a}_{1,t}, \vec{a}_{2,t}, \ldots, \vec{a}_{n,t} \rangle$	A length-$(t+1)$ joint action sequence; tuple of local sequences
s	$(s_0, s_1, \ldots, s_{h-1})$	Sequence of states at different time steps $t < h$

To be able to appreciate how hard the problem actually is we will discuss some complexity results related to Dec-POMDPs. In order to avoid confusions, the algorithm complexity will be denoted through big O notation, while $\mathcal{O}$ is reserved for the observation function.

A Dec-POMDP is a very flexible framework, however the flexibility comes at a price. At a specific timestep t, there are $(|A_i| \cdot |Z_i|)^t$ action-observation combinations for an agent i. This means that for an agent i, the number of sequences is:

$$\sum_{t=0}^{h-1}(|A_i| \cdot |Z_i|)^t \tag{2.7}$$

Applying the geometric progression formula where the ratio absolute value is greater than or equal to one ($|ratio| \geq 1$) we get:

$$\sum_{t=0}^{h-1}(|A_i| \cdot |Z_i|)^t = \frac{(|A_i| \cdot |Z_i|)^h - 1}{(|A_i| \cdot |Z_i|) - 1} \tag{2.8}$$

total number of action-observation histories that need to be tracked.

Raising the number of possible actions to the total number of observations an agent could potentially receive will give us the number of policies to be evaluated for one agent:

$$|A_i|^{\frac{|Z_i|^h - 1}{|Z_i| - 1}} \tag{2.9}$$

Assuming n agents that are heterogeneous, we get the number of joint policy evaluations in case of an exhaustive brute-force algorithm to be:

$$O\left(|A_*|^{\frac{n(|Z_*|^h - 1)}{|Z_*| - 1}}\right) \tag{2.10}$$

where $|A_*|$ and $|Z_*|$ stand for the largest individual action and observation sets. The cost of evaluating one joint policy is:

$$O(|S| \cdot |Z_*|^{nh}) \tag{2.11}$$

Therefore, total cost is derived by multiplying (2.10) and (2.11):

$$O\left(|A_*|^{\frac{n(|Z_*|^h - 1)}{|Z_*| - 1}} \cdot |S| \cdot |Z_*^{nh}|\right) \tag{2.12}$$

As we can see in expression (2.12), the first product term (from equation (2.10)) has an exponent which itself involves a power of h. Therefore we say that a Dec-POMDP has double exponential complexity, and that it is double exponential in the horizon h.

Regarding the communication between agents, there are multiple Dec-POMDP variations. In its most general form, we assume there is no communication between the agents. In this situation, complexity is NEXP, or double-exponential as stated previously (see also [Bernstein et al., 2002]). In case we assume complete communication between the agents, the system effectively boils down to a POMDP structure. A POMDP was shown to be PSPACE-complete (see [Papadimitriou and Tsitsiklis, 1987]). There can also be hybrid scenarios, where the agents can communicate, but only periodically, i.e. after a certain number of time steps less then the total horizon. This is a reasonable structural assumption to make in order to reduce the complexity of the problem, and it is employed by people in teams as well in the form of exchange meetings. In this case, the system is decentralized, but only for a certain period of time, after which all the individual beliefs of the agents are merged into a centralized belief.

2.2 Cross-entropy optimization

Cross-entropy optimization is a method of finding a vector argument $\vec{x}$ from a hypothesis space H that maximizes some performance metric $M : H \to \mathbb{R}$

$$\vec{x}^* = \underset{\vec{x} \in H}{\operatorname{argmax}}\, M(\vec{x}) \tag{2.13}$$

For details on cross-entropy refer to [De Boer et al., 2005]. The following section will touch however on the essentials necessary for this thesis. At first, the notion of cross-entropy will be arrived at theoretically through an example-proof in the context of how learning happens in a Frequentist approach, and afterwards it will be shown how the cross-entropy method for policy search looks like in practice and what concrete steps it entails.

There are many ways in which one can conceptualise and arrive ultimately to the concept of cross-entropy (CE). There is however one interesting proof in the form of an example that gives a big picture and ties together many core concepts used in this thesis:

- Maximum Likelihood Estimation, or MLE
- Kullback-Leibler divergence, or KLD
- Information and Entropy

Example 2.2.1. Assuming a Frequentist, Maximum Likelihood Estimation approach to learning, our task is to maximize the likelihood of seeing the training data X by changing the parameter θ:

$$\max \ \log P(X|\theta) \tag{2.14}$$

We will designate the optimal θ as $\hat{\theta}$ and express it as:

$$\begin{aligned} \hat{\theta} &= \underset{\theta}{\operatorname{argmax}} \ P(X_{1:n}|\theta) = \underset{\theta}{\operatorname{argmax}} \ \log P(X_{1:n}|\theta) \\ &= \underset{\theta}{\operatorname{argmax}} \left[\frac{1}{N} \sum_{i=1}^{N} \log P(X_i|\theta) - \frac{1}{N} \sum_{i=1}^{N} \log P(X_i|\theta_0) \right] \\ &= \underset{\theta}{\operatorname{argmax}} \ \frac{1}{N} \sum_{i=1}^{N} \log \frac{P(X_i|\theta)}{P(X_i|\theta_0)} \end{aligned} \tag{2.15}$$

where θ_0 is the parameter of the true underlying model governing the process that we are analyzing. Note that we were allowed to add a log operator before $P(X)$ in the first line of (2.15) and subtract $\frac{1}{N}\sum_{i=1}^{N} \log P(X_i|\theta_0)$ in the second line of (2.15) because argmax is an operator that remains invariant when we scale a function, as well as when we remove a bias from it.

Moving on with the proof, we will apply next the law of large numbers and get:

$$\begin{aligned} \lim_{N\to\infty} \underset{\theta}{\operatorname{argmax}} \ \frac{1}{N} \sum_{i=1}^{N} \log \frac{P(X_i|\theta)}{P(X_i|\theta_0)} &= - \underset{\theta}{\operatorname{argmax}} \ \int \log \frac{P(X|\theta_0)}{P(X|\theta)} P(X|\theta_0) dX \\ &= \underset{\theta}{\operatorname{argmin}} \ \int \log \frac{P(X|\theta_0)}{P(X|\theta)} P(X|\theta_0) dX \end{aligned} \tag{2.16}$$

where $\log \frac{P(X_i|\theta_0)}{P(X_i|\theta)}$ is what we call the Kullback-Leibler Divergenge, or KLD between the distributions in the numerator and the denominator. An insightful fact is that the problem of computing the maximum likelihood estimate is equivalent to minimizing the KLD between the true model distribution from which the input X stems ($X \sim P(X|\theta_0)$), and the learned distribution that we compute iteratively $P(X|\theta)$.

If we go one step further and apply the logarithm quotient rule we get:

$$(2.16) = \underset{\theta}{\operatorname{argmin}} \left[\int P(X|\theta_0) \log P(X|\theta_0) dX - \int P(X|\theta_0) \log P(X|\theta) dX \right] \tag{2.17}$$

where $\int P(X|\theta_0) \log P(X|\theta_0) dX$ represents the information in the world that can theoretically be learned and $\int P(X|\theta_0) \log P(X|\theta) dX$ is the information in our model. The negative of this information: $-\int P(X|\theta_0) \log P(X|\theta) dX$ is the cross-entropy term that we tried to arrive at (Incidentally, notice that the prefix "cross-" can be related to the fact that θ_0 and θ are juxtaposed inside the integral). It is in our interest to minimize the cross-entropy as much as possible to get as close as possible to the total information objectively available in the world: $\int P(X|\theta_0) \log P(X|\theta_0) dX$ $\triangle$

Cross-entropy (CE) is an important methodology of estimating the probability of rare events and optimize combinatorial problems (see [De Boer et al., 2005]). In this work we will use it as a means of finding good Dec-POMDP policies, which is in essence a combinatorial problem. The CE method for searching policies consists of 2 phases (see also Figure 2.2 for illustration):

1. Sample a random variable X from a parametric distribution f
2. Take N_b best samples and use these to update the parameters θ of function f

In order to avoid issues with local optima, a smoothing procedure may be used:

$$\theta_{t+1} = \alpha\theta_{t+1} + (1 - \alpha)\theta_t \tag{2.18}$$

where α is a specified learning rate. In our case the algorithm will run iteratively a certain predefined number of steps N_k. An alternative could be to run the algorithm until no improvements in the policies occur for a certain number of successive iterations. Since this is an anytime algorithm, it can also be stopped after a random period of time.

2.3 Variational Autoencoders

Variational Autoencoders are a class of generative models specifically designed for learning latent representations of a continuous or discrete input X. For a detailed analysis of this topic check [Doersch, 2016]. The content in this chapter however will be self-contained in explaining the most relevant aspects of Variational Autoencoders in the context of image processing and computer vision. The initial aim is to understand the placement of Variational Autoencoders in the broader class of generative models. Subsequently, the goal is to theoretically prove how learning is accomplished in a Variational Autoencoder such that the compression, data augmentation and noise robustness aspects are emphasized and understood in the context of many flying agents analyzing a 2D image scene.

Generative models are a useful emerging class of networks that can generate function distributions of the form $P(X|Y = y)$, which are called density functions. There are 2 main types of generative models: the ones that explicitly model the density function and the ones that elicit an implicit density function.

Implicit density estimation models will not be used in this work, but it is worth mentioning that they subsume the ubiquitous Generative-Adversarial Networks. Shortly explained, in this case we have a Game Theory type of setting with 2 players: a generator trying to generate images as close as possible to realistic images, and a discriminator trying to discern whether the generated images are real or fake. Therefore, an implicit density function arises in the process of the Generator network trying to deceive the Discriminator counterpart into believing that the generated data is actually taken from real life, and not artificially created.

The explicit density generative models on the other hand are more relevant to what we will use, and here we can distinguish between: tractable density calculation models and approximate density estimation models.

Some of the most popular models from the tractable density class at the moment of writing this thesis are: PixelCNN [Van den Oord et al., 2016] and PixelRNN [Oord et al., 2016]. What they do at their core is generate a probability distribution of the type:

$$P(X) = \prod_{i=1}^{n} P(X_i|X_1, ..., X_{i-1}) \tag{2.19}$$

where $P(X)$ represents the likelihood of an image X, and $P(X_i|X_1, ..., X_{i-1})$ represents the probability of the pixel value at index i to be X_i, given that the previous pixels seen had values $X_1, ..., X_{i-1}$. In other words, it is a similar problem with the one from classical information theory, when trying to predict the next bit given the previously seen bits. In this case however, our bits are actually pixels, and we try to predict the next pixel in an image given the previous ones. This type of modelling suffers from being too slow in practice, especially PixelRNN networks. Networks inspired from PixelCNN try

to mitigate this prediction speed issue, but the problem still persists, since we still have to exhaustively visit all pixel groups sequentially in order to make the next prediction.

The next class, i.e. approximate estimation models is more successful at overcoming the prediction speed issue. They can be divided into: Markov Chains (for example: Boltzmann Machines) and Variational Autoencoders. What presents special interest however for this thesis are the Variational Autoencoders (VAEs) and a simpler subtype of these called Autoencoders (AEs). We will go into more depth about VAEs, since they are more generic than AEs and will enlighten other aspects about later algorithmic implementations for this thesis.

VAEs consist of 2 conceptual components: an encoder and a decoder, as can be seen in Figure 1.5. Simply put, the encoder tries to compress an input X into a latent space L. The decoder on the other hand samples the latent function to produce an output Y, which will look as if it stems from the same distribution as X. The Autoencoder will try to make Y indistinguishable from X, while the Variational Autoencoder will be more creative and generate a Y that is not exactly X, but rather with novel features, similar in nature to what the input X could have exhibited.

More formally, Variational Autoencoders define a density function of the form:

$$P_\theta(X) = \int P_\theta(L)P_\theta(X|L)dL \tag{2.20}$$

where X is the image and L is a latent variable representing features used to generate X (ex: how much green there is in an object, what is the orientation of the object in the image etc.)

- $P_\theta(X)$ represents the data likelihood
- $P_\theta(L)$ is the set prior, which is a Gaussian many times for convenience
- $P_\theta(X|L)$ is the conditional distribution showing how the input depends on the latent variable.

Distribution $P_\theta(X|L)$ is computed with the help of a neural network, more concretely: by the decoding network in the VAE.

According to Bayes' formula, we can compute also the posterior, i.e. the encoding distribution as:

$$P_\theta(L|X) = \frac{P_\theta(X|L)P_\theta(L)}{P_\theta(X)} = \frac{P_\theta(X|L)P_\theta(L)}{\int P_\theta(X|L)P_\theta(L)dL} \tag{2.21}$$

Because of the intractable integral in the denominator, the posterior is also intractable. Hence, another neural network will be necessary to approximate it as well, and it is called the encoder network in the VAE. The encoder defines a distribution $Q_\psi(L|X)$ that approximates the true posterior distribution $P_\theta(L|X)$ from equation (2.21):

$$Q_\psi(L|X) \approx P_\theta(L|X) \tag{2.22}$$

Statement 2.3.1. We will obtain an encoding distribution $Q_\psi(L|X)$ by trying to optimize a variational lower bound (hence the title: "Variational" AEs instead of simply AEs).

The aim of the following proof of the above statement is to mathematically justify the presence of the 2 physical networks: the encoder and decoder, and emphasize the aspect of approximation that the overall VAE gives as a feature to the COGNet-DICE algorithm. Both the employment of cross-entropy optimization and variational autoencoding place COGNet-DICE in the class of approximate planning algorithms.

We start by expressing the log-likelihood of the input and applying Bayes' formula on it subsequently:

$$\log P_\theta(X_i) = \mathop{\mathbb{E}}_{L\sim Q_\psi(L|X_i)}[\log P_\theta(X_i)] = \mathop{\mathbb{E}}_{L}\left[\log \frac{P_\theta(X_i|L)P_\theta(L)}{P_\theta(L|X_i)}\right] \tag{2.23}$$

Next we apply the trick of multiplying and dividing with a term in order to inject distribution Q into the derivation process:

$$(2.23) = \mathop{\mathbb{E}}_{L}\left[\log\left(\frac{P_\theta(X_i|L)P_\theta(L)}{P_\theta(L|X_i)}\cdot\frac{Q_\psi(L|X_i)}{Q_\psi(L|X_i)}\right)\right] \tag{2.24}$$

The step above mathematically explains the presence of a lossy, approximating encoder $Q_\psi(L|X)$ and hence, it represents one of the approximation steps. Next, purposefully combining convenient pairs together will lead us to:

$$\begin{aligned}
(2.24) &= \mathop{\mathbb{E}}_{L}\left[\log\left(P_\theta(X_i|L)\cdot\frac{P_\theta(L)}{Q_\psi(L|X_i)}\cdot\frac{Q_\psi(L|X_i)}{P_\theta(L|X_i)}\right)\right] \\
&= \mathop{\mathbb{E}}_{L}\left[\log\left(P_\theta(X_i|L)\cdot\left(\frac{Q_\psi(L|X_i)}{P_\theta(L)}\right)^{-1}\cdot\frac{Q_\psi(L|X_i)}{P_\theta(L|X_i)}\right)\right] \\
&= \mathop{\mathbb{E}}_{L}\left[\log P_\theta(X_i|L) - \log\frac{Q_\psi(L|X_i)}{P_\theta(L)} + \log\frac{Q_\psi(L|X_i)}{P_\theta(L|X_i)}\right] \\
&= \mathop{\mathbb{E}}_{L}\left[\log P_\theta(X_i|L)\right] - \mathop{\mathbb{E}}_{L}\left[\log\frac{Q_\psi(L|X_i)}{P_\theta(L)}\right] + \mathop{\mathbb{E}}_{L}\left[\log\frac{Q_\psi(L|X_i)}{P_\theta(L|X_i)}\right] \\
&= \mathop{\mathbb{E}}_{L}\left[\log P_\theta(X_i|L)\right] - D_{KL}\left[Q_\psi(L|X_i)||P_\theta(L)\right] + D_{KL}\left[Q_\psi(L|X_i)||P_\theta(L|X_i)\right]
\end{aligned} \tag{2.25}$$

where D_{KL} stands for the Kullback-Leibler Divergence.

Note the following interesting facts about the result:

- The term $\mathbb{E}_L\left[\log P_\theta(X_i|L)\right]$ is computed by the decoder network.
- The term $D_{KL}\left[Q_\psi(L|X_i)||P_\theta(L)\right]$ has a closed form, since both distributions: the conditional $Q_\psi(L|X_i)$ and the prior $P_\theta(L)$, are purposefully set to be convenient functions for which we know how to derive analytical properties. Gaussian functions are typically used in practice.
- The third term $D_{KL}\left[Q_\psi(L|X_i)||P_\theta(L|X_i)\right]$ is intractable since $P_\theta(L|X_i)$ is intractable (because of the integral in equation (2.21)). However, we can take advantage of the known fact that the KL Divergence is always non-negative (i.e. ≥ 0).

Therefore, we can say that what we started with, i.e. $\log(P_\theta(X_i))$ is greater than or equal to some lower bound B:

$$B(X_i,\theta,\psi) = \mathop{\mathbb{E}}_{L}\left[\log P_\theta(X_i|L)\right] - D_{KL}\left[Q_\psi(L|X_i)||P_\theta(L)\right] \tag{2.26}$$

△

More explicitly, we have the following expression to optimize with respect to θ and ψ:

$$\log(P_\theta(X_i)) \geq \mathop{\mathbb{E}}_{L}\left[\log P_\theta(X_i|L)\right] - D_{KL}\left[Q_\psi(L|X_i)||P_\theta(L)\right] \tag{2.27}$$

This therefore represents another approximation step. If we now optimize the lower bound, we will find the optimal parameters $\hat{\theta}$ and $\hat{\psi}$. Using $\hat{\psi}$ we will be able to compute $Q_{\hat{\psi}}(L|X_i)$ as a good approximation for the intractable true posterior $P_\theta(L|X_i)$. Using $\hat{\theta}$ we will be able to find a good decoding distribution $P_{\hat{\theta}}(X_i|L)$ that reconstructs the input X.

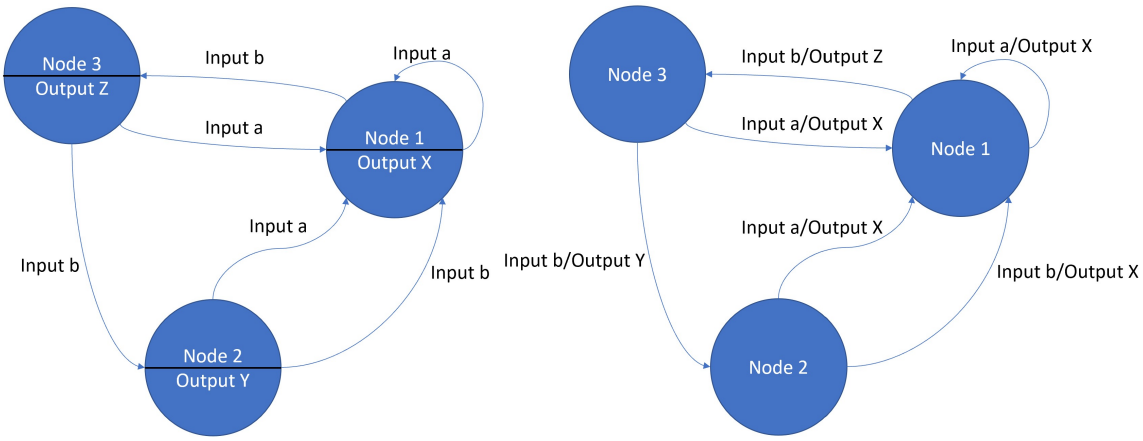
Now that we proved how VAEs work internally, one can hopefully have an intuition about why for example $Q(L|X)$ can be useful for other tasks. We can treat it as a distribution that intelligently compresses the input and be able to sample from it instead of from the high-dimensional images when it comes to planning tasks. For tasks involving teams of many agents, compression is very important to speed up the training procedure. Another aspect is that with a compressed variable L, the VAE can decode different outputs for a certain ground truth input. This is extremely useful for data augmentation purposes. We can generate a training dataset with which we can train other neural networks and make them more robust to noise. This is extremely important in transfer learning, when the agents can be trained in a simulator and be able to cope with noisy signal measurements in the real world where they are embodied as physical robots.

2.4 G-DICE

This section explains the G-DICE algorithm: "Graph-based Direct Cross-Entropy" search, which is the main methodology for searching policies in this work. A short discussion will be carried on the two important graphical controller types used in engineering and about which type requires less storage theoretically. The second part will be about how G-DICE contributes to the shaping of these controllers through learning the parameters governing node transition and action selection.

The fun fact about the "G-DICE" acronym is that it actually carries a meaning: this algorithm is essentially an evolutionary algorithm in nature, and hence: "throwing the dice" analogy. The D stands for "direct" and describes the fact that we will do a direct search on the policy space, without attempting to permanently prune it. This is note-worthy and speaks for the importance of this algorithm to handle large state spaces. Other methods for example rely on permanently pruning the space to arrive at smaller sizes they are able to handle.

Before diving into the steps of the algorithm, let us have a short introduction into the type of graphs we will make use of. For each agent in the system, we will define a finite state automaton (short FSA). The FSA will be used to learn the controller for each agent, in other words the block dictating the local policy of the agent. There are 2 main classes of FSAs, namely: the Moore machine and the Mealy machine. Any controlling task we might think of that can be modeled using a Moore machine, does have an equivalent Mealy counterpart. The difference between them is how they model outputs and node transitions. In a Mealy machine, an input causes a node transition and generates an output during the node transition itself. The Moore machine however, does not generate the output during the node transition, but rather in the node itself. Figure 2.1 emphasizes precisely this aspect. Because of this aspect, a Mealy machine does actually require less number of nodes on average, therefore less storage. However, this is an insignificant gain for the purposes of this thesis because we will consider a small number of nodes anyway and therefore a Moore machine is preferred because of its more intuitive nature. In other situations however, when policy controllers with a lot of nodes are unavoidable, we might be able to consider Mealy Machines to save space. At an implementation level, this means that, for the purposes of this thesis, the neural net classifier is an object that is assigned as an attribute to the node object of the graph. In a Mealy scenario, the classifier would be assigned as an attribute to the edge object of the node.



(a) Moore (b) Mealy

Figure 2.1: Moore vs Mealy Machines. In a Moore machine the output is placed in nodes. In a Mealy machine the output is placed on node transitions. Mealy machines have a lower space complexity on average, but this effect is negligible for our purposes.

Moving on to the G-DICE algorithm itself, we can say that its purpose is to find the policy controllers. More specifically: we set the number of nodes we want the controllers to have, and the G-DICE algorithm should build the action distributions of each node and the next node transition distributions. A controller is initialized such that there is a uniform distribution in every node for what actions to

take, and a random next node classifier represented by arbitrarily weighted edges between the nodes.

G-DICE has the following gist:

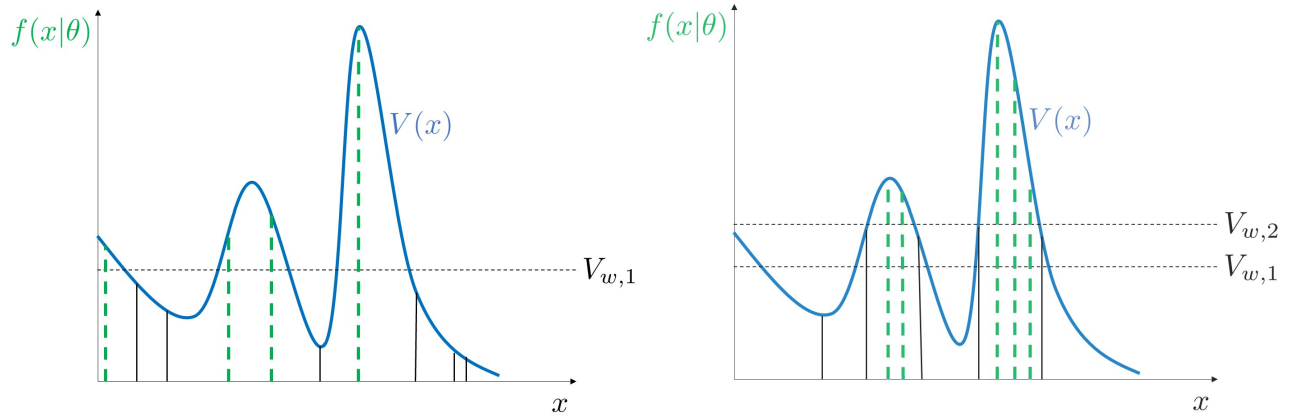
1. We create a graph for each agent containing N_n nodes.
2. Each node of the graph outputs a distribution over actions that the agent will stochastically sample and carry out.
3. Once an action is chosen and performed by each agent, a joint observation is received.
4. Based on the current node $q_{i,t}$ and observation $z_{i,t}$, each agent will transition to its corresponding sampled next node $q_{i,t+1}$.
5. Every G-DICE iteration builds many N_s policy samples. A policy sample is a deterministic controller where a deterministic action is assigned to each node and a deterministic next node is assigned for each (node, observation) pair.
6. Each controller is evaluated N_{sim} times to calculate the policy average reward, which represents the policy value.
7. According to the cross-entropy method, the update of the action distribution and next node classifier is done using only a subset N_b of all samples that meet a quality standard V_w.
8. The procedure by which action distribution update is done is MLE (maximum likelihood estimation) in the form of frequency count of the best actions that were taken in their corresponding nodes from the N_b best samples.
9. The next node distributions must also be updated by counting the discrete observations received in their associated nodes, from the best N_b samples.
10. The update is followed by a distribution smoothing procedure to avoid degeneration to local optima.

Note an important fact about what steps are impractical for robotics applications and therefore addressed by this thesis. Step 9 says it is necessary to update the next node distribution by counting observations. This implies that observations are categorized easily into simple, uniform, countable integers: an agent can observe something that could be labeled as 0, 1, 2,... etc. In everyday applications this is however rarely the case. Signals such as electromagnetic waves for example measured by analog sensors are intrinsically continuous and cannot be simply binned into predefined categories. The agent has to learn on its own how to create non-uniform binning and clustering. Hence, for our purposes, a frequency count approach is only useful in the case of actions, not in the case of observations. In other words, we can keep step 8, but not step 9 from the above recipe.

This basic routine of G-DICE will be performed many times: N_k iterations. In each iteration a number N_s of policy samples is taken. These samples are evaluated and only the first N_b best samples are retained at the end of the iteration step in order to improve the policy. The improvement results in a policy threshold value V_w that is supposed to be overcome in the next iteration steps. Each time the threshold is overcome, it gets updated to an even higher value. This monotonically increasing V_w ensures the policy improvements over time. Figure 2.2 shows this process, namely how the green samples that overcome the $V_{w,1}$ barrier are used to improve this barrier to $V_{w,2}$, such that at the next iteration, green samples will be the ones that overcome $V_{w,2}$ and so on. The black samples are the ones that will not make it in the list of best samples with N_b elements. In time, the algorithm will learn to put more emphasis on those regions that generate good green samples, and converge towards sampling from local maxima regions and hopefully around the global maximum peak. The algorithm will store inside the iteration loop the new best value V_b and new best policy such that this becomes an anytime algorithm, i.e. it will return a valid solution to the task even if it is interrupted before it is programmed to officially end.

One point to note is that there might be a time when no new sample X can overcome the threshold $V_{w,k}$, after many k iterations. This does not necessarily mean that the global maximum has been reached. The cross-entropy method is still a sub-optimal approximate algorithm. We can only hope

to find very good solutions, but there is no guarantee we will find the best solution. To increase the chances of finding good solutions, especially when no new X can surpass V_w, we have to increase the sampling variance and employ exploration techniques that might discover new interesting regions in the space of policies to sample from.



(a) One Iteration

(b) Two Iterations

Figure 2.2: G-DICE policy search. Thick, dashed, green samples are used to increase the threshold V_w for future iterations. Thin, solid, black samples are below the current V_w and therefore do not contribute to policy improvement.

To reiterate and clarify, G-DICE is detailed in Listing 1. The algorithm requires the number of controller nodes N_n to be given by the user. Typically all agents have the same N_n, but policies may exhibit varying complexity especially for heterogenous agents, and so the user can specify a different N_n for each agent. Additionally, the user has to specify how many iterations N_k will the algorithm use to update the controller graphs, how many policy samples N_s will be sampled each iteration, how many of the best samples N_b will be kept for the update. The policy update at the end of the iteration requires the specification of a learning rate α, which describes the step size in the update direction. The evaluation of a policy is done through specifying N_{sim}, i.e. the number of simulations to carry based on which to calculate the policy value. Each simulation runs a specified number of h time steps and the reward of each time step is discounted with a specified parameter γ.

The algorithm starts with the initialization of all parameters in lines 1 to 6. Line 1 initializes the best policy object as an empty object. Line 2 initializes the variable V_w, i.e worst value based on which filtering of policies will be done. Additionally, line 2 also initializes the best value V_b of the best policy. Line 3 starts the initialization process of all distributions for all agent controller graphs. Line 4 initializes the parameters for the action distributions $\theta_0^{(i)(a|q)}$ in all the nodes of the controller corresponding to agent i. Line 5 initializes the parameters for the next node distributions $\theta_0^{(i)(q'|q,z)}$ for each node in the controller corresponding to agent i.

After initialization the algorithm starts looping through a predefined number of iterations N_k as presented by line 7. Each iteration has the goal of performing a policy update at the end. In line 8 a container is defined for storing the policies that meet the hard quality requirement of surpassing V_w. Line 9 defines another container that will store the subset of best N_b policies which will be used for the policy update procedure at the end of each iteration. Every iteration will build N_s policy graph samples and evaluate them. Lines 12 and 13 build controller graphs according to custom sampling techniques and then the controllers are evaluated in line 14. The evaluation procedure will run through the predefined horizon h, i.e. total number of time steps the agents interact and will calculate the cumulative rewards using a discount factor γ according to equation (2.6). N_{sim} cumulative rewards will be calculated and used to compute one single average reward/policy score. Only if the evaluation score is greater than the threshold V_w, the data will be collected for later use as presented by lines 15 to 17. If the evaluation score is greater than all scores seen in past iterations, then this is registered as the new best value and the corresponding best policy is also updated as shown by lines 18 through

21. After the N_s sampling iterations have elapsed, the list of best policies is populated with the top-performing N_b samples as seen in line 23. New quality standard for the future iteration is set in line 25. The iteration is ended by performing the policy update through maximum likelihood estimation of the action distribution parameters in line 27 and next node distribution parameters in line 29 using the best collected samples. Lines 28 and 30 are additional smoothing steps to avoid degeneration to points of local optima. In a smoothing process the idea is to take only a small step in the direction of the updated distributions. After all iterations, the algorithm returns the best seen policy and its value, which may or may not be the optimal solution. There is no optimality guarantee, but there is a high likelihood of finding very good solutions.

Algorithm 1 G-DICE with deterministic policies

Input: Number of nodes (N_n), Number of iterations (N_k), Number of samples (N_s), Number of best samples (N_b), Number of evaluation simulations (N_{sim}), Learning rate (α), Discount factor (γ), Number of lookahead steps (h)

Output: $bestPolicy$, $bestValue$

1: $bestPolicy \leftarrow \emptyset$
2: $V_w, V_b \leftarrow -Infinity$ ▷ Worst and best values are very small initially
3: **for** $i \leftarrow 0, ..., numAgents$ **do**
4: $\quad \theta_0^{(i)(a|q)} \leftarrow UniformDistribution(), \forall q$
5: $\quad \theta_0^{(i)(q'|q,z)} \leftarrow UniformDistribution(), \forall q, z$ ▷ q' means next node, z is the observation
6: **end for**
7: **for** $k = 0, ..., N_k - 1$ **do**
8: $\quad storageList \leftarrow \emptyset$
9: $\quad bestPolicies \leftarrow \emptyset$
10: $\quad bestValues \leftarrow \emptyset$
11: $\quad$ **for** $sample = 0, ..., N_s - 1$ **do**
12: $\quad\quad deterministicAction \leftarrow Sample(actionDistribution), \forall q$
13: $\quad\quad deterministicNextNode \leftarrow Sample(nextNodeDistribution), \forall q, z$
14: $\quad\quad reward = Evaluate(h, N_{sim},\ \gamma)$
15: $\quad\quad$ **if** $reward > V_w$ **then**
16: $\quad\quad\quad storageList \leftarrow tuple(deterministicAction, reward, deterministicNextNode)$
17: $\quad\quad$ **end if**
18: $\quad\quad$ **if** $reward > V_b$ **then**
19: $\quad\quad\quad bestPolicy \leftarrow tuple(deterministicAction, deterministicNextNode)$
20: $\quad\quad\quad V_b \leftarrow reward$
21: $\quad\quad$ **end if**
22: $\quad$ **end for**
23: $\quad bestPolicies = Sort(storageList)[0 : N_b]$ ▷ Sort and take best N_b elements
24: $\quad bestValues \leftarrow getRewards(bestPolicies)$
25: $\quad V_w \leftarrow \min(bestValues)$
26: $\quad$ **for** $i \leftarrow 0, ..., numAgents$ **do**
27: $\quad\quad \theta_{k+1}^{(i)(a|q)} \leftarrow MLE(bestPolicies)\ \forall q$
28: $\quad\quad \theta_{k+1}^{(i)(a|q)} \leftarrow \alpha * \theta_{k+1}^{(i)(a|q)} + (1-\alpha) * \theta_k^{(i)(a|q)}\ \forall q$ ▷ Smoothing
29: $\quad\quad \theta_{k+1}^{(i)(q'|q,z)} \leftarrow MLE(bestPolicies)\ \forall q', z$
30: $\quad\quad \theta_{k+1}^{(i)(q'|q,z)} \leftarrow \alpha * \theta_{k+1}^{(i)(q'|q,z)} + (1-\alpha) * \theta_k^{(i)(q'|q,z)}\ \forall q', z$
31: $\quad$ **end for**
32: **end for**
33: **return** $bestPolicy$, V_b

3 Related Work

This chapter will present the state of the art in the field of decentralized systems with cooperating agents. One aspect discussed will be how the framework of reasoning about these systems, namely Dec-POMDPs are solved, and what methods are used to find optimal, as well as sub-optimal solutions to the policy search problem. Another discussion point will treat the observation side of the problem and how to scale these methods to higher-dimensional inputs.

3.1 Solving Dec-POMDPs

There are two ways in which decentralized policies can be found: either using exact algorithms or approximate algorithms. Exact algorithms have the advantage that they guarantee to find an optimal policy. One example would be a plain brute force, exhaustive search of any possible combination of random variables in the space of random variables considered for the search. This would guarantee finding the optimal solution, but for certain problems could take more time than the current age of the Universe. To have a better intuition about the magnitude: some problems have a search space bigger than the total number of atoms in the Universe. Therefore, exact methods can be very costly, precisely because of the fact that Dec-POMDPs are NEXP-complete. Nevertheless, given certain prerequisites, it is still possible to successfully apply them on certain problems and in special situations.

Approximate algorithms are on the other hand more practical. They aim to find sub-optimal, but empirically good solutions to problems for which they are specifically tailored to address and efficiently approximate. The advantage is speed of computation at the expense of missing the best solution to a problem.

3.1.1 Exact solutions

One of the important exact methods is Dynamic Programming. It is an approach that at its core employs the idea of divide and conquer. The procedure is recursive and consists in going backwards in time through a tree data structure and calculate the value of sub-trees, such that at each new iteration, one is able to reuse what has already been calculated in the previous time step. By "value" we mean the solution to the Bellman Equation:

$$V^*(s,a) = \mathop{\mathbb{E}}_{s' \sim b}[R + \gamma \max_{a'} V^*(s',a')|s,a] \tag{3.1}$$

which states the Bellman optimality principle that: the optimal strategy is to select that action which maximizes the expected future value $R + \gamma V^*(s',a')$, whatever the initial state and decision are. Dynamic Programming is a very popular and useful method for solving MDPs and POMDPs. In the Dec-POMDP case however, this approach has limited use and can be applied only when certain structural assumptions are made that reduce the Dec-POMDP to either a POMDP, or even MDP. Such practices have been demonstrated in works such as [Hansen et al., 2004] and [Bertsekas, 2019].

A second exact method is called "heuristic search". In the context of Dec-POMDPs it is more efficient than dynamic programming provided that we have good enough heuristics to guide the search. The most popular algorithm for a single agent case is A*. The heuristic in this case is such a function that consistently underestimates the total cost of taking an action. For the decentralized multi-agent case, one of the most popular algorithms is an extension of the A* algorithm called Multiagent A* [Szer et al., 2012].

3.1.2 Approximate solutions

More promising however for practical applications are methods that approximately solve Decentralized POMDPs. One such way is called: Bounded Dynamic Programming (or BDP) [Amato et al., 2007]. The idea here is to prune policies so that memory space is saved. Such an approach mitigates the main issue of dynamic programming, which suffers from too much memory requirements. Approximation comes from the fact that when we prune certain policies, we remove also optimal solutions to the problem.

Another popular generic principle stems from game theory and consists in finding a local optimal best response. The algorithm JESP: Joint Equilibrium-based search [Nair et al., 2003] implements this principle of finding a best response policy for each agent by fixing the policies of some agents and improving the policies of the remaining agents. There is also an exact version of this approach with much higher computational effort.

Heuristic methods presented earlier can also have approximations. One example of approximate heuristic search is the approximate version of Generalized Multiagent A* or GMAA* [Oliehoek et al., 2008c]. Just like JESP, it also has an exact version. This approach in particular is a hybrid method that combines ideas from game theory of solving collaborative Bayesian games (CBG) and heuristic search like in Multiagent A*. In fact Multiagent A* is a special case of GMAA*.

A graphical approach based on trees, and not closed-loop graphs used in this thesis, is proposed by [Lauri et al., 2019]. The method is based on caching policy values in the nodes of trees, so that new policies are evaluated more quickly reusing previously computed results. To calculate the belief over states, a Bayes filter is applied at each time step. Related to this approach, it can be noted that a closed loop graph can be run for infinite horizon problems, whereas a tree data structure will be used only for finite horizon problems. However, since most problems are finite horizon, or infinite horizon with a discount factor that force policy values to converge, a tree structure applies very often.

One interesting approach that makes use of a new data structure is presented by [Pajarinen and Peltonen, 2011]. The authors propose a more general design of a finite state controller (FSC), namely periodic finite state controllers. A periodic FSC is composed of M layers and the constraint is that nodes in one layer are connected only with nodes from the next layer, so no bypassing of layers is allowed. Therefore, an ordinary FSC as we know it is a special case of a periodic FSC with just one layer. The algorithm is able to handle infinite horizon policies by finding deterministic finite horizon policies and then transforming them to infinite horizon through connecting the last layer of the FSC with the first layer in a closed loop.

Another important approach that has been acclaimed for not pruning the search space is the Cross-Entropy method. Cross-Entropy optimization over the space of policies has been used in works such as [Oliehoek et al., 2008a] and [Oliehoek et al., 2008b]. In some experiments from [Oliehoek et al., 2008a] the policies are not required to surpass a certain quality threshold value V_w, whereas in [Oliehoek et al., 2008b] the filtered policies are forced to have better values each new iteration in order to ensure policy improvements. Constantly increasing the threshold of policy values will result in higher quality results and better convergence time, however may run into the risk of finding very few policies based on which to do the improvement.

A graph-based implementation of the cross-entropy method is presented by [Omidshafiei et al., 2016]. The method also makes use of the idea of action aggregation, i.e. combining many low-level/primitive actions into macro-actions. The framework is more general than a Dec-POMDP because it uses automatically generated macro-actions, and is therefore called Dec-POSMDP: Decentralized Partially Observable Semi-Markov Decision Processes. A natural continuation of this work to handle continuous observations resulted in [Omidshafiei et al., 2017]. The authors use radial basis functions to cluster the input into regions. Observations that belong to the same region will trigger similar behaviour of the agent.

Another work that considers continuous observations and a graph implementation of cross-entropy policy search is presented by [Clark-Turner and Amato, 2017]. Unlike [Omidshafiei et al., 2017], it discusses the use of a beta distribution for dividing the input space into regions, and not radial basis functions. It also does not explicitly consider macro-actions, and investigates only aggregation in the observation space, not the action space. This work shows very good results when applied to a

one-dimensional source localization problem, but is not easily scalable to multi-dimensional inputs.

The cross-entropy graph-based approaches above use Moore controllers to represent policies. The work of [Amato et al., 2010] on the other hand investigates Mealy controllers in centralized, as well as decentralized POMDPs. The authors reach the conclusion that Mealy-based approaches are more powerful than Moore counterparts based on the experimental results where Mealy controllers either achieved equal solutions or outperformed Moore controllers of the same size. They claim it is relatively easy to adapt existing graph-based solutions to use Mealy controllers. In addition, they confirm the lower spatial complexity we also touched upon in the background theory chapter.

3.2 High-dimensional observations in decision-making under uncertainty

One of the most ground-breaking work in the recent history of artificial intelligence (AI) comes from the field of reinforcement learning (RL). With access to more computation power it became feasible to adapt RL algorithms to higher-dimensional inputs. For example [Mnih et al., 2013] present a new way of using reinforcement learning with only raw pixel inputs, coming from an Atari 2600 games emulator. The inputs were not compressed with the help of encoders, however they were down-sampled to a size of $84 \times 84 \times 4$. The methodology relies on a so called Deep Q-Network (DQN) which accepts states as input and outputs the action to take. This approach was only applied on games with a single agent.

A continuation of the work on DQN was done in [Mnih et al., 2015]. It was applied to many more games (49 Atari 2600 games) and shown that it can outperform human expertise on the majority of them. The experiments also revealed some of the shortcomings of the approach, namely: because the frame in a game is high-dimensional and only a limited number of most recent frames is used to create one input (m=4 frames), some games that have very sparse rewards still remain a challenge. Such an example is "Montezuma's Revenge", a game that requires very long sequences of actions and therefore many frames stacked together to receive a single reward. The problem was not overcome even with the important trick of experience replay, i.e. storing the successful episodes which bring the rare occasional rewards and replaying them many times to the deep neural network.

Departing from the games environment into a simulation environment such that more sophisticated actions from the continuous space are considered, accompanied by raw pixel input, [Lillicrap et al., 2015] have proposed the DDPG: deep deterministic policy gradient algorithm. It uses the structure of an actor-critic system, i.e. an actor component that outputs an action to take and a critic component that evaluates the performance of the actor judging by the action it proposed and its result. It was shown that on some tasks, having raw pixel input is better than processed lower-dimensional features.

Another work which also considers only a single agent and visual inputs is [Lange and Riedmiller, 2010]. It was inspired from the neural fitted Q-learning, or NFQ model by [Riedmiller, 2005]. Unlike the previous work on Atari, this method does not accept raw observations, but rather compresses them using a deep Autoencoder network. The authors also evaluate the generated features through 4 different criteria and provide valuable insight into the potential of Autoencoders for the reinforcement learning applications. One aspect is that the state of the system is reliably captured by the feature vectors. The second aspect is that noisy images taken at the same position of the agent have almost the same exact latent representation, which allows the agent to reliably filter noise. Third is that close by high-dimensional raw inputs have correspondingly close to one another feature vectors. The last aspect is topology resemblance, i.e. the relative relation of system states is at least partially preserved in the feature space as well.

The task of navigation is very relevant in all sorts of higher-level applications such as finding a signal emitting source, self-driving cars, autonomous patrolling and it was investigated on 3D environments by [Hussein et al., 2018]. The resulting method is labeled as DAI: Deep Active Imitation. DAI also considers raw pixels like DQN, but unlike DQN the agent has a dynamic 3D viewpoint instead of a static one like in the Atari games. This work is also different in the sense that the networks employed are trained on demonstrations of correct navigation behaviour and additionally the system uses active learning, i.e. asks an expert in situations of high uncertainty to generalize to unseen situations. A lesson that might be drawn therefore is that instead of reinforcement learning with high-dimensional

observations and many trial and error examples, it may be more efficient to use high(-er)-dimensional observations but only showing the "correct" examples that the agent can imitate and fine-tune by posing questions to an expert.

Application of RL on real robots is more of a challenge than in a simulation environment and work in this direction was performed by [Levine et al., 2016]. This work presents a policy gradient method with raw RGB pixel input captured by a camera attached to a robot. Unlike the previous DQN and DAI approaches, this method was tested on real world environments and tasks such as screwing the cap of a bottle, stacking lego blocks, and others. This work is interesting not only because of application to real robots, but also because it investigates the joint training of the perception and control blocks as well as separate training of these components. In the end, the separate training achieves poor results, whereas the combined training gives much better visuomotor policies in shorter computation time.

One of the exciting directions of artificial intelligence is using lessons learned from simulation and self-play and transfer them with fine-tuning and additional training to real agents. Recent works such as [Muratore et al., 2019] argue that even a direct transfer is possible through randomizing simulator parameters, without any additional training on the robot. The work of [Akkaya et al., 2019] for example demonstrates 100 % transfer learning on an unprecedentedly complex task that uses image data. Multiple RGB camera inputs were used for vision pose estimation of a Rubik's cube and the policies were learned entirely in the virtual environment. Other works such as [Levine et al., 2015] have demonstrated transfer learning using high-dimensional representations with the necessity to at least carry a minimal or modest fine-tuning on the real robot. The work on manipulation of a Rubik's cube however goes to the extreme and shows that, provided appropriate simulation environments, and more importantly: simulation techniques, the learning algorithms can generalize to handle real world unseen and quite significant perturbations without any additional real world training.

All in all, the approaches that consider high-dimensional 2D and 3D observations are applied on single agent scenarios, which creates a void to investigate for the multi-agent case. At the same time there needs to be a way to ensure a two phase training process: a first phase of simulation in a purely virtual environment and then a second phase of fine-tuning on a real robot. It is therefore very important to make the simulation geared to handle real world dynamics and noise. This however does not mean that the exact physical properties of the real world task have to be captured in the simulator itself. More important and practical is the simulation technique, e.g. injection of noise, model parameter randomization, etc.

4 Methods

This chapter will go into the details of the newly proposed algorithm entitled: COGNet-DICE, which is a development of the discrete version G-DICE for the case of continuous observations. The chapter describes how deep neural networks are used for partitioning the observation space into regions and at the same time learning how to associate these regions to next nodes in the graph. It also describes how the action distributions are learned for each node. Subsequently, the algorithm will be adapted for computer vision applications with high-dimensional observations. The issue of transfer learning will also be discussed and the algorithm will be adapted to handle noise such that it can be readily available for applications on real robots.

4.1 The COGNet-DICE planning algorithm

COGNet-DICE: Continuous Observation Graph Neural Net-based Direct Cross-Entropy policy search is an algorithm that leverages deep learning for joint perception and planning. The "Net" term stands for the fact that a neural network is built into each node of a graph data structure. The neural network provides a more intimate connection between the process of observation and planning, as it accepts a certain observation as input and directly outputs the next node within the graph to transition to. At the same time each node transition will be associated with an action being carried out. Hence the intricate connection between observing and acting. In its plain version COGNet-DICE is meant to deal with 1D continuous signals. With the addition of specialized compression neural nets, it can handle multi-dimensional signals as well.

4.1.1 Policy representation

The setup for this problem is as follows: There are n agents indexed by $i \in [1 \dots n]$, each one following a policy π_i represented by a finite state controller, short FSC. Each FSC has its own number of nodes N_n. Every node outputs a distribution over actions $a_{i,t} \sim f_{\theta_i}(a_{i,t} \mid q_{i,t})$. All agents sample an action at each time step t from these distributions. This results in a joint action $a_t = \langle a_{1,t}, a_{2,t}, \dots, a_{n,t} \rangle$. After the actions are taken, the agents receive a reward $R(s_t, a_t)$. At the next time step, the agents receive a joint observation $z_{t+1} = \langle z_{1,t+1}, z_{2,t+1}, \dots, z_{n,t+1} \rangle$. Based on the current node $q_{i,t}$ and the observation received, the controller transitions to the next node $q_{i,t+1}$ according to the distribution $g_{\phi_i}(q_{i,t+1} \mid z_{i,t+1}, q_{i,t})$. Note that this node transition must not be confused with the state transition. The state transition follows its own distribution $P(s_{t+1} \mid s_t, a_t)$ and there may be many policy graph transitions encoding one single state.

A local policy in the graph framework can be thought of as a tuple of the form

$$\pi_i = \langle Q_i, \theta_i, \phi_i \rangle \tag{4.1}$$

where Q_i is a set of nodes, θ_i are the parameters of an action distribution function f and ϕ_i are the parameters for a node transition distribution g. A joint policy is a combination of these tuples for all agents.

Further explanations will be consistent with the notation from Table 2.1 and with the additional notation from Table 4.1

Table 4.1: Notation extension

$q_{i,t}$		Local node of agent i at time t
$\vec{q}_{i,t}$	$(q_{i,0}, q_{i,1}, \ldots, q_{i,t})$	A length-$(t+1)$ local node sequence of agent i
q_t	$\langle q_{1,t}, q_{2,t}, \ldots, q_{n,t} \rangle$	Joint nodes at time t; tuple of local nodes
$\vec{q}_t$	$\langle \vec{q}_{1,t}, \vec{q}_{2,t}, \ldots, \vec{q}_{n,t} \rangle$	A length-$(t+1)$ joint node sequence; tuple of local sequences

4.1.2 Learning

The overall goal of the algorithm is to learn θ_i that parametrizes the action distribution and ϕ_i that parametrizes the next node distribution. Learning requires ground truth data to be acquired via a simulator of the problem. Assuming we've built a simulator for the concrete task that we want to solve, the next step is to run it for many simulations called "rollouts" in order to accumulate ground truth training data as described by Listing 2.

Algorithm 2 Rollout using a stochastic joint policy

Input: FSC π_i for each agent $i = 1 \ldots, n$, initial belief state b_0, horizon h
Output: Sequences of states, FSC nodes, observations, actions and rewards

procedure ROLLOUT(π, b_0, h)
  $s_0 \sim b_0$ ▷ Sample initial state
  $q_0 = \langle q_{1,0}, \ldots, q_{n,0} \rangle = \langle 1, \ldots, 1 \rangle$ ▷ Set initial nodes by convention to first node
  $\vec{s}_0 = (s_0)$; $\vec{q}_0 = (q_0)$; $\vec{z}_0 = ()$; $\vec{a}_{-1} = ()$; $\vec{r}_{-1} = ()$ ▷ Initialize sequences
  for $t = 0, 1, \ldots, h-1$ **do**
    $a_t = \langle a_{1,t}, \ldots, a_{n,t} \rangle \sim \langle f_{\theta_1}(a_{1,t} \mid q_{1,t}), \ldots, f_{\theta_n}(a_{n,t} \mid q_{n,t}) \rangle$ ▷ Sample next actions
    $r_t \leftarrow R(s_t, a_t)$ ▷ Get reward
    $s_{t+1} \sim P(s_{t+1} \mid s_t, a_t)$ ▷ Sample next state
    $z_{t+1} = \langle z_{1,t+1}, \ldots, z_{n,t+1} \rangle \sim P(z_{t+1} \mid s_{t+1}, a_t)$ ▷ Sample next observation
    $q_{t+1} = \langle q_{1,t+1}, \ldots, q_{n,t+1} \rangle \sim \langle g_{\phi_1}(q_{1,t+1} \mid z_{1,t+1}, q_{1,t}), \ldots, g_{\phi_n}(q_{n,t+1} \mid z_{n,t+1}, q_{n,t}) \rangle$
    $\vec{s}_{t+1} = (\vec{s}_t, s_{t+1})$; $\vec{q}_{t+1} = (\vec{q}_t, q_{t+1})$; $\vec{z}_{t+1} = (\vec{z}_t, z_{t+1})$; $\vec{a}_t = (\vec{a}_{t-1}, a_t)$; $\vec{r}_t = (\vec{r}_{t-1}, r_t)$
  end for
  return $\vec{s}_h$, $\vec{q}_h$, $\vec{z}_h$, $\vec{a}_{h-1}$, $\vec{r}_{h-1}$
end procedure

Rollout evaluation One rollout will result in the following sequence of rewards:

$$\vec{r}_{h-1} = (r_0, r_1, \ldots, r_{h-1})$$

All rewards are summed up using a discount factor γ meant to put more emphasis on early rewards:

$$v(\vec{r}_{h-1}) = \sum_{t=0}^{h-1} \gamma^t r_t. \tag{4.2}$$

After many such rollouts are executed, the value of the joint policy is estimated as the sample average of all the cumulative rewards:

$$\hat{V}(\pi) = \frac{1}{k} \sum_{j=1}^{k} v(\vec{r}_{h-1}^{(j)}). \tag{4.3}$$

To be statistically significant, at least 30 rollouts have to be carried out and then take the average. In case less simulations need to be carried out to save computation time, Bessel's correction is used to more accurately calculate the average rewards, which essentially means we compute the average by normalizing with $\frac{1}{k-1}$ instead of $\frac{1}{k}$ in equation (4.3).

Action distribution To gather training data, only a subset of filtered rollouts are considered. The number of times node $q_i \in Q_i$ is visited is:

$$n(q_i) = \sum_{j=1}^{\ell} \sum_{t=0}^{h-1} \delta(q_{i,t}^{(j)}, q_i), \tag{4.4}$$

where δ is the Kronecker delta function[1]. The number of times local action a_i is chosen in node q_i is:

$$n(a_i, q_i) = \sum_{j=1}^{\ell} \sum_{t=0}^{h-1} \delta(q_{i,t}^{(j)}, q_i)\delta(a_{i,t}^{(j)}, a_i). \tag{4.5}$$

The maximum likelihood estimate, or MLE is calculated by counting the number of times action a_i was chosen in node q_i and dividing by the total number of times the node q_i was visited:

$$\hat{\theta}_{i,q_i}(a_i) = \frac{n(a_i, q_i)}{n(q_i)}. \tag{4.6}$$

Subsequently, smoothing is done with a learning rate of α:

$$\theta_i^{\text{new}} = \alpha\hat{\theta}_i + (1 - \alpha)\theta_i, \tag{4.7}$$

Node transition distribution For the filtered jth rollout, the sequence of local observations is $\vec{z}_{i,h}^{(j)} = (z_{i,1}^{(j)}, \ldots, z_{i,h}^{(j)})$ and the sequence of nodes is $\vec{q}_{i,h-1}^{(j)} = (q_{i,0}^{(j)}, q_{i,1}^{(j)}, \ldots, q_{i,h-1}^{(j)})$. Algorithm 3 is applied on these sequences of local observations and nodes to extract a set $D^{(j)}$ of training data. The entire training dataset for node q_i is obtained by combining the datasets from each filtered rollout:

$$\mathcal{D}(q_i) = \bigcup_{j=1}^{\ell} D^{(j)}. \tag{4.8}$$

Algorithm 3 Gathering training data for the node transition function of node q_i

Input: Sequence of local observations and nodes, desired node q_i
Output: Sequences of states, FSC nodes, observations, actions and rewards

```
procedure ExtractTrainingData(z⃗_{i,h}, q⃗_{i,h}, q_i)
    D ← ∅                                          ▷ Initialize training dataset
    for t = 0, 1, ..., h − 1 do
        if q_{i,t} = q_i then
            D ← D ∪ (z_{i,t+1}, q_{i,t+1})         ▷ Add next observation and node pair.
        end if
    end for
    return D
end procedure
```

During the training process, a categorical cross-entropy loss is used, where the input to the neural network embedded in node $q_{i,t}$ is the observation $z_{i,t+1}$ and the target is a one-hot vector with the attribute corresponding to the next node $q_{i,t+1}$ set to one.

[1] $\delta(x, y) = \begin{cases} 1 & \text{if } x = y, \\ 0 & \text{otherwise.} \end{cases}$

Thresholding To be able to ensure policy improvements over time there needs to be a way to select good incremental policies based on which to do the policy upgrade. The algorithm stores in a list U all policies that surpass a certain threshold V_w:

$$U = U \cup X,\ \forall X \text{ such that } V(X) > V_w$$

, where $V(X)$ stands for the value of policy sample X. At the beginning of an iteration cycle, U is an empty set: $U = \emptyset$. At the end of the iteration, only a small number N_b of best samples will be selected as a subset of U to update the policy: $B \subset U$, $\pi = u(B)$, where B is the set of best N_b samples from U and $u(B)$ is the function that carries the update based on maximum likelihood estimation using the best policy samples.

One important point is how exactly to fix and evaluate new policy sample candidates for improvement. The solution chosen stems from calculus and is explained with analogy to calculation of partial derivatives and gradients. The analogy with a gradient is made because it points into the direction af maximum growth of a function, which is also what we strive for in finding an optimal policy π^*. When we intend for example to calculate the gradient of a multivariable function $f(X, Y, Z)$, i.e. $\vec{\nabla} f = \langle \frac{\partial f}{\partial X}, \frac{\partial f}{\partial Y}, \frac{\partial f}{\partial Z} \rangle$, for each partial derivative we consider the change in only one of the variables. For instance, when computing $\frac{\partial f}{\partial X}$, the other Y and Z arguments are kept constant.

The same principle is applied also in this algorithm: when certain samples X are selected as potentially useful for a policy update, a clone of the current joint policy object is created, where the actions are deterministically set in the corresponding nodes in which they were taken in the rollout. The idea is therefore to see how does the joint policy behave if only a certain action in a certain node is changed, and all other parameters of the joint policy are kept as in the previous iteration. If the cloned joint policy has a value surpassing V_w, then the sample X is stored in the set U.

The choice to apply this algorithmic design decision is based not only on the analogy with derivatives and gradients in calculus, but also from a philosophical and biological point of view. Given that G-DICE is an evolutionary algorithm where only the fittest policies survive, the decision to apply this trick is also based on evidence from evolution: nature makes many small mutations of single genes and not huge leaps of faith with sudden multiple gene mutations. When too many genes are mutated, there is a greater risk of degenerate offspring. It is safer and more optimal to make small steps in the direction of gradually better generations.

Entropy Injection One of the edge cases that need to be handled is when the algorithm converges very slowly to a fixed joint policy value. This might mean that it heads to a local optimum value or it already got stuck in it without observing other alternative better policy it could climb to. In such cases we need to somehow "shake the policy" so that new policy space regions are explored as a result. Another way is to actually restart the algorithm from the very beginning, but this is more overhead since some of the good learned representations will have to be relearned, which takes a very long time depending on the task. Shaking the policy however is a better option and mathematically it means injecting noise into the node distributions. This however does not have sufficient theoretical soundness, since it can lead to erasing some important agent behaviours that required many iterations to learn. More justified is the approach from [Omidshafiei et al., 2017] of performing maximum entropy injection: no random noise is used, but rather a distribution that has maximum entropy from the classes of distributions used when the algorithm first got initialized.

The equation looks as follows:

$$\theta \leftarrow (1 - \alpha_{EI})[\alpha\theta_{k+1} + (1 - \alpha)\theta_k] + \alpha_{EI}\theta_{ME} \tag{4.9}$$

where $\alpha\theta_{k+1} + (1 - \alpha)\theta_k$ is the already described maximum likelihood estimation and smoothing steps and α_{EI} is the entropy injection rate typically from 1 to 3 %. This means that from time to time, the algorithm will not only exploit a current trend of joint policy samples that lead to only small value improvements, but it will also try to explore other potentially better regions to sample from. The maximum entropy distribution θ_{ME} in our case is a uniform distribution, and hence very convenient to work with.

The algorithm is programmed to initiate the entropy injection procedure when the standard deviation of joint policy values in the past k iterations was below 0.5:

$$\sigma([V_{i-k}(X), \ldots, V_i(X)]) < 0.5 \tag{4.10}$$

where V_i denotes the value at iteration i of the policy X. A good number for k in practice is 4.

Additionally, to speed up the process of finding better policies, entropy injection also occurs when the best value V_b has not been improved for more than b past iterations. Related to this, a good side-effect of entropy injection is that we can also increase the overall learning rate α to speed up the learning process and not worry much about degenerate convergence issues. That's because, even if a degenerate state is reached, the algorithm will start injecting entropy and will continue to do so until the large-α step in the wrong direction that led to degeneracy is undone by α_{EI}.

4.2 Multi-dimensional COGNet-DICE with image data

For a multi-dimensional computer vision case, the observation Z is a 2D image, i.e. $Z \in \mathbb{R}^{n \times m}$. Therefore what needs to change is the architecture of the neural network embedded in each node, such that it is able to accept a batch of $n \times m$ images.

Additionally, we would like to have the ability to feed to the neural nets in each node more training data than what the simulator can provide. This is not mandatory, but desirable, so that agents can become more adapted to noise they will surely encounter in real world applications. For this reason we will make use of a global Variational Autoencoder that will accept the observation from the simulator, compress it and decode many times into several variations of the original input. The overall setup is presented in Figure 4.1.

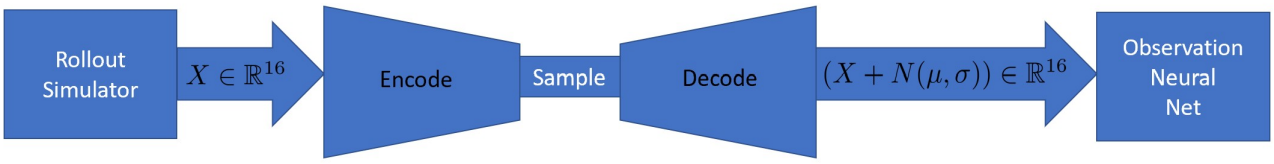


Figure 4.1: Using a global VAE for augmenting the observation dataset of each node's observation neural net. Such an approach creates noise robustness for a multiagent system. For this thesis the VAE accepts 4 by 4 inputs with a flattened size of 16. In practice the input layer can be adapted for any n by m input.

4.3 The Encoded COGNet-DICE planning algorithm

The aim of this methodology is to train the perception block on compressed inputs, so that we reduce the time and space complexity by a significant factor. This approach involves incorporating the same Variational Autoencoder used for data augmentation and noise robustness, however this time using it only partially, i.e. to encode the observation into a smaller compressed version. The decoder component is therefore omitted. When the simulator outputs the observation for an agent, it first goes through the encoder part of the VAE which compresses it to a latent vector L consisting of just one real number. As such, we can feed this number to the exact same architecture of baseline COGNet-DICE without changing it in any way. The setup is shown graphically by Figure 4.2. The vector L can also have other shapes, in which case the input layer of the observation neural nets has to be correspondingly adapted.

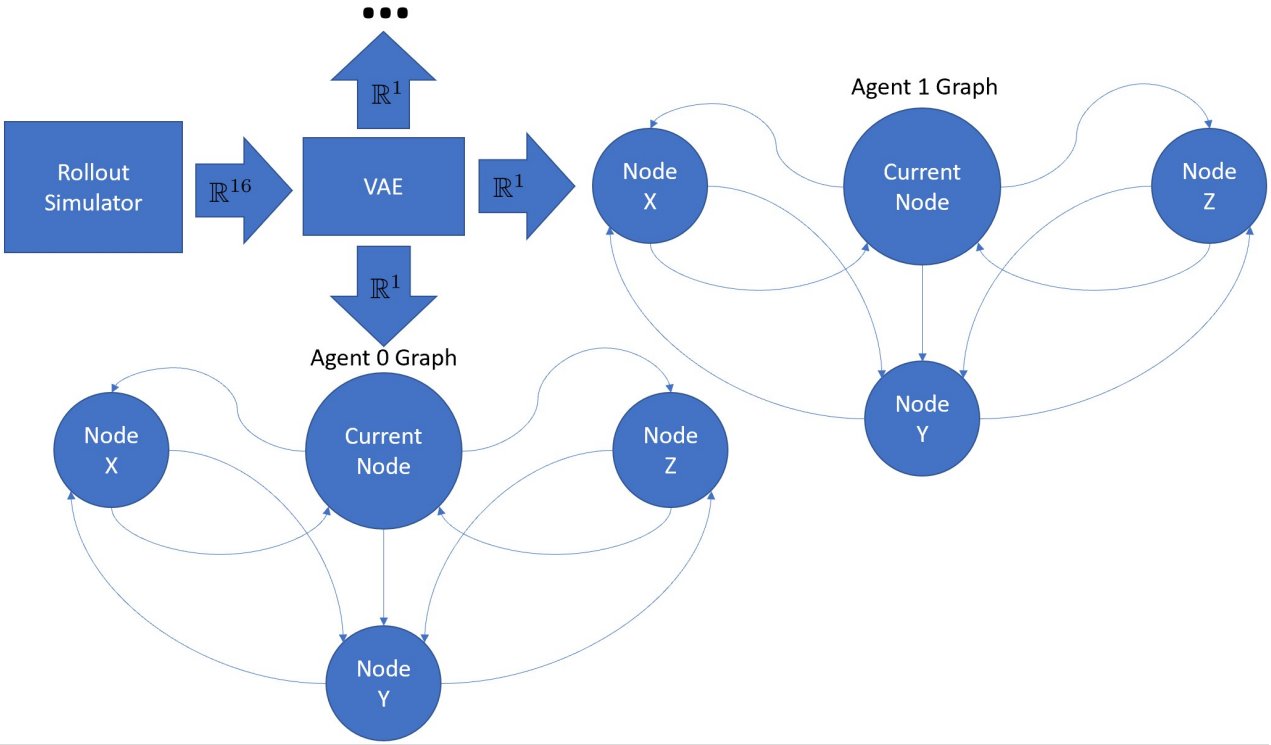


Figure 4.2: Compressing the observation space. Only the encoder of the global noise robustness Variational Autoencoder is used to compress the observation space. For this thesis the VAE accepts 4 by 4 inputs with a flattened size of 16 and compresses them to a size 1. In practice the input layer can be adapted for any n by m input and the compression variable L can have any reasonable length $l < n * m$

5 Experiments and Results

In this chapter we will describe experiments designed to compare the performance of COGNet-DICE with the state of the art. We will also show results from investigation experiments on the properties of COGNet-DICE when separately training the perception block. Subsequently, other investigation experiments will study scaling COGNet-DICE to higher-dimensional observations.

We will present the two simulators built for the signal source localization problems: a tagging simulator for a dynamic source localization problem in a one-dimensional (1D) world, and a heatmap simulator for a static source localization problem in a two-dimensional (2D) world. Using the tagging simulator we will compare the performance of COGNet-DICE and the already established algorithm COG-DICE from [Clark-Turner and Amato, 2017]. The heatmap simulator will help investigate how COGNet-DICE scales to multi-dimensional observations. Related to the 2D world, we will compare different flavours of COGNet-DICE: one that accepts raw visual inputs and the other that accepts compressed proxy inputs from a Variational Autoencoder, whose architecture and training procedure will also be described.

5.1 Comparison experiment: COGNet-DICE vs COG-DICE

To be able to compare the new COGNet-DICE algorithm with the already established COG-DICE algorithm, a simulator has been implemented based on the "tagging simulation" description from [Clark-Turner and Amato, 2017]. Since COG-DICE is meant for observations in 1D, the simulator is constrained to a space along a single line. The simulator essentially animates a moving signal emitting source that is supposed to be effectively localized by many agents.

The main results are presented in Table 5.1. As can be seen, the new COGNet-DICE algorithm reaches state of the art results. An in-depth analysis of the experimental setup and how these results were achieved will be carried out in the following sections. The in-depth discussion will refer only to experiments where $N_{nodes} = 3$ simply because it's more convenient to illustrate the results graphically for a smaller number of nodes.

Table 5.1: COGNet-DICE vs COG-DICE

N_{nodes}	COGNet-DICE policy value	COG-DICE policy value
3	1.594	1.540
6	11.103	1.794

5.1.1 Moving source localization simulator

In the attempt to answer the second research question, namely: how does the COGNet-DICE algorithm perform in a source localization problem with continuous observations and states, a simulation was developed called the "tagging simulation". The idea is as follows: there are a number of agents called taggers and one special kind of agent called an evader. As the name implies the evader tries to evade/run away from the taggers, that conversely try to tag/catch the evader. This problem is essentially a canonical source localization problem. The source is dynamic and the agents need to find its position almost exactly. The states are the agent positions in the world and they are continuous, i.e. floating point numbers on the real axis. The observations are also continuous floating point numbers, measuring the distance between the evader and each individual tagging agent. The distance is not an absolute value and can be either positive if the tagger is to the right of the evader, or negative if it stands to the left. A graphical impression about the general setup can be acquired from Figure 5.1.

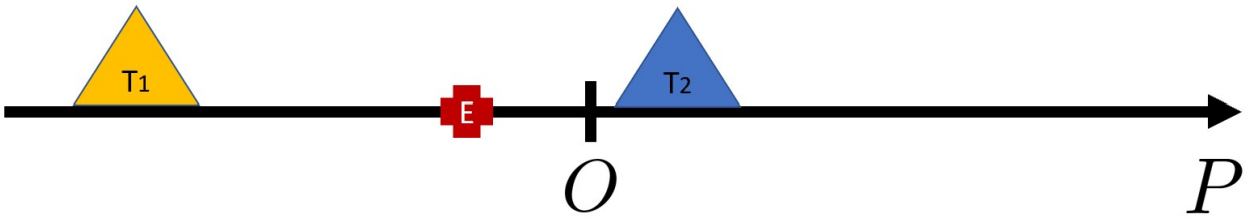


Figure 5.1: Tagging simulation setting: two tagger agents (T_1 and T_2) are trying to catch a moving evader agent (E). All agents are moving left and right with a certain configurable speed. The random variable $P \in \mathbb{R}$ denotes the state position of the agents. Additionally, the tagger agents can sample observations with a certain stochastic error that is also configurable.

Using this tool, simulations of the joint policies can be run, called rollouts. A rollout will run a horizon of h time steps and will result in accumulated experience in the form of sequences of actions, observations, nodes and discounted cumulative rewards. After a certain number of rollouts, the accumulated tuples will be sorted in the order of highest rewards and data from the top performing episodes will be used for parameter updates.

For the comparison experiments the simulator was set up such that a successful tag is one where the evader is within one unit of distance from the tagging agent. Therefore, the expected behaviour to learn is that an agent has to tag when its observation is within the interval of -1 and 1: $z_t \in [-1, 1]$ The 1D simulation world was set to have a length of 10 units and the rewards are presented in Table 5.2.

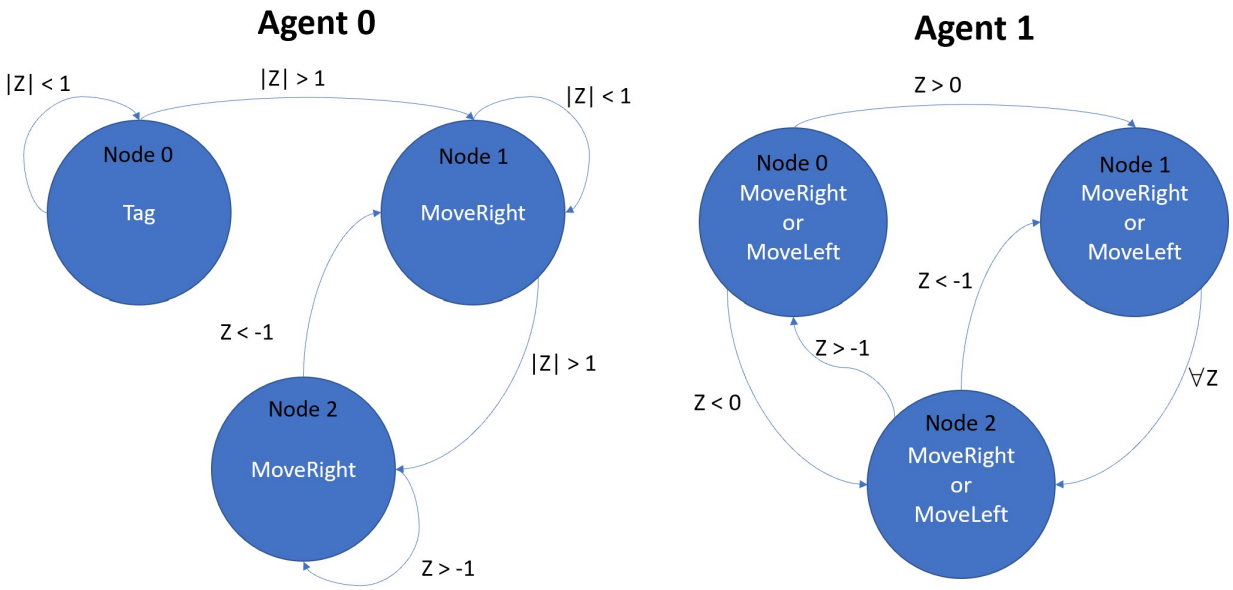
Table 5.2: Rewards on the tagging simulation problem. The state S of an agent is its position along the axis of real numbers $\mathbb{R}$. A successful tag is when the tagger performs it standing within a distance of one with respect to the evader.

Action	$\lvert S(Tagger) - S(Evader)\rvert \in [0, 1]$	$\lvert S(Tagger) - S(Evader)\rvert \in (1, 10]$
Tag	+50	-20
Move Left	-1	-1
Move Right	-1	-1

5.1.2 COGNet-DICE for one-dimensional dynamic source localization

The experiment presented in Figures 5.3 to 5.5 was set up for a problem with 2 agents and a horizon of 3. The aim was to learn a very compressed policy and hence the number of nodes was set to 3 for each agent. The experiment ran for 500 iterations and in each iteration it evaluated the top 100 joint policy candidates out of 1000 rollouts, and only 10 were kept for the policy update procedure. To avoid the likelihood of convergence to local optima, as well as increase the speed of finding good policies, the learning rate α was set to 0.2 with entropy injection enabled. This resulted in a best policy value of 1.594.

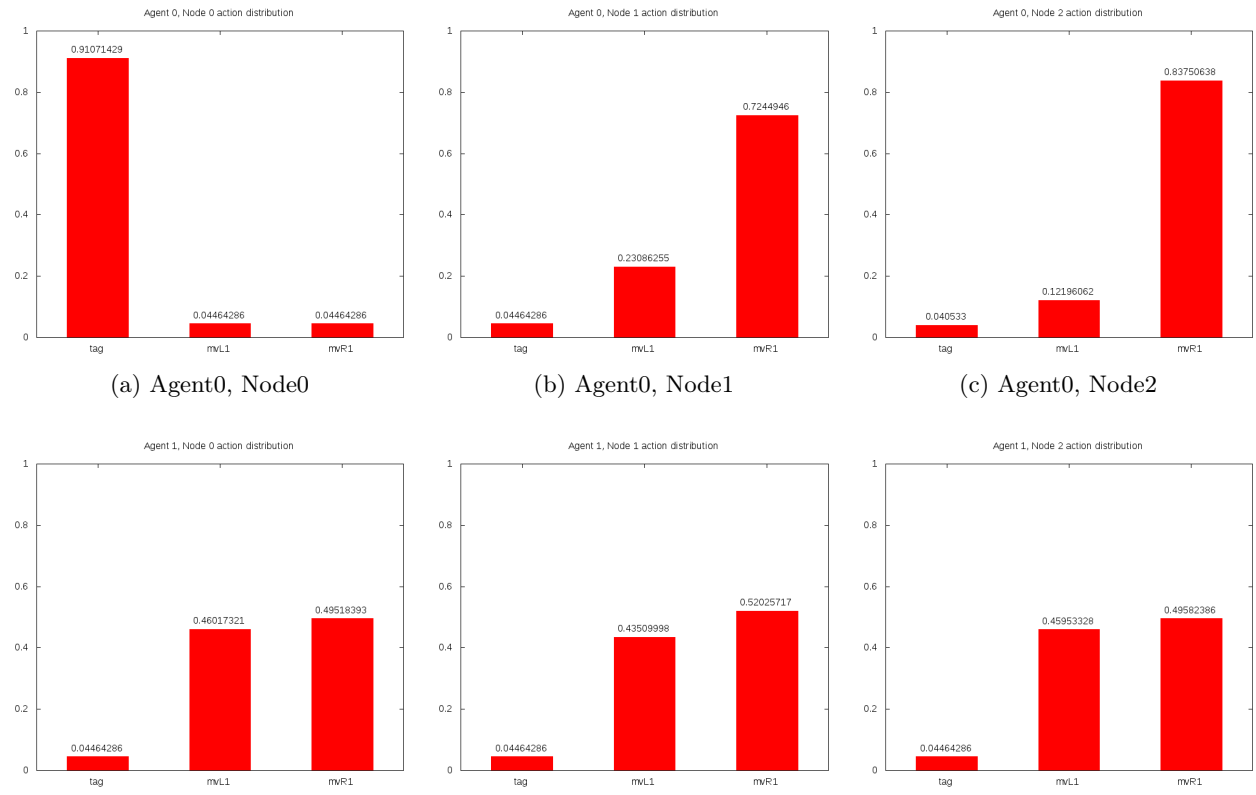
Combining the information from the action distributions in Figure 5.3 and next node distributions for each agent and node in Figures 5.4 and 5.5, we can synthesize the following controller graphs presented in Figure 5.2



(a) Agent 0 (b) Agent 1

Figure 5.2: Plain COGNet-DICE learned controllers. One agent does the tagging, and the other behaves conservatively.

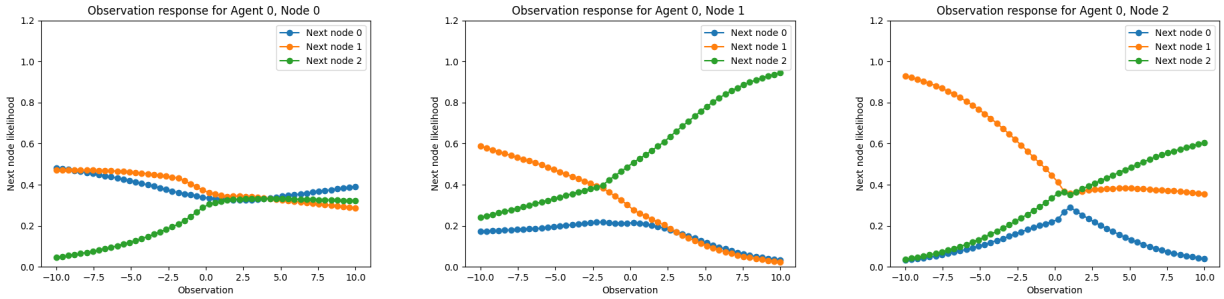
The policy learned is such that one agent will tag if it sees precisely the interval set in the simulator for a successful tag, i.e.: $[-1, 1]$. The other agent however will be more conservative and never risk tagging, because it is more safe to move left and right receiving -1 than wrongly attempting a tag that would cost -20.



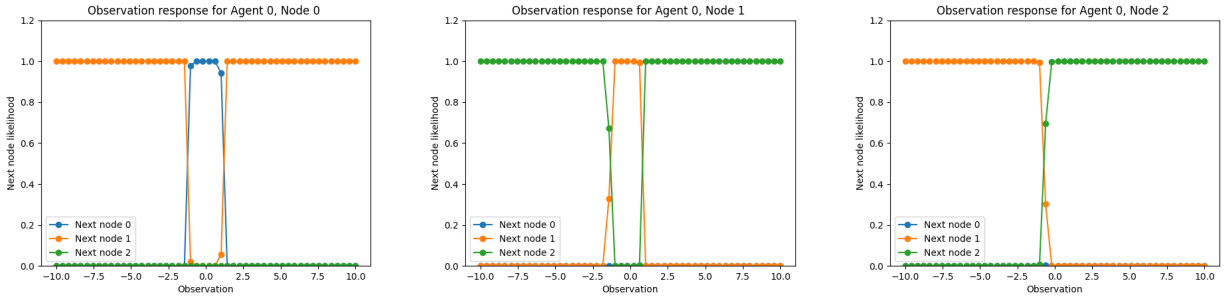
(a) Agent0, Node0 (b) Agent0, Node1 (c) Agent0, Node2

(d) Agent1, Node0 (e) Agent1, Node1 (f) Agent1, Node2

Figure 5.3: Plain COGNet-DICE action distributions for all agents and nodes.

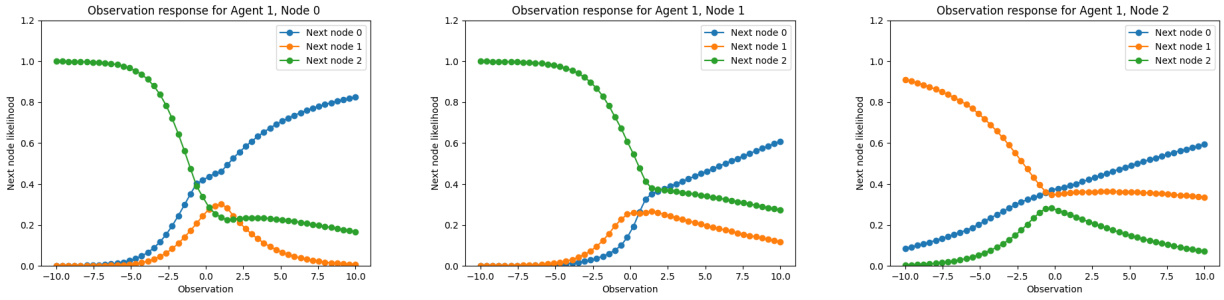


(a) Node 0 before training (b) Node 1 before training (c) Node 2 before training

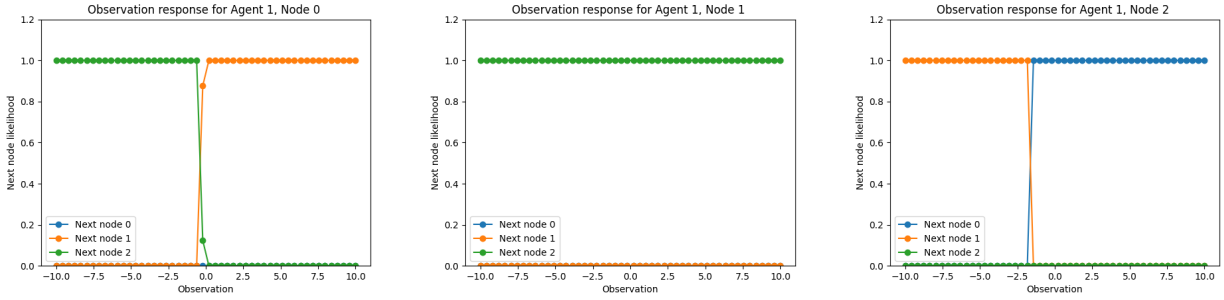


(d) Node 0 after training (e) Node 1 after training (f) Node 2 after training

Figure 5.4: Agent 0 next node distributions for plain COGNet-DICE: initial distributions on the top row, final distributions on the bottom row.



(a) Node 0 before training (b) Node 1 before training (c) Node 2 before training

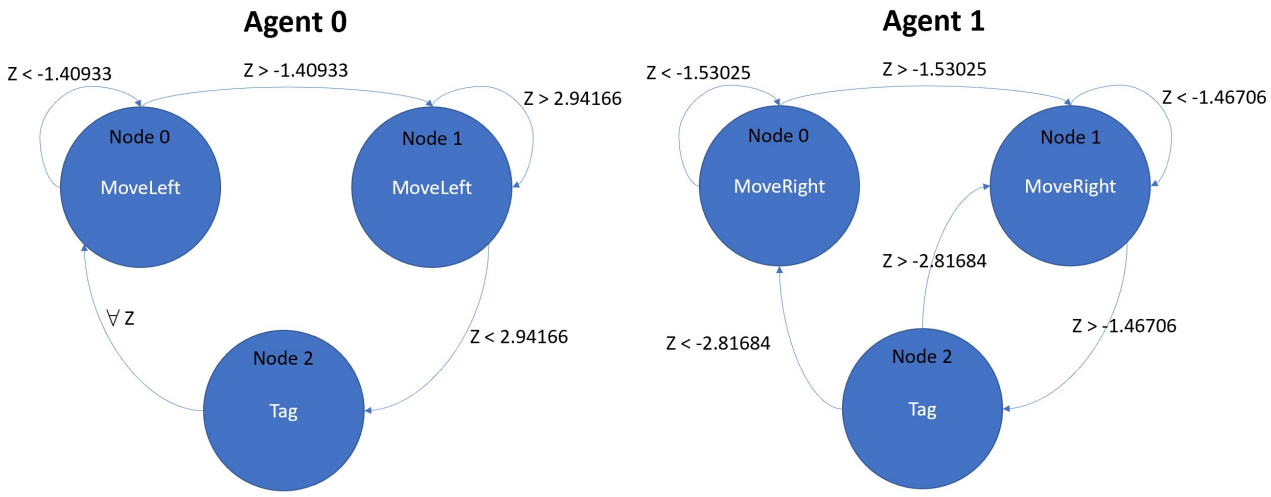


(d) Node 0 after training (e) Node 1 after training (f) Node 2 after training

Figure 5.5: Agent 1 next node distributions for plain COGNet-DICE: initial distributions on the top row, final distributions on the bottom row.

5.1.3 COG-DICE for one-dimensional dynamic source localization

The experiment above is competitive with COG-DICE by [Clark-Turner and Amato, 2017] with the same exact parameters, excepts for the number of iterations, which for COG-DICE is 50 instead of 500 used for COGNet-DICE. This is not surprising since neural networks are notoriously data-hungry. Also it should be noted that COG-DICE by [Clark-Turner and Amato, 2017] does require to specify the number of regions the observation space should be divided into. For this experiment this parameter was set to 2 with the rational that each node is forced to decide in what region it should tag, and when it should abstain from tagging. In the end, the value of COG-DICE for this setup is 1.54 as opposed to 1.594 from COGNet-DICE. Based on the information from the Tables 5.3 to 5.6 we can build the controllers for both agents (see Figure 5.6)



(a) Agent 0 (b) Agent 1

Figure 5.6: COG-DICE learned controllers. Both agents risk tagging sometimes.

It should be noted that a neural net approach such as the one in COGNet-DICE is more general in the sense that it decides on its own the number of observation regions to consider based on the seen data. On the other hand, a manual setup of this parameter like in COG-DICE also has its advantages in practice, because it represents injection of domain knowledge for faster training time and better quality solutions. We can apply therefore a random search or grid search on this parameter to find out the best value for each task individually. The purpose of a neural net approach is to avoid as much as possible domain knowledge injection. Only a simulator of the task is needed, and the network should learn just from that alone. The price to pay is that more training data is required than for model based solutions.

	Agent 0	**Agent 1**
Node 0	move left	move right
Node 1	move left	move right
Node 2	tag	tag

Table 5.3: Actions taken in the COG-DICE experiment.

	Agent 0	**Agent 1**
Node 0	-1.40933	-1.53025
Node 1	2.94166	-1.46706
Node 2	-5.35941	-2.81684

Table 5.4: Decision boundaries in the observation space for the COG-DICE experiment.

	Before division	After division
Node 0	Next node is 0	Next node is 1
Node 1	Next node is 2	Next node is 1
Node 2	Next node is 0	Next node is 0

Table 5.5: Next nodes for agent 0 as given by the COG-DICE algorithm.

	Before division	After division
Node 0	Next node 0	Next node 1
Node 1	Next node 1	Next node 2
Node 2	Next node 0	Next node 1

Table 5.6: Next nodes for agent 1 as given by the COG-DICE algorithm.

5.2 Separate training of the perception block in COGNet-DICE

A special case that works much faster for the tagging problem is when each node prefers one particular action according to a Dirac delta distribution as exemplified in Figure 5.7. The idea is for each node to be responsible only for one action and all actions to be represented by the graph. As such, the number of nodes has to be greater than or equal to the number of actions each agent is capable of, such that every action is represented by at least one node. The training becomes much faster because the task is to train the perceptual block individually. There is no joint training of the policy and perception components, but rather: we fix the planning component and train the perception system.

This approach works well in practice and is capable of reaching a policy value greater than 2 much faster than through joint training. In the 1D tagging problem this setup achieves high values within minutes of training instead of hours. It represents however injection of domain knowledge and defeats the purpose of having a generic system that can learn on its own no matter what task in front of it.

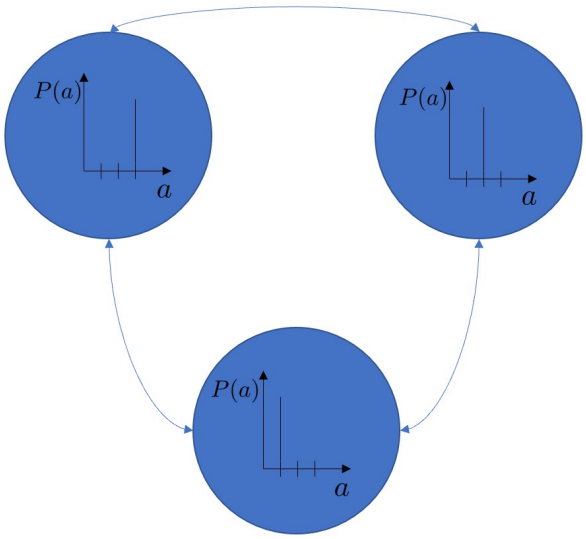


Figure 5.7: Cyclic Dirac distributions spanning the entire action set, used to always sample a deterministic action per node.

Table 5.7: Results in the special case when the perception block is trained separately. Separate training of the perception block leads to high rewards in a matter of tens of minutes instead of several hours.

N_{nodes}	Policy value
3	consistently greater than 2
6	consistently greater than 8

5.3 Multi-dimensional COGNet-DICE

This section will investigate scaling the COGNet-DICE algorithm to observations in higher dimensions. At first the 2D simulator will be described, followed by the performance of the 2D COGNet-DICE on raw images that the simulator outputs. Subsequently, the performance of the encoded version of COGNet-DICE will be shown and compared with the 2D version. The summary of the main results are presented in Table 5.8. Just as previously, only the experiments with a smaller number of graph nodes will be discussed in depth, in this case for $N_{nodes} = 4$. This is purely for the purpose of more intuitive, easier illustrations.

Table 5.8: 2D COGNet-DICE vs Encoded COGNet-DICE

N_{nodes}	2D COGNet-DICE policy value	Encoded COGNet-DICE policy value
4	44.328	43.829
5	45.001	44.346

5.3.1 2D Intensity heatmap simulator

The simulator built for higher-dimensional observations is designed to generate signals propagating in space. We assume each agent has a sensor that captures the signal and transforms it into an intensity heatmap. The overall world scene is static and has a size of 16 by 16. We assume the signal to be shaped as a Gaussian with a peak in the top-left corner of the scene as presented by Figure 5.8. Each agent is capable of only perceiving 4 by 4 patches from the overall scene as the ones from Figure 5.9. The agent always gets a reward of -1 when it detects only small amplitudes of the signal, such that a sense of urgency is created for it to find the signal source faster. When the agent reaches a position that is a radius of 4 away from the source, it receives a reward of +1 to encourage the behaviour of finding significantly higher amplitudes of the signal than previously. When it is in the immediate vicinity of the source, it receives +50. The ultimate goal is to teach all agents to find the exact position of the emitting source in a 2D space. The agents have 4 actions, i.e. move vertically: up and down and move horizontally: left and right. There is no movement in depth. The rewards are presented in Table 5.9.

Table 5.9: Rewards on the heatmap simulation problem. Radius signifies the distance from the agent to the signal emitting source.

Action	$Radius < 3$	$3 \leq Radius < 4$	$Radius \geq 4$
Move Up	+50	+1	-1
Move Down	+50	+1	-1
Move Left	+50	+1	-1
Move Right	+50	+1	-1

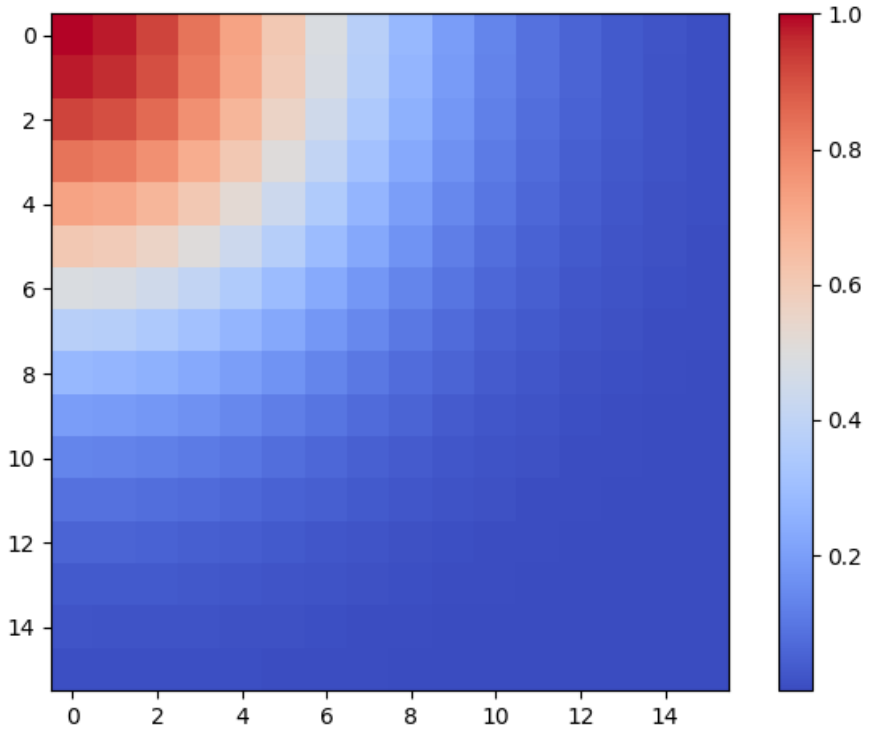
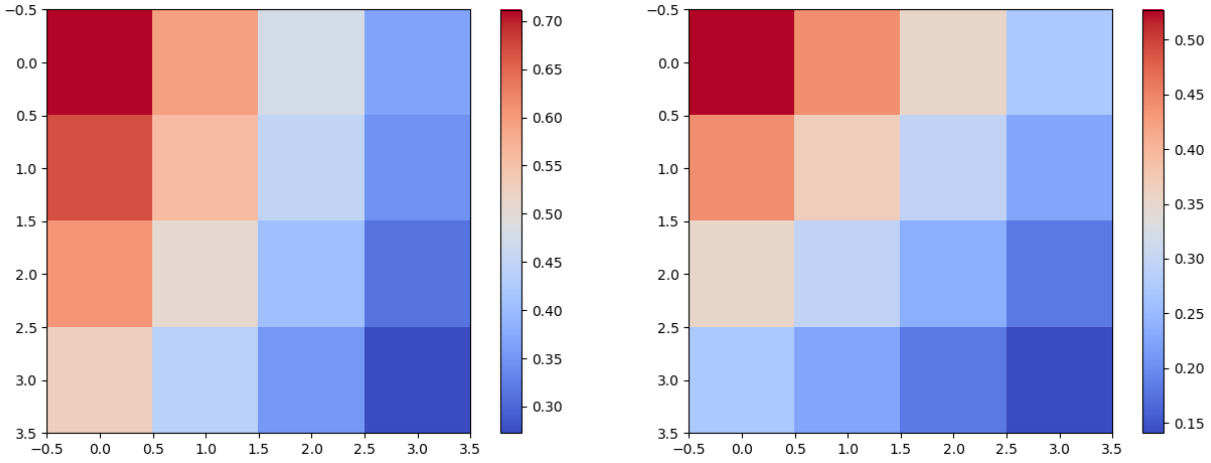


Figure 5.8: A two-dimensional scene of size 16 by 16, representing a static source in the top-left corner emitting a signal that decays from left to right according to a bell-shaped / Gaussian exponential function. The signal amplitude is normalized between 0 and 1 as shown by the color bar to the right.



(a) Vision patch 1

(b) Vision patch 2

Figure 5.9: Examples of 4 by 4 vision patches the Variational Autoencoder is trained on. These patches represent the field of vision of an agent: a 4 by 4 grid with float values representing the intensity of electromagnetic waves. The resulting images are therefore heatmaps of analogue waves propagating from a fixed source in a 2D space.

5.3.2 Training the Variational Autoencoder

The Variational Autoencoder training dataset was created by generating scenes as the one presented in Figure 5.8 and adding white noise with various means and standard deviations. Each noisy scene is swiped from left to right and top to bottom, one pixel at a time in order to crop 4 by 4 patches that an

agent is supposed to observe. In the end, many such noisy patches are generated and the Variational Autoencoder is trained on them such that the reconstructed output captures the same relative ratio between the pixel values as exhibited by the input. There is no emphasis on getting necessarily similar absolute values between the input and the output.

The architecture of the VAE encoder is as follows: one linear input layer of size 16 that accepts a flattened image patch of 4 by 4; a fully connected linear hidden layer of size 8; a latent layer of size 1, i.e μ and Σ of the latent space are both vectors with just one attribute. The decoder is symmetrical to the encoder without any extra layers.

Intuitively, the encoded system should work provided that we have a consistent and meaningful encoding of the high-dimensional space. In our case, the latent variable L seems to encode a metric proportional to the physical distance from the peak of the Gaussian heatmap as can be seen in Figure 5.10.

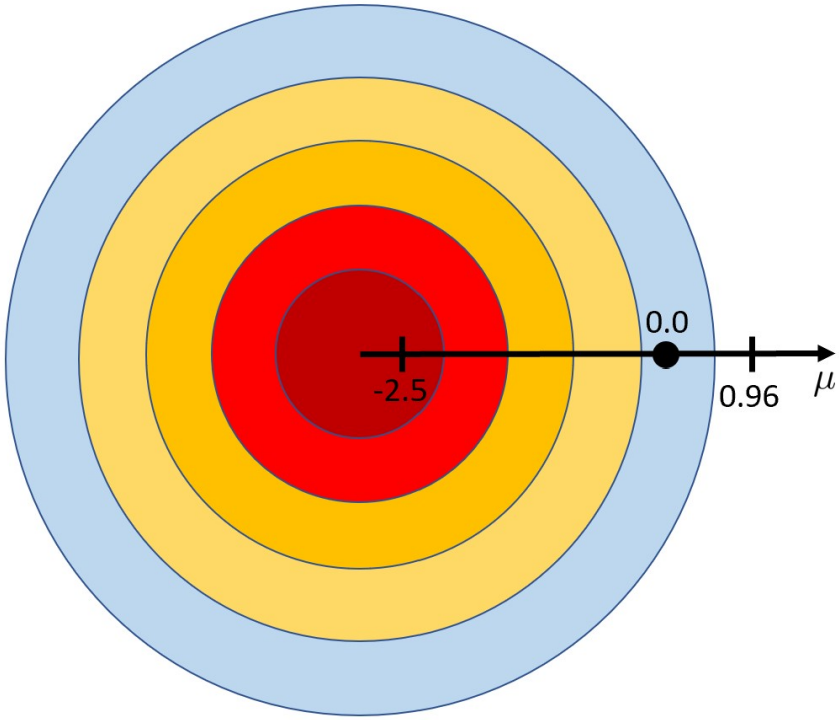
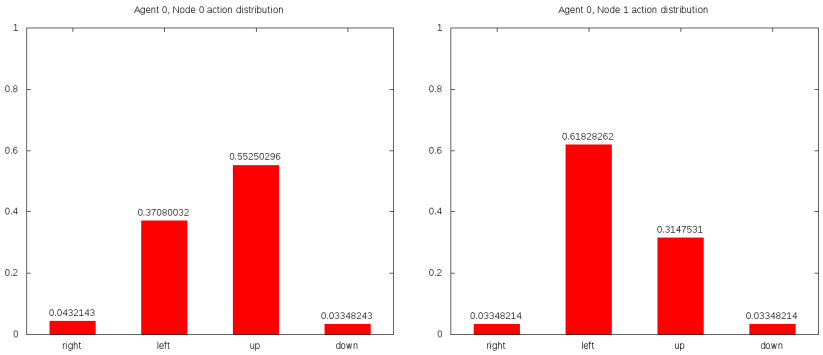


Figure 5.10: The μ component of the latent space L encodes a metric that correlates with the radial distance from the peak of the intensity heatmap.

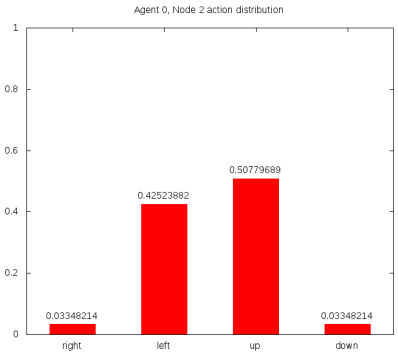
5.3.3 2D COGNet-DICE experiment

The 2D version of COGNet-DICE was set up as follows: 2 agents have to learn controllers with 4 nodes, with a lookahead horizon of 4 steps, during 500 iterations. The number of rollouts per iteration was 1000, out of which the top 100 were evaluated by calculating average rewards. From the 100 evaluated, only 10 were kept for policy improvement. The learning rate was set to 0.2 and entropy injection was allowed. Unlike the plain version of COGNet-DICE, this version has a neural net with 160 hidden layer nodes instead of 10, to accommodate for the inputs of size 4 by 4 (or flattened size of 16) instead of size 1 as it was the case for the original 1D implementation. The learned action distributions are presented in Figures 5.11. As it can be observed, the algorithm correctly identifies the action of moving up and to the left to identify the static signal source in the top left corner of the scene. In the end the value of the best policy was 44.328, which was reached at iteration 221.

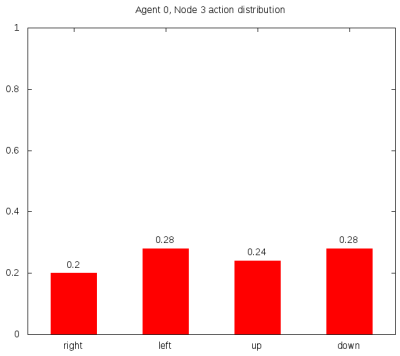


(a) Agent0, Node0

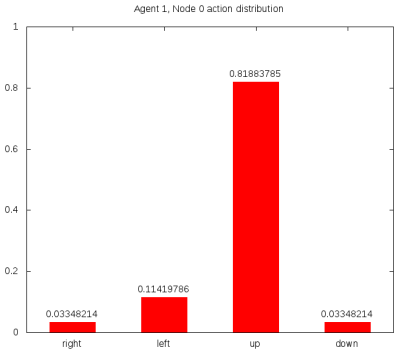
(b) Agent0, Node1



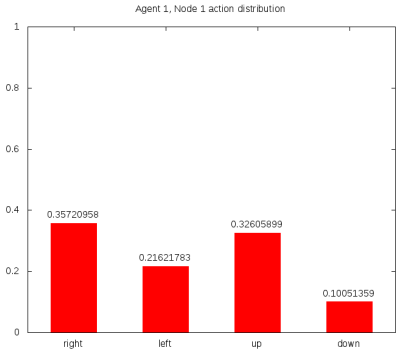
(c) Agent0, Node2



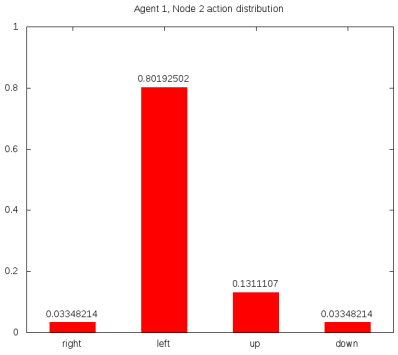
(d) Agent0, Node3



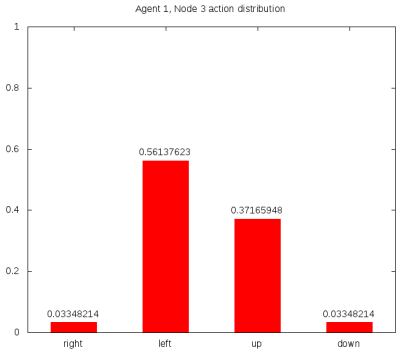
(e) Agent1, Node0



(f) Agent1, Node1



(g) Agent1, Node2



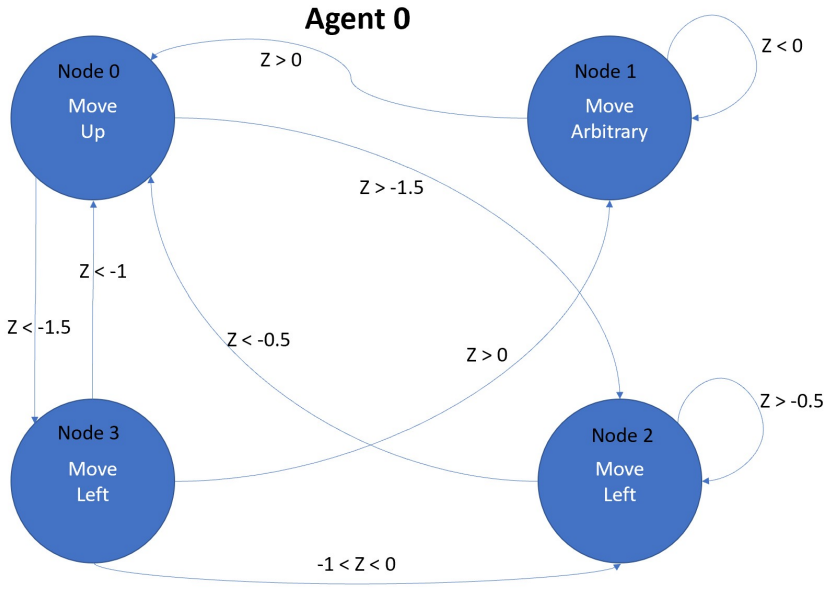
(h) Agent1, Node3

Figure 5.11: 2D COGNet-DICE action distributions for all agents and nodes. The agents correctly prefer moving left and up towards the signal source.

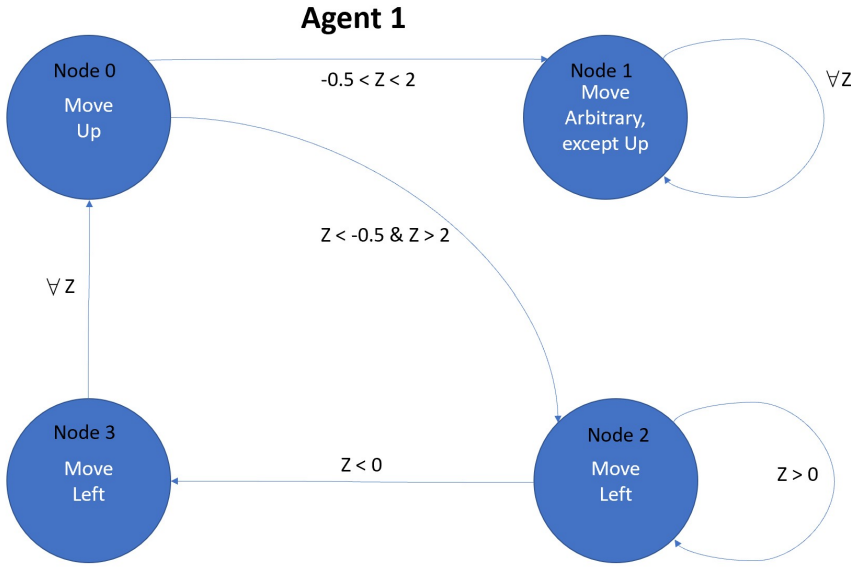
5.3.4 Encoded COGNet-DICE experiment

The encoded COGNet-DICE was ran with exactly the same parameters as the 2D version, except for the number of hidden nodes being 10 instead of 160, since the compressed input has a size of 1, a factor of 16 smaller than the raw 2D input. The value learned by the encoded version of COGNet-DICE is 43.829, which is very close to what the two-dimensional version learned using fully-sized image patches, namely: 44.328. Unlike the 2D version however, the encoded version reaches the best value at iteration 99, more than twice as fast than in the case of using raw inputs. The action distributions are shown in Figure 5.13 and it can be observed how it learns the same correct behaviour of moving mainly up and to the left. The next node distributions are presented in Figures 5.14 and 5.15.

The information from Figures 5.13 to 5.15 is summarized in the controllers from Figure 5.12. Note how when the observation goes further away from -2.5 as presented by Figure 5.10, the controllers have a tendency to explore arbitrarily in all directions. They create a sub-policy of exploration as shown by node 1 in both graphs. Getting closer to -2.5 will consistently result in greedily choosing the actions of moving up or to the left, in other words: a sub-policy of exploitation to get higher rewards.

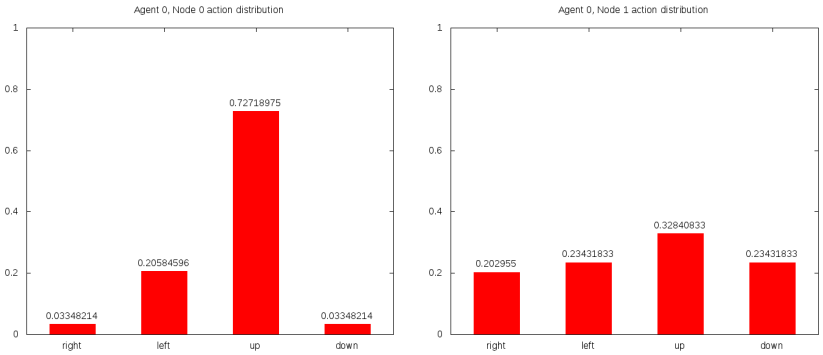


(a) Agent 0



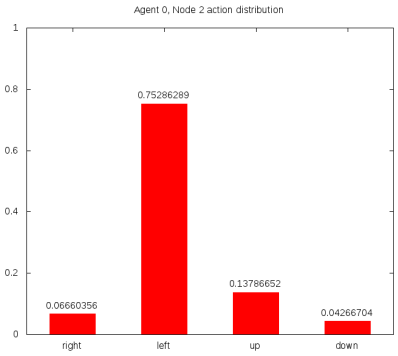
(b) Agent 1

Figure 5.12: Encoded COGNet-DICE learned controllers. Node 1 for both graphs serves as an exploration sub-policy. Other nodes describe an exploitation sub-policy to gain high rewards.

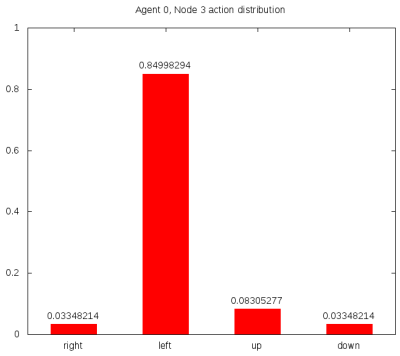


(a) Agent0, Node0

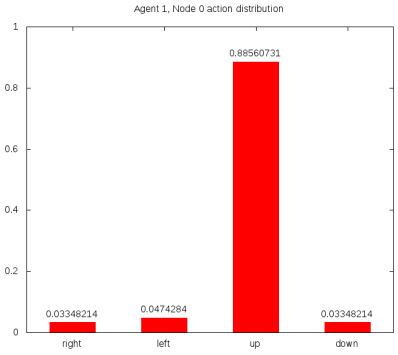
(b) Agent0, Node1



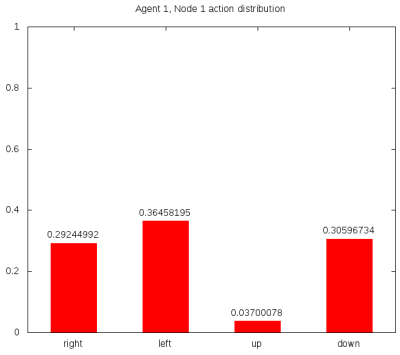
(c) Agent0, Node2



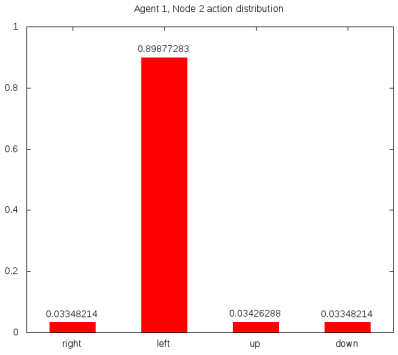
(d) Agent0, Node3



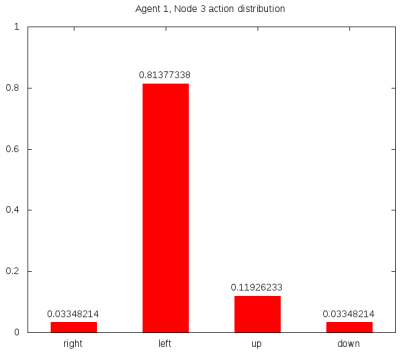
(e) Agent1, Node0



(f) Agent1, Node1

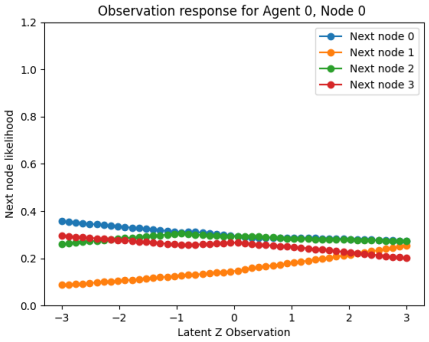


(g) Agent1, Node2

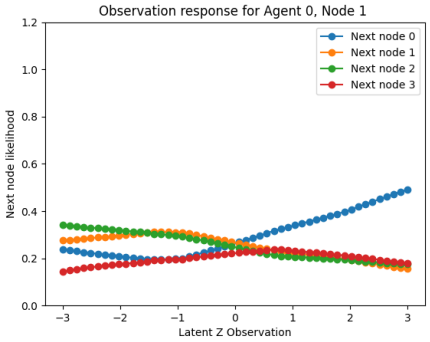


(h) Agent1, Node3

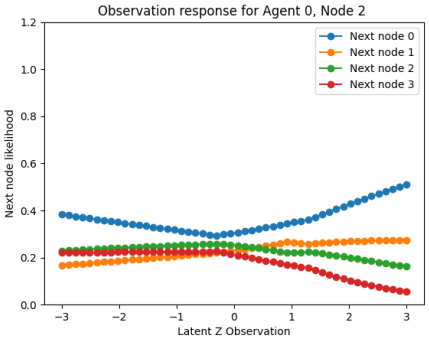
Figure 5.13: Encoded COGNet-DICE action distributions for all agents and nodes. Correct actions of moving up and to the left are preferred, just like in the case of 2D COGNet-DICE.



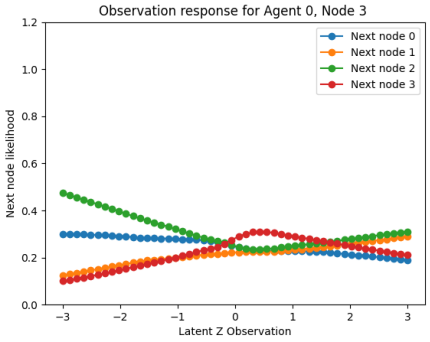
(a) Node 0 before training



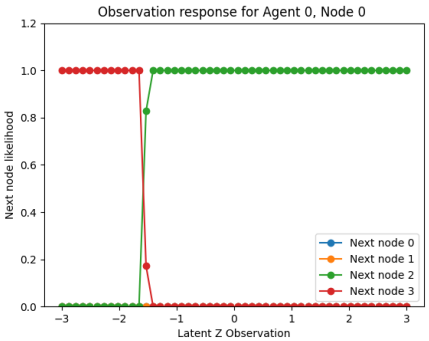
(b) Node 1 before training



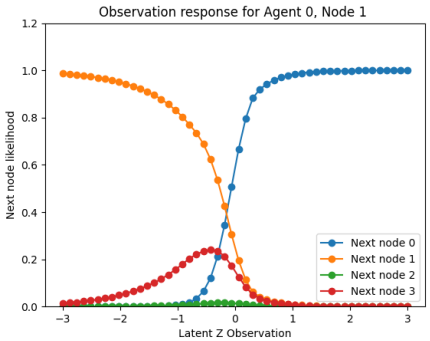
(c) Node 2 before training



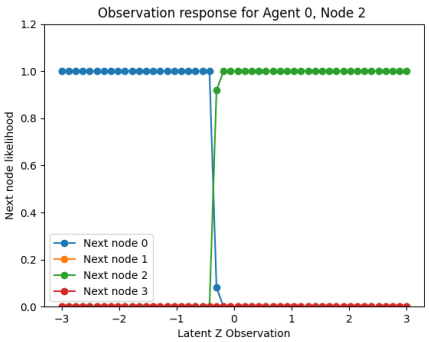
(d) Node 3 before training



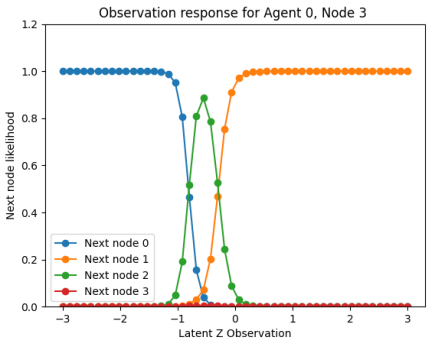
(e) Node 0 after training



(f) Node 1 after training

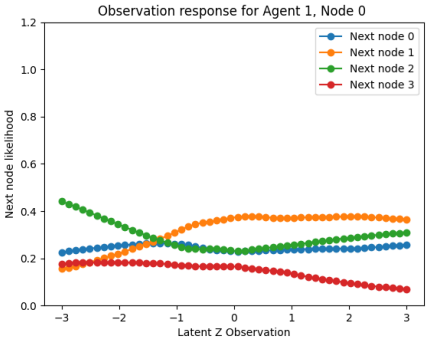


(g) Node 2 after training

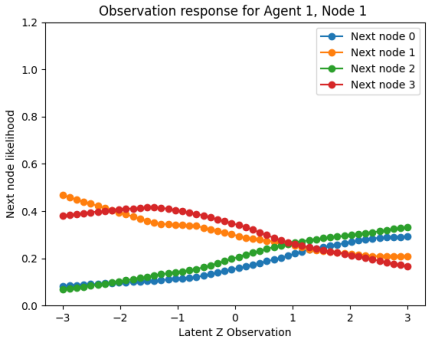


(h) Node 3 after training

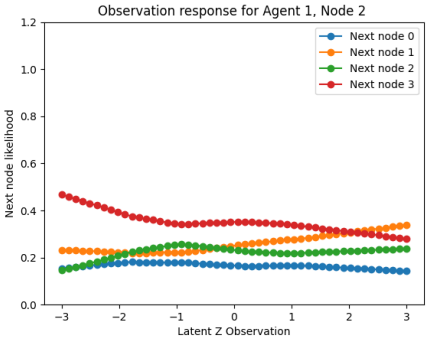
Figure 5.14: Agent 0 next node distributions using encoded COGNet-DICE



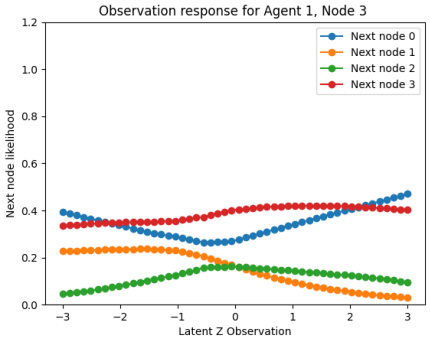
(a) Node 0 before training



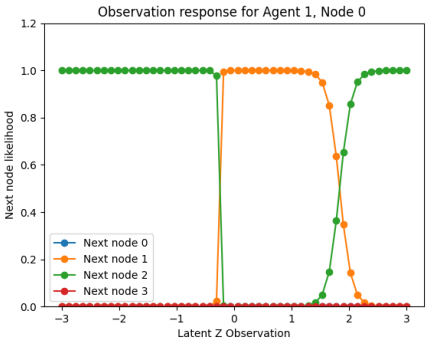
(b) Node 1 before training



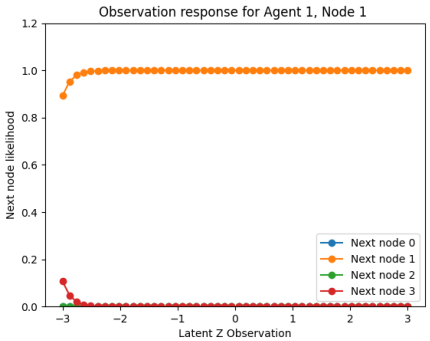
(c) Node 2 before training



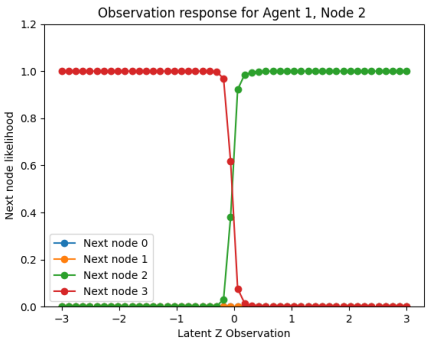
(d) Node 3 before training



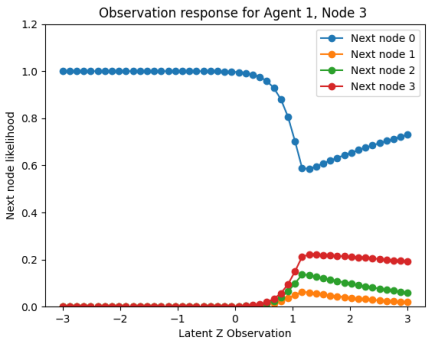
(e) Node 0 after training



(f) Node 1 after training



(g) Node 2 after training



(h) Node 3 after training

Figure 5.15: Agent 1 next node distributions using encoded COGNet-DICE

6 Conclusions

This thesis proposed a new algorithm for joint perception and planning, namely COGNet-DICE: Continuous Observation Graph Neural Net-based Direct Cross-Entropy policy search. The algorithm makes use of graphs and customized neural networks embedded in all the graph nodes, as well as a unique, global Variational Autoencoder. The algorithm is capable of handling one-dimensional continuous observation problems with many agents, and without pruning the space of policies, as well as very high-dimensional discrete observations for computer vision applications. The algorithm was verified on a dynamic source localization, as well as a static source localization problems.

One of the advantages of the neural net approach is that there is no need to specify the number of division regions in the observation space, which is a requirement for other solutions such as COG-DICE. Another important aspect is that it leverages deep learning and hence goes more in the direction of cognition than other methods using plain machine learning models. Deep learning has the advantage that it can always be scaled with deeper and more sophisticatly architectured networks, and it also emulates how natural processors, e.g. human brains adapt to new behaviour. The results mentioned in this thesis can probably be placed among the beginning of a trend where graphical models will be combined with all sorts of differently designed neural nets to reverse engineer brain-like structures. The promising thing about having graphs combined with neural nets is that they can emerge an architecture of specialized blocks just like in a human brain, as suggested by the principle of compositionality in [Lake et al., 2017] about how to build machines that learn and think like people.

The research questions posed at the beginning where answered in a rather promising way. The first question: *"Can a graph-based direct cross-entropy algorithm be combined with neural networks and successfully trained to directly couple the tasks of perception and planning ?"* has a positive answer. Combining graphs for planning and neural networks for perception results in a system that can be successfully trained in a joint, end-to-end manner. Separate training of the perception block was also attempted as an extra experiment and produced very good results. However, there is no guarantee separate training will fit any problem, and the experiment took advantage of human prior knowledge about the nature of the task involved in the experiment. Nevertheless, if offline training speed is a concern, separate training is a viable solution.

Second question: *"If yes, how does such an algorithm perform in the case of continuous observations and states ? In particular: is such an algorithm useful for source localization problems with continuous observations and states ?"* also has "yes" as an answer. The algorithm can handle continuous source localization problems successfully, just like other state of the art methods. The extra detail is that COGNet-DICE requires more training time and more data points on average. It was also observed that with more nodes in the policy graphs, very good results are usually much easier to reach, even if the training dataset is not increased substantially.

Third question was: *"Does this algorithm scale to very high-dimensional discrete inputs, such as images, when we use a lossy compression scheme derived with the help of a Variational Autoencoder ?"*. The answer is: "Yes it does". The Variational Autoencoder needs to be customized for the data it compresses. In this thesis the high-dimensional observations were reduced to a vector with just one attribute, so that the baseline COGNet-DICE could readily be used. However, the observation neural nets can easily be adapted to accept other shapes of the compressed input as well, in case it is not possible to meaningfully compress the original input to a single float number. The added benefit of compressing is not only less computation in the offline training phase, but also noise robustness in the online application phase. The Variational Autoencoder is therefore a valuable component for transfer learning to real uncontrolled environments.

Just like any other solutions, COGNet-DICE is not perfect. One of the drawbacks is that it requires a lot of training time. The speed of training and experimental results can be improved dramatically with the injection of domain knowledge. However, in its baseline form, COGNet-DICE does not require

human prior knowledge and the original intention was to avoid any type of task-related hard coding. As a result, it is able to learn on its own given enough time, with the currently attempted experiments having a duration of 1-2 days on average. Another point of trade-off with model based machine learning approaches is that COGNet-DICE does require at least 10 times as much data points to reach the same level of performance as other state of the art solutions.

One of the possible further improvements is to use experience replay. It was not attempted so far due to concerns of biasing the learning process, i.e. human-in-the-loop tampering. The experience replay trick is nonetheless justified if done right and is strongly backed by evidence from the natural world as shown by [Bendor and Wilson, 2012]. Also next to come is to use deeper and more sophisticated neural nets that would result in much better representation learning for more complicated source localization problems, as well as application to other tasks. The ultimate goal is to transfer the learning done in the simulator to embodied robots and apply it to a real world task, executed in a noisy, uncontrolled environment. Such an endeavour requires a more thorough training of the global Variational Autoencoder and more careful modeling of the noise this component injects in the training dataset of each graph node.

Bibliography

[Akkaya et al., 2019] Akkaya, I., Andrychowicz, M., Chociej, M., Litwin, M., McGrew, B., Petron, A., Paino, A., Plappert, M., Powell, G., Ribas, R., et al. (2019). Solving Rubik's Cube with a Robot Hand. *arXiv preprint arXiv:1910.07113*.

[Amato et al., 2010] Amato, C., Bonet, B., and Zilberstein, S. (2010). Finite-state controllers based on Mealy machines for centralized and decentralized POMDPs. In *Twenty-Fourth AAAI Conference on Artificial Intelligence*.

[Amato et al., 2007] Amato, C., Carlin, A., and Zilberstein, S. (2007). Bounded dynamic programming for decentralized POMDPs. In *AAMAS 2007 workshop on multi-agent sequential decision making in uncertain domains*.

[Bendor and Wilson, 2012] Bendor, D. and Wilson, M. A. (2012). Biasing the content of hippocampal replay during sleep. *Nature neuroscience*, 15(10):1439.

[Bernstein et al., 2002] Bernstein, D. S., Givan, R., Immerman, N., and Zilberstein, S. (2002). The complexity of decentralized control of Markov decision processes. *Mathematics of operations research*, 27(4):819–840.

[Bertsekas, 2019] Bertsekas, D. (2019). Multiagent Rollout Algorithms and Reinforcement Learning. *arXiv preprint arXiv:1910.00120*.

[Clark-Turner and Amato, 2017] Clark-Turner, M. and Amato, C. (2017). COG-DICE: An Algorithm for Solving Continuous-Observation Dec-POMDPs. In *IJCAI*, pages 4573–4579.

[De Boer et al., 2005] De Boer, P.-T., Kroese, D. P., Mannor, S., and Rubinstein, R. Y. (2005). A tutorial on the cross-entropy method. *Annals of operations research*, 134(1):19–67.

[Doersch, 2016] Doersch, C. (2016). Tutorial on variational autoencoders. *arXiv preprint arXiv:1606.05908*.

[Hansen et al., 2004] Hansen, E. A., Bernstein, D. S., and Zilberstein, S. (2004). Dynamic programming for partially observable stochastic games. In *AAAI*, volume 4, pages 709–715.

[Hussein et al., 2018] Hussein, A., Elyan, E., Gaber, M. M., and Jayne, C. (2018). Deep imitation learning for 3D navigation tasks. *Neural computing and applications*, 29(7):389–404.

[Lake et al., 2017] Lake, B. M., Ullman, T. D., Tenenbaum, J. B., and Gershman, S. J. (2017). Building machines that learn and think like people. *Behavioral and brain sciences*, 40.

[Lange and Riedmiller, 2010] Lange, S. and Riedmiller, M. (2010). Deep auto-encoder neural networks in reinforcement learning. In *The 2010 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE.

[Lauri et al., 2019] Lauri, M., Pajarinen, J., and Peters, J. (2019). Information Gathering in Decentralized POMDPs by Policy Graph Improvement. In *Proceedings of the 18th International Conference on Autonomous Agents and MultiAgent Systems*, pages 1143–1151. International Foundation for Autonomous Agents and Multiagent Systems.

[Law and Gold, 2009] Law, C.-T. and Gold, J. I. (2009). Reinforcement learning can account for associative and perceptual learning on a visual-decision task. *Nature neuroscience*, 12(5):655.

[Levine et al., 2016] Levine, S., Finn, C., Darrell, T., and Abbeel, P. (2016). End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research*, 17(1):1334–1373.

[Levine et al., 2015] Levine, S., Wagener, N., and Abbeel, P. (2015). Learning contact-rich manipulation skills with guided policy search (2015). *arXiv preprint arXiv:1501.05611*.

[Lillicrap et al., 2015] Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., Silver, D., and Wierstra, D. (2015). Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*.

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

[Mnih et al., 2015] Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M. G., Graves, A., Riedmiller, M., Fidjeland, A. K., Ostrovski, G., et al. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

[Muratore et al., 2019] Muratore, F., Gienger, M., and Peters, J. (2019). Assessing Transferability from Simulation to Reality for Reinforcement Learning. *IEEE transactions on pattern analysis and machine intelligence*.

[Nair et al., 2003] Nair, R., Tambe, M., Yokoo, M., Pynadath, D., and Marsella, S. (2003). Taming decentralized POMDPs: Towards efficient policy computation for multiagent settings. In *IJCAI*, volume 3, pages 705–711.

[Oliehoek et al., 2016] Oliehoek, F. A., Amato, C., et al. (2016). *A concise introduction to decentralized POMDPs*, volume 1. Springer.

[Oliehoek et al., 2008a] Oliehoek, F. A., Kooij, J. F., and Vlassis, N. (2008a). A cross-entropy approach to solving Dec-POMDPs. In *Advances in Intelligent and Distributed Computing*, pages 145–154. Springer.

[Oliehoek et al., 2008b] Oliehoek, F. A., Kooij, J. F., and Vlassis, N. (2008b). The cross-entropy method for policy search in decentralized POMDPs. *Informatica*, 32(4):341–357.

[Oliehoek et al., 2008c] Oliehoek, F. A., Spaan, M. T., and Vlassis, N. (2008c). Optimal and approximate Q-value functions for decentralized POMDPs. *Journal of Artificial Intelligence Research*, 32:289–353.

[Omidshafiei et al., 2016] Omidshafiei, S., Agha-Mohammadi, A.-A., Amato, C., Liu, S.-Y., How, J. P., and Vian, J. (2016). Graph-based cross entropy method for solving multi-robot decentralized POMDPs. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 5395–5402. IEEE.

[Omidshafiei et al., 2017] Omidshafiei, S., Amato, C., Liu, M., Everett, M., How, J. P., and Vian, J. (2017). Scalable accelerated decentralized multi-robot policy search in continuous observation spaces. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 863–870. IEEE.

[Oord et al., 2016] Oord, A. v. d., Kalchbrenner, N., and Kavukcuoglu, K. (2016). Pixel recurrent neural networks. *arXiv preprint arXiv:1601.06759*.

[Pajarinen and Peltonen, 2011] Pajarinen, J. K. and Peltonen, J. (2011). Periodic finite state controllers for efficient POMDP and DEC-POMDP planning. In *Advances in Neural Information Processing Systems*, pages 2636–2644.

[Papadimitriou and Tsitsiklis, 1987] Papadimitriou, C. H. and Tsitsiklis, J. N. (1987). The complexity of Markov decision processes. *Mathematics of operations research*, 12(3):441–450.

[Riedmiller, 2005] Riedmiller, M. (2005). Neural fitted Q iteration–first experiences with a data efficient neural reinforcement learning method. In *European Conference on Machine Learning*, pages 317–328. Springer.

[Szer et al., 2012] Szer, D., Charpillet, F., and Zilberstein, S. (2012). MAA*: A heuristic search algorithm for solving decentralized POMDPs. *arXiv preprint arXiv:1207.1359*.

[Van den Oord et al., 2016] Van den Oord, A., Kalchbrenner, N., Espeholt, L., Vinyals, O., Graves, A., et al. (2016). Conditional image generation with pixelcnn decoders. In *Advances in neural information processing systems*, pages 4790–4798.

Eidesstattliche Versicherung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe angefertigt und mich anderer als der im beigefügten Verzeichnis angegebenen Hilfsmittel nicht bedient habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.
Ich bin mit einer Einstellung in den Bestand der Bibliothek des Fachbereiches einverstanden.

Hamburg, den _____ Unterschrift: _____