Partially Observable Markov Decision Processes

Beata Baczyńska

Abstract
In real life, partially observable problems are very common. Solving these problems, with the state of uncertainty, is more complicated than solving fully observable problems. The main reasons are the cures of dimensionality and the curse of histories that often make problems to be computationally intractable. In this paper, three different methods will be presented. Each introduced completely different ideas and improvements.

The methods presented are: Deep Recurrent Q-network approach, POMCP a Monte Carlo-based approach and QMDP-Net.

Introduction

There are many algorithms that are able to solve problems modeled as Markov Decision Processes. However, often it's very common for real-life problems to be only partially observable. For example in robotics, like autonomous vehicles when we are able to perceive only a limited set of information or when the robot has a limited number of sensors or these sensors are imperfect. These partially observable problems we can not solve as MDP. These kinds of problems we need to model with POMDP framework.

A Partially observable Markov decision process (POMDP) is an MDP with state uncertainty. A POMDP is used to model problems where the results of actions are nondeterministic and the state is only partially observable so we don't know the true state. We only receive the observations of the states and based on that we have a belief - the probability distribution over states (in MDP observation equals state, as we have a fully observable system). In POMDP it's possible that agents can perceive the same observations when being in different states.

We can formalize these types of problems with a tuple (S,A,T,R,Z,O), where S,A,T,R are states, actions, transitions and rewards are known from the MDP process and Z and O are the observation function (a conditional probability function that represents the probability of observing the agent may perceive when it is in state s after executing action a) and observation space (the set of all observations the robot can perceive) respectively. The belief defines the probability of being in a state s according to history of actions and observations.

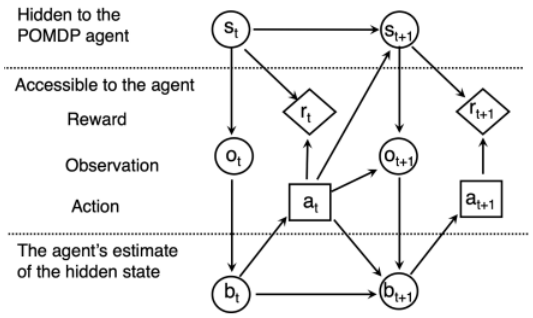


Figure 1. Process under the POMDP. Source [1]

The mechanics of the POMDP system is as follows (see Figure 1): the agent is in some (hidden) state s_t, it perceives some observations o_t and as a result maintains a belief b_t as an estimate of its state. Based on the current belief the agent executes some action a_t and as a result the hidden state s_t may change to a new, also hidden, state s_{t+1}. The agent also perceives a new

observation o_{t+1} and the reward r_t based on the reward function R. The possible state s_{t+1} and the observation it may perceive follows the transition T and observation functions Z. Since the state s_{t+1} is also hidden, the agent updates its estimate of the state from belief b_t to belief b_{t+1} via Bayesian inference (see Equation 1) based on the previous estimate b_t, the action a_t it just performed, and the observation o_{t+1} it just perceived.

$$b_{t+1}(s_{t+1}) = \tau(b_t, a_t, o_{t+1}) = \eta Z(s_{t+1}, a_t, o_{t+1}) \sum_{s \in S} T(s, a_t, s_{t+1}) b_t(s) \tag{1}$$

Curse of dimensionality For MDP problems with the Bellman equation, we need to estimate the expected reward for each of the states. When the number of states is large then even for MDP it may be difficult to cover all the states. However, for POMDP instead of computing the best action for the state, we need to compute the best action for belief. And as the belief is a probability distribution over states, the belief space is an (n - 1) dimensional continuous space. For example for two real states, we have a one-dimensional continuous belief space between 0 and 1 that assigns the probability of being in one of these states. We can see that even for a problem with two hidden states when using the Bellman equation for the belief states the number of belief states to cover would be much larger. And as we usually have problems with much more states, the belief state space, even when using some discretization, would be too huge to compute the optimal solution. We name it the 'curse of dimensionality and that curse is one of the reasons why the POMDP problems are computationally intractable. That means that even if the methods for finding the optimal policy exist, the computational complexity is too high to use them in practice.

Curse of history To calculate with good accuracy a long-term return, we can perform a k-steps look ahead of consequences of actions (collect action-observation histories). The longer the history the more accurate the predictions. Unfortunately, the number of possible histories grows exponentially with the number of k. The exponential growth of the size of space of all possible POMDP histories makes it impossible to simulate distant consequences. We call this problem a 'curse of history'.

The curse of dimensionality and the curse of history are two main reasons why finding the optimal solution for POMDP problems is so difficult.

In section 2, section 3 and section 4 different methods for POMDP framework will be explained.

1. Background

1.1 Online planning methods

In the offline planning methods, the best action for each possible state is calculated in advance. And then, during execution, the agent only needs to estimate its current belief and execute the best action for that belief. For large POMDP this idea doesn't seem to work well as the number of all possible belief states is too large and it would be too many computations to make in advance. For this reason, the online methods seem to be more efficient as the best action is calculated only for the current belief state. This allows us to avoid many unnecessary computations as we consider only the belief states that we are able to reach from the current one. We start in the current belief state, and in that state, we need to compute the best action to take. This can be done for example in the same way as for the offline methods, so we can compute the estimated reward for all belief states that are reachable from the current one and then select the best one we have found, or by using a black box simulator (what will be described more detailed in section 3)). As the best action was founded we can execute it and as a result, the agent will perceive a new observation, update its belief, and the process will repeat for the next action prediction.

1.2 Particle filter

The particle filter is a filtering method based on Monte Carlo and recursive Bayesian estimation.

It is often used for object tracking problems like car tracking and distance estimation or people detection and tracking. It estimates efficiently the state of a dynamic system from a series of measurements. Particle filtering can be used to update the model state variables each time a new observation is available (so we can improve our estimate of the state). The more accurately we know our state the better actions our agent is able to select.

On the Figure 3 we can see three stages of the particle filter algorithm: Selection, Prediction, and Measurement.

In the selection stage, we generate a new particle set according to the probability distribution.

In the prediction step, for all the particles the simulation of executing some specific action is made. Here, important is that the simulation is made only for the generated particles not for all possible states. Thanks to that the curse of dimensionality is broken, as the number of simulations to be made don't depend on the number of possible states but only on the number of particles we generated. So it can be also constant. Next, for all the particles, we estimate what would be the observations after a specific action is executed. Then we compare observations received from the simulations with observations actually received. The more similar the observations are the higher particle's weight. And based on the weight a new probability distribution over beliefs is generated and the process is repeated.

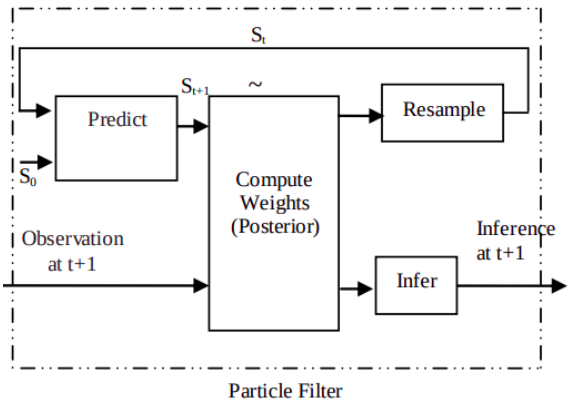


Figure 2. Particle filtering algorithm - block diagram. Source [2]

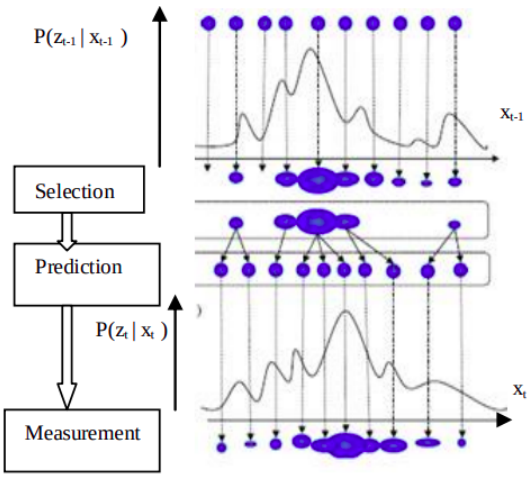


Figure 3. Particle filtering algorithm - main steps. Source [2]

1.3 Monte Carlo Tree Search

Monte Carlo Tree Search is a well-known approach to online planning (see subsection 1.1) that has exceptional performance in large, fully observable domains. The MCTS was used in both AlphaGo Fan - the first neural network that was able to win with professional Go players and the AlphaGO Zero - the first neural network that was trained without any human knowledge or hand-crafted features and outperformed all previous AlphaGo versions [3] [4]. The MCTS is doing a selective tree search (best-first order), and thanks to that computation time of the search decrease (doesn't visit all possible nodes), as well as the curse of dimensionality, is broken. That makes it possible to search deeper in the tree within an acceptable time limit.

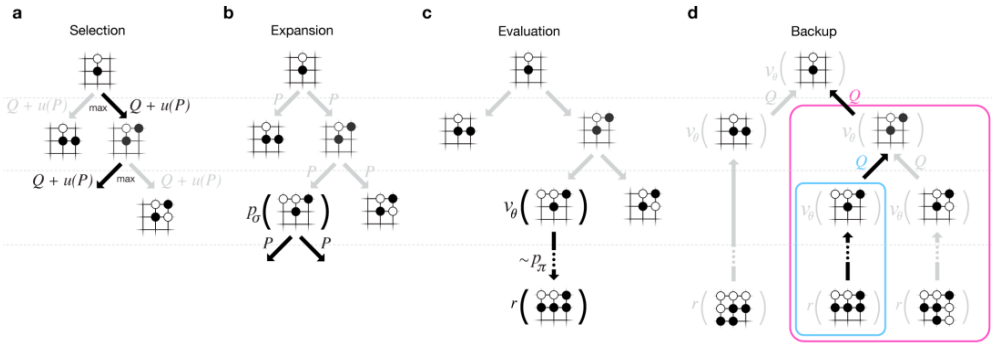


Figure 4. Monte Carlo Tree Search process. The example on Go game. Source [3]

Once all actions from state s are represented in the search tree, the tree policy selects the action maximizing the Q value, otherwise, the actions are selected according to the rollout policy (see: subsubsection 1.3.2).

The final reached value is then forwarded to the top of the tree and Q values are updated on each of the nodes (Figure 4). Each simulation ends up with a new node added.

The MCTS is also an anytime algorithm which means that we can stop the search at any time and always receive some result, we do not need to wait till the search finish. However longer we explore the tree the more accurate the results are.

1.3.1 UTC algorithm

The improvement for the MCTS approach where the actions are selected according to Q values is the UTC algorithm. The UTC algorithm introduces different action selections in MCTS, where the actions are augmented by an exploration bonus that increases the Q value for rarely visited actions.

$$Q^*(s,a) = Q(s,a) + c\sqrt{\frac{logN(s)}{N(s,a)}} \tag{2}$$

The scalar constant c determines the relative ratio of exploration to exploitation (c = 0 means no exploration).

In this UTC version of MCTS next actions are selected based on the Q^* value (calculated with the Equation 2). This helps us to try actions that were rarely visited before (and maybe by visiting them a few times more the approximation of the Q value will be more accurate).

1.3.2 Rollout

The fast way to evaluate the state is to perform a rollout. It means that the next states, till a terminal state or discount horizon, is reached) are selected according to the rollout policy. This policy is often a random policy as the rollout aims to approximate the expected return of a state (and so it can proceed randomly).

2. Deep Recurrent Q-network approach

2.1 Solved problem

Many Atari games are POMDP when taking into account only single frame. These games are for example Pong when with only one frame we only know about position of the ball, but we don't know the speed neither direction. To solve these games as MDP we can just input four consecutive frames. In this way neural network is able to extract missing information about speed and direction so as a result to problem is fully observable. That is the easiest and the most popular solution for that kind of problems.

Figure 5. Some of the atari games with moving object (POMDP when only single frame is provided). Pong; Frostbite; Double Dunk. Source [5]

The method was addressed especially for problems when longer history would be needed (or even the history of unknown length) to model the POMDP problem as MDP. Authors also compared the efficiently of the approach with set of frames as the input (MDP) with the approach where only single frame is given (POMDP) but the model is using recurrency (so the information from previous time steps is also provided).

2.2 Method description

In the paper [5] authors used extra recurrent layer to provided the knowledge about past events without the need to hard-code the number of time steps we remember about.

Authors modified the DQN architecture in the minimum possible way to extract the effect of recurrency. Both compared architectures (DRQN and DQN) were trained from scratch.

It was shown that the DRQN performs good even with only one frame per time step. In that case the information about speed and direction was not extracted by convolution layers (what has place in standard DQN), but by the high-level recurrent layer.

The experiment with non-recurrent 10-frame DQN and recurrent 1-frame DRQN that was trained using backpropagation through time for the last ten time steps (both have access to the history of the same length) shown that the recurrent network can be used as a alternative for currently used DQN with sequence of frames on input (as the result were similar, depending on the game sometimes the score was higher with DQN sometimes with DRQN and sometimes the difference was between std tolerance).

2.3 Conclusion

The results shown that for problems where information from finite number of time steps are sufficient to model the POMDP problem as MDP, providing information from all these time steps as an input at once and process with DQN can result in similar results as when providing only one frame from single time step but when using recurrency (DRQN).

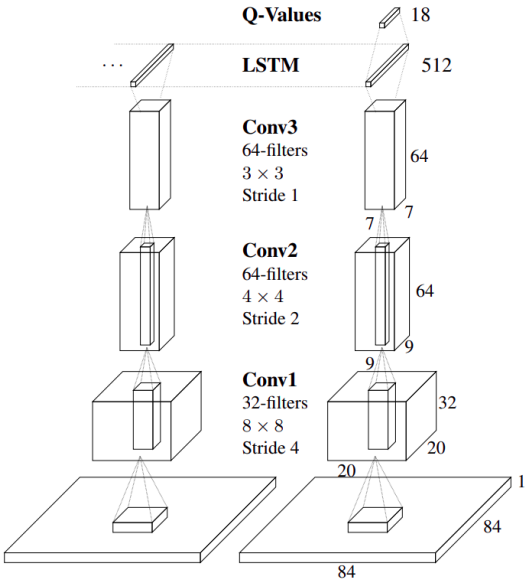


Figure 6. Architecture of the DRQN network. Last two time steps we shown. Through LSTM layer the information between time steps are passed. Source [5]

3. POMCP - Monte Carlo-based approach

3.1 Solved problem

Because of the curse of dimensionality and curse of history it is computationally infeasible to find an optimal policy in especially in large POMDP. The method was addressed to large POMDP problems. It was able to break the curse of dimensionality as well the curse of history. As a result it was able to converges to the optimal policy for any finite horizon POMDP.

3.2 Method description

David Silver and Joel Veness in the [6] paper introduced Monte-Carlo algorithm for online planning in large POMDPs.

In that paper the Monte Carlo Tree Search method (see: subsection 1.3) has been extended to POMDP environments. The method also uses UCT search to select action at each time step (with the formula Equation 3) and a particle filter that updates the agent's belief state (see: subsection 1.2). Belief state has been represented as a set of particles and by doing the sampling the curse of dimensionality has been broken (as we do not consider all the possible belief states). Also the curse of history has been broken by sampling histories using a black-box simulator.

3.2.1 Search Tree of histories

Instead states nodes (subsection 1.3) the nodes of histories were introduced. Analogically to the MDP MCTS each node consists of the counter of how many times the history has been visited and the value of history that is estimated by weighted mean of all simulations starting from the current history. To decide which is the next node to visit again the UTC algorithm was used (Equation 3).

$$V^*(h,a) = V(h,a) + c\sqrt{\frac{logN(h)}{N(ha)}} \tag{3}$$

At the end of search tree procedure the rollout procedure is executed and then the final value is calculated and forwarded to the top of the created path.

In small POMDP the belief space is often updated by Bayes' theorem. For large POMDP that is the part that makes problem to be computationally infeasible. To handle to computations also for large POMDP an approximation method was introduced. The belief state was approximated by using the particle filter (see subsection 1.2). To break the curse of histories, the black box simulator was used (that reduced the computational complexity).

3.2.2 Black box simulator

A generative model of the POMDP was used as a simulator, so it was not necessary to know explicitly the model of the system. When the state and action was given, the simulator outputs the successor state, observation and reward ($G(s_t, a_t) \sim$

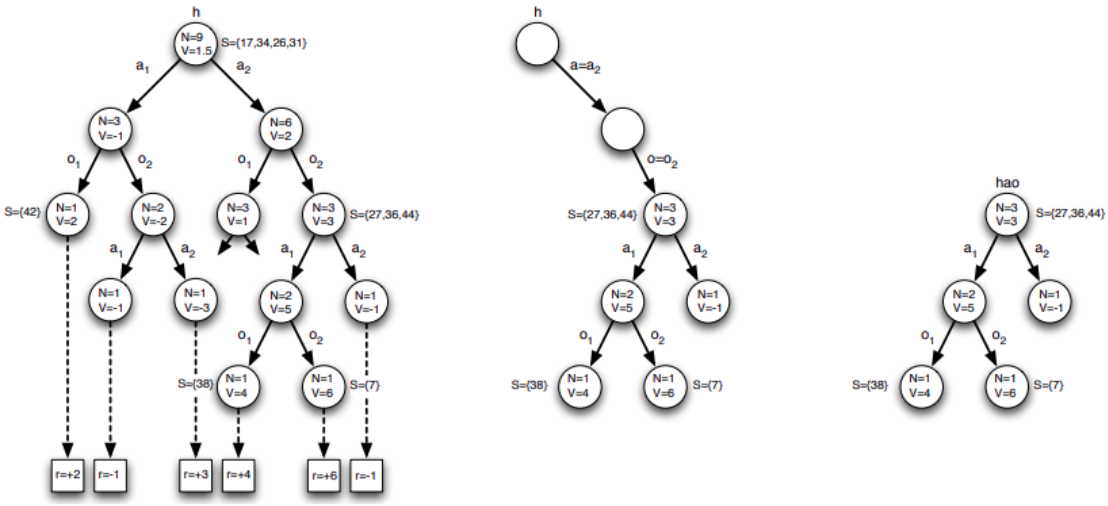


Figure 7. Monte Carlo Tree Search for POMDP problems. Source [6]

$(s_{t+1}, o_{t+1}, r_{t+1})$). The simulator was used to generate sequences of states, observations and rewards. These were later compared with real outputs of the system so the agent could better approximate his belief state.

3.3 Conclusion

The main advantage of this method is that with appropriately controlled exploration of belief states it is able to converges to the optimal policy even for large POMDP (with finite horizon). By using the particle filtering we can prevent to complexity to grow with number of belief states, what breaks the curse of dimensionality. Also the curse of histories was broken by using the black box simulator when estimating the state, observation and reward within the the MCTS.

4. QMDP-Net

4.1 Solved problem

The method to learn policy, for the POMDP problems, that will generalize well also for the previously not seen tasks.

4.2 Method description

In this example, the task is parameterized what means that the values of the task can change what will result in different tasks to solve. The parameters may define for example how the environment looks like or what is the goal of the agent. By embedding these parameters inside the model we may be able to learn a policy that depends on these parameters and as the result, the agent will perform well for all the possible parameter values (so for all the possible environments and all possible goals that can be defined).

However, the assumption here is that the underlying state space, action space, and observation space must remain the same.

The main difficulty of the problem is that we would like to learn the policy that will work well also for previously not seen tasks. So we want to learn a policy that will fit for all the possible configurations of the parameters without trying all of them beforehand. The proposed solution for training the agent in that way is to embed the task specification inside the trained network. So all the components: the model and the algorithm are trained together in a single, fully differentiable neural network in an end-to-end way. That allows the policy to generalize over different model settings. That results in good approximation and good generalization to different task settings.

A QMDP-net is a recurrent policy network. The network's architecture is presented at the section 4. It consists of a Bayesian filter (see Equation 1 and subsubsection 4.2.1), that integrates the history of an agent's actions and observations into a belief. Second, represents the QMDP algorithm (as described in [7]), which chooses the action given the current belief. It does it by solving the problem as a fully observable MDP and performing a one-step look-ahead search on the MDP values weighted by the belief.

As we can embed the algorithm in a neural network and perform the same operations by matrix multiplication (all linear operations), convolutional layers (all the summations), and max-pooling (to find maximum), we can perform end-to-end training that will provide good generalization over all possible tasks in POMDP.

The training was performed with set of expert trajectories (sequence of demonstrated actions and observations $(a_1, o_1, a_2, o_2, ...)$ for some specific task) and randomly chosen task parameters. The agent didn't know the ground-truth states or beliefs defined

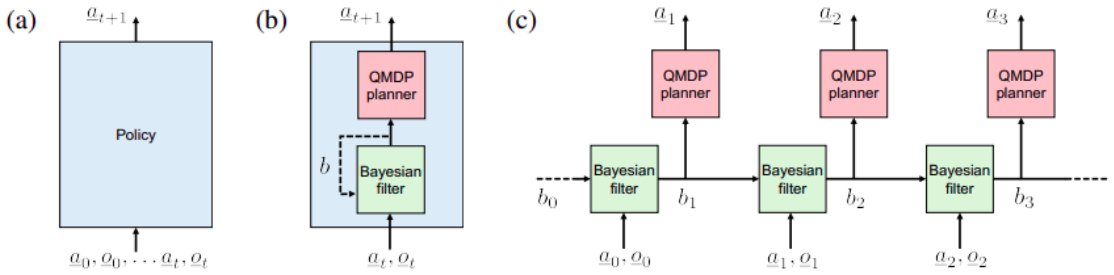


Figure 8. Architecture of QMDP-net. It is a recurrent network (you can see it unfolded in time on picture c), that per each time step receives action and observation and outputs the new action to execute. The architecture consist of Baysian filter and the QMDP algorithm. Source [8]

by expert during the training. Cross entropy between predicted and demonstrated action sequences was used as a loss function. The RMSProp optimizer was used during training.

When testing different parameter values were used (to check the perfomance for previously unseen scenarios).

4.2.1 Filter module

The filter module Figure 9 implements a Bayesian filter. It maps from a belief, action, and observation to a next belief, $b_{t+1} = f(b_t|a_t, o_t)$. The belief is updated in two steps. The first accounts for actions, the second for observations:

$$b'_t(s) = \sum_{s' \in S} T(s, a_t, s') b_t(s') \tag{4}$$

$$b_{t+1}(s) = \eta Z(s, o_t) b'_t(s) \tag{5}$$

Equation 4 can be implemented as a convolutional layer with $|A|$ convolutional filters. The output of that convolutional layer, $b'_t(s,a)$, is a N×N×$|A|$ tensor that will consist of the updated beliefs after taking each of the actions.

Equation 5 consists of the observations from an observation model $Z(s,o)$. The $Z(s,o)$ is a tensor that represents the probability of receiving observation o in state s (the size is: N×N×$|O|$).

As a result we can calculate $b_{t+1}(s)$, by multiplying $b'_t(s)$ and $Z(s)$ element-wise, and normalizing over states.

4.2.2 Planner module

After estimating the belief state, we need to select the best possible action in that state.

The QMDP planner Figure 9 performs value iteration at its core. Q values are computed by iteratively applying Bellman updates according to Equation 6 and Equation 7,

$$Q_{k+1}(s,a) = R(s,a) + \gamma \sum_{s' \in S} T(s,a,s') V_k(s') \tag{6}$$

,

$$V_k(s) = max_a Q_k(s,a) \tag{7}$$

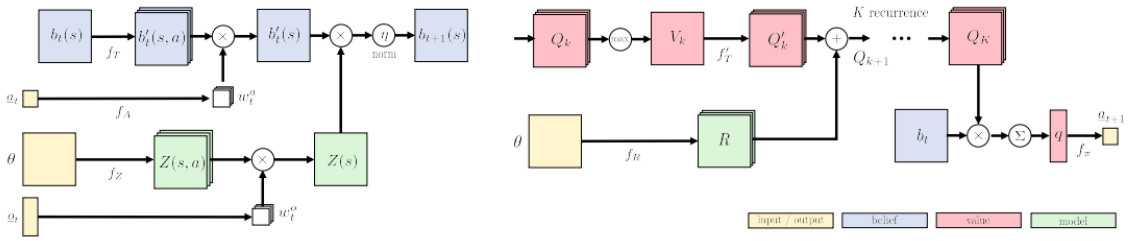
.

We select actions by weighting the Q values with the beliefs (probability distribution over the states).

Analogically to the filter module, we can implement all the calculations by using appropriate neural network layers. We will use convolutional layers and addition operation to implement the Equation 6 and max-pooling layers to extract the max (Equation 7).

We implement K iterations of Bellman updates by stacking the layers representing Equation 6 and Equation 7 K times (as you can see on the Figure 9). Each iteration improves the approximation of Q values for each state-action pair. The higher the K the better we can approximate these values, however, it will also increase the computation time. We weight the Q values by the belief to obtain action values,

$$q(a) = \sum_{s \in S} Q_k(s,a) b_t(s) \tag{8}$$



(a) Bayesian filter module (b) QMDP planner module

Figure 9. QMDP-net filter and planner module architecture. Source [8]

At the end, according to q(a) we select the next action a_{t+1} (with the highest q value).

This architecture and training setting was able to outperform the QMDP algorithm implemented in classical, not a neural network form. It was also able to generalize well over new, unseen before states, which proves that by parametrizing the task settings we can generalize better.

4.3 Conclusion

This approach showed that by embedding the know formulas, like the Bayesian filter, as neural network operations and training them in the end-to-end way we can improve the performance of the model. It also shown that by parametrizing the task and embedding these parameters inside the model, we are able to learn policy that will perform well for the tasks unseen before (the policy will be able to generalize well over new possible scenarios).

This approach, unfortunately, didn't break the curse of dimensionality and as a result for larger POMDP, the QMDP net could learn to discretize belief states too much.

5. Results and Discussion

We could see that the POMDP is much more difficult to solve than the MDP. There are a lot of very different approaches to solving problems in the POMDP framework. Sometimes we can also very easily transform the problem from being a POMDP to the MDP form (section 2). We could see that the curse of dimensionality and histories are the two main reasons why solving POMDP problems is so difficult, especially for the large POMDP (paragraph and paragraph). However, it is possible to break even both of them (section 3) and perform well for the large POMDP problems (where the complexity doesn't depend on the size of the belief space. In the last described method (section 4) we could also see the approach that enables to learn a policy that will be able to generalize well over even tasks that the agent hasn't seen before.

To sum up, the methods to solve the POMDP are necessary as these POMDP scenarios are very common in real-life. All the methods described introduce some improvements, advantages, and disadvantages. The final technique, we will implement, should be chosen carefully and appropriately to the problem we want to solve.

References

[1] Hanna Kurniawati. Partially observable markov decision processes (pomdps) and robotics. 2021.

[2] G.Mallikarjuna Rao and Ch Satyanarayana. Visual object target tracking using particle filter: A survey. *International Journal of Image, Graphics and Signal Processing*, 5:57–71, 05 2013.

[3] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016.

[4] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nature*, 550:354–, October 2017.

[5] Matthew Hausknecht and Peter Stone. Deep recurrent q-learning for partially observable mdps, 2015.

[6] David Silver and Joel Veness. Monte-carlo planning in large pomdps. In J. Lafferty, C. Williams, J. Shawe-Taylor, R. Zemel, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 23. Curran Associates, Inc., 2010.

[7] Michael L. Littman, Anthony R. Cassandra, and Leslie Pack Kaelbling. Learning policies for partially observable environments: Scaling up. In Armand Prieditis and Stuart Russell, editors, *Machine Learning Proceedings 1995*, pages 362–370. Morgan Kaufmann, San Francisco (CA), 1995.

[8] Peter Karkus, David Hsu, and Wee Lee. Qmdp-net: Deep learning for planning under partial observability. 12 2017.