Machine Intelligence Assignment 2

(Chapters Summary)

Team 1

Mohamed Shawky Zaky, Sec : **2**, BN : **15**
Remonda Talaat Eskarous, Sec : **1**, BN : **19**
Mohamed Ahmed Mohamed Ahmed, Sec : **2**, BN : **10**
Mohamed Ramzy Helmy, Sec : **2**, BN : **13**

Chapter 13

Chapter 14

Chapter (14)

The previous chapter introduced the probability theory and independence relationships' role in probabilistic representation. This chapter introduces Bayesian Networks' syntax and semantics and their use to capture uncertain knowledge. Also, practical and approximate algorithms for probabilistic inference are introduced for real-world applications.

Representing knowledge in an uncertain domain

A Bayesian Network is a data structure representing the relationship between variables that consists of an acyclic directed graph of nodes and edges where nodes represent random variables, and directed edges represent influence between variables. For example, if variable X has a directed edge to variable Y, then X influences Y, and X is called the parent of Y. Also, each node X has a conditional probability distribution **p(X | parents(X))** representing the influence of the node's parents on it. That probability distribution is recorded in a Conditional Probability Table (CPT) for each variable.

The Semantics of Bayesian Networks

Bayesian's Network, that has n nodes {X1, X2, ….. Xn}, represents a joint probability distribution between the whole random variables of all nodes.

$$p(x1,\ x2,\ \dots\ xn)\ =\ \prod_{i=0}^{n} p(xi\,|\,parents(xi))$$

So to construct a Bayesian Network:

1) Start with determining the set of random variables of interest.
2) For each variable, Choose all the other random variables that the current variable depends on, add edges from them to the existing variable and write p(xi | parents(xi)in the conditional probability table.

So, to make the representation of Bayesian networks tractable, Bayesian Networks constrain each node to be influenced by most k parents, so $n2^{k}$numbers represent a Bayesian network with n nodes and a boolean relationship. In contrast, billions of numbers can represent the joint probability distribution. In some applications, each node is affected by all nodes in the network so that the Bayesian network will represent the joint probability distribution. Some independence relationships that don't affect the accuracy can be removed to eliminate this complexity.
Each node is conditionally independent of its non-descendants given its parents. Also, each node is conditionally independent of all the network's nodes given its Markov blanket, which consists of its parents, children, and children's parents.

Efficient Representation of Conditional Distributions

Representing the Bayesian network using the constraint of limiting each node to be affected by only k parents helps make Bayesian networks tractable.

But the worst-case complexity of filling the CPT for a single node is *O(2k)* so, another efficient representation for the Conditional distribution is provided.

Deterministic nodes which are only affected by their parents are efficient to represent in the CPT as their value depends only on the value of their parents. Non-deterministic nodes influenced by variables other than their parents can be represented by a causal distribution using noisy-or operation that consists of the logical or in addition to some uncertainty. Filling the CPT table for non-deterministic nodes using noisy-or can be done in *O(k)* as follows:

$$p(Xi \mid parents(Xi) \;=\; 1 \;-\; \prod_{\{j:\,Xj\,=\,True\}} p(\neg Xi \mid Xj,\, \neg(parents(Xi) \;-\; Xj)$$

Continuous variables in a Bayesian network can be easily represented by discretizing them, but this causes an accuracy loss. Other methods to represent it are using a parametric representation, such as representing it with a gaussian distribution or a non-parametric representation explored in chapter 18.

Exact Inference in Bayesian Networks

Inference in Bayesian Networks determines the probability of a query variable X given a particular event e.

As shown in chapter 13, calculating *p(x | e)* can be done by Bayes rule using enumeration.

Unnecessary extra computation is done by calculating a particular conditional probability more than once. So a variable elimination algorithm is introduced, which calculates each probability exactly once and reuses it for later computations. Also, node clustering can help reduce complexity by joining nodes together to make a simpler graph.

The complexity of exact inference in Bayesian Networks that have polytree structure is linear, and It is exponential in Multiply connected networks. Inference in Bayesian Networks in an NP-hard problem.

Approximate Inference in Bayesian Networks

Given the complexity of exact inference methods, Other fast approximate inference algorithms are introduced, which depend on sampling the probabilities of the network's nodes from a known probability distribution using direct and rejection sampling methods with likelihood weighting, which avoids inconsistent samples. Also, the Monte Carlo methods sample next states by making random changes to the current state. Using Approximate Inference faster in real-time applications, but it suffers some accuracy loss. Exact inference methods can be used when a high level of accuracy is required.

Relational and First-order Probability Models

Combining First-Order models and relational probability models with probability theory can help in solving more problems due to the power of representation that they provide that will help in defining well-defined probability distributions and their ability to deal with infinite spaces with first-order possible worlds.

Other Approaches to Uncertain Reasoning

Methods other than Probability theory were used in probabilistic reasoning, such as rule-based systems, expert systems, and fuzzy logic, as human judgment reasoning is qualitative, not quantitative reasoning.

Chapter 15

Chapter (15)

In previous chapters, we discussed probabilistic reasoning techniques in the context of static worlds. In this chapter, we move on to temporal (dynamic) worlds, in which the state of the world changes through time.

Representation

We discuss the representation of probabilistic temporal processes. Here, the world is viewed as a series of snapshots (time slices), each containing a set of random variables. We denote the set of unobservable state variables by **X**t and the set of observable evidence variables by **E**t at time *t*.

The transition model specifies the probability distribution over the latest state variables, given the previous values, $\mathbf{P}(\mathbf{X}_t \mid \mathbf{X}_{0:t-1})$. However, to overcome size constraints, we use the **Markov assumption**, which states that the current state depends on only a *finite fixed number* of previous states. The simplest assumption is **the first-order Markov process**, which is given by:

$$\mathbf{P}(\mathbf{X}_t \mid \mathbf{X}_{0:t-1}) = \mathbf{P}(\mathbf{X}_t \mid \mathbf{X}_{t-1})$$

The sensor model (sometimes called the observation model) is given by $\mathbf{P}(\mathbf{E}_t \mid \mathbf{X}_t)$.

Inference

The basic inference tasks are :

1) **Filtering :** This is the task of computing the belief state (the posterior distribution over the most recent state) given all evidence to date. Filtering is also called state estimation. It is given by :

$$\mathbf{P}(\mathbf{X}_{t+1} \mid e_{1:t+1}) = \alpha\, \mathbf{P}(e_{t+1} \mid \mathbf{X}_{t+1}) \sum_{Xt} \mathbf{P}(\mathbf{X}_{t+1} \mid x_t)\mathbf{P}(x_t \mid e_{1:t})$$

2) **Prediction :** This is the task of computing the posterior distribution over the future state, given all evidence to date. It is useful for evaluating possible courses of action based on their expected outcomes.
3) **Smoothing :** This is the task of computing the posterior distribution over a past state, given all evidence up to the present. It provides better estimates of the state than what was available at the time, due to more evidence.
4) **Most likely explanation :** This is the task of finding the sequence of states that is most likely to have generated a given sequence of observations. It is useful for many applications, including speech recognition.

Hidden Markov Models

A Hidden Markov Model **(HMM)** is a probabilistic temporal model in which a single discrete random variable describes the state. The possible values of this variable are the possible states of the world. We can still fit a model with two or more state variables into the HMM framework by combining the variables into a single "megavariable" whose values are all possible tuples of values of the individual state variables.

HMM uses simplified matrix representation for both forward and backward messages in **T** (transition matrix) and $\mathbf{O}_t$ (observations diagonal matrix at time *t*).

Kalman Filters

A Kalman Filter is a way of handling the continuous state variables, in which the next state $\mathbf{X}_{t+1}$ must be a linear function of the current state $\mathbf{X}_t$ plus some Gaussian noise. It performs two-step filtering to incorporate both transition and sensor models given each of the following :

1) The current distribution $\mathbf{P}(\mathbf{X}_t \mid e_{1:t})$ is Gaussian and the transition model $\mathbf{P}(X_{t+1} \mid x_t)$ is linear Gaussian, then :

$$\mathbf{P}(\mathbf{X}_{t+1} \mid e_{1:t}) = \int_{xt} \mathbf{P}(\mathbf{X}_{t+1} \mid x_t)\mathbf{P}(x_t \mid e_{1:t})\, dx_t$$

2) The prediction $\mathbf{P}(\mathbf{X}_{t+1} \mid e_{1:t})$ is Gaussian and the sensor model $\mathbf{P}(e_{t+1} \mid \mathbf{X}_{t+1})$ is linear Gaussian, then :

$$\mathbf{P}(\mathbf{X}_{t+1} \mid e_{1:t+1}) = \alpha\, \mathbf{P}(e_{t+1} \mid \mathbf{X}_{t+1})\mathbf{P}(\mathbf{X}_{t+1} \mid e_{1:t})$$

The Extended Kalman Filter (**EKF**) attempts to overcome the nonlinearities in the system being modeled. Also, **Switching Kalman Filter** is used to run multiple Kalman Filters in parallel, each using a different model of the system. A weighted sum of predictions is used, where the weight depends on how well each filter fits the current data.

Dynamic Bayesian Networks

A Dynamic Bayesian Network (**DBN**) is used to handle any number of state variables $\mathbf{X}_t$ and evidence variables $\mathbf{E}_t$. For simplicity, we assume that the variables and their links are exactly replicated from slice to slice and that the DBN represents a first-order Markov process. HMM can be represented by DBN with a single state variable and a single evidence variable. Also, DBN expands the Kalman filter to model arbitrary distributions. Exact inference in DBN is done through unrolling the time slices, while approximate inference can be done through Markov Chain Monte Carlo (**MCMC**).

Particle filtering is designed to focus on the set of samples on the high-probability regions of the state space. It starts with a population of *N* initial-state samples, created by sampling from the prior distribution $\mathbf{P}(\mathbf{X}_0)$. Then, it performs the update cycle as follows :

1) Each sample is propagated forward based on the transition model $\mathbf{P}(\mathbf{X}_{t+1} \mid x_t)$.
2) Each sample is weighted by the likelihood it assigns to the new evidence $\mathbf{P}(e_{t+1} \mid x_{t+1})$.
3) The population is resampled to generate a new population of *N* samples.

Keeping Track of Many Objects

When trying to keep track of many objects, uncertainty arises from which observations belong to which objects (the **data association** problem). The data association problem is associating observation data with the objects that generated them.

Chapter 21

Chapter (21)

In this chapter, we examine how an agent can learn from rewards and punishments.

Introduction

Reinforcement learning uses some kind of feedback from the environment, namely **reward** or **reinforcement,** in order to learn and evaluate actions. The framework for agents regards the reward as part of the input percept, but the agent must be "hardwired" to recognize that part as a reward rather than as just another sensory input. These rewards are used to define optimal policies in a Markov Decision Process (**MDP**). An **optimal policy** is a policy that maximizes the expected total reward. The task of **reinforcement learning** is to use observed rewards to learn an optimal (or nearly optimal) policy for the environment. Previously, we assumed the agent to have a complete model of the environment and knows the reward function, however, here, we assume no prior knowledge of either.

We will consider three of the agent designs :

- A **utility-based agent** learns a utility function on states and takes actions that maximize the expected utility.
- A **Q-learning agent** learns an action-utility function giving the expected utility of taking an action in a state.
- A **reflex agent** learns a policy that maps directly from states to actions.

Passive Reinforcement Learning

In **passive learning**, the agent's policy $\boldsymbol{\pi}$ is fixed: in state **s**, it always executes the action $\boldsymbol{\pi}$(**s**). Its goal is simply to learn the utility function $\mathbf{U}^{\pi}$(**s**). Passive learning task is similar to the policy evaluation task, however the main difference is that the passive learning agent does not know the transition model nor does it know the reward function (which specifies the reward for each state).

The main is that the agent executes a set of trials in the environment using its policy $\boldsymbol{\pi}$**.** The utility is defined as the expected sum of rewards obtained if policy $\boldsymbol{\pi}$ is followed. We can follow one of the following methods :

1) **Direct utility estimation :** The main idea is that each trial provides a sample of the utility for each visited state, which is the expected total reward from that state onward. This, basically, reduces reinforcement learning into a standard inductive learning problem. Unfortunately, it doesn't consider the fact that the utilities of the states are not independent.
2) **Adaptive dynamic programming :** It takes advantage of the constraints among the utilities of the states by learning a transition model that connects them and solving the corresponding **MDP** using dynamic programming. Thus, we have a supervised learning task, whose input is a state-action pair and the output is the resulting state. It's based on Bellman equation :

$$\mathbf{U}_{\pi}(s) = \mathbf{R}(s) + \gamma \textstyle\sum_{s'} \mathbf{P}(s' \mid s, \pi(s))\, \mathbf{U}^{\pi}(s')$$

3) **Temporal-difference learning :** It's named temporal difference, because its update rules uses the difference in utilities between successive states, as follows :

$$\mathbf{U}^{\pi}(s) \leftarrow \mathbf{U}^{\pi}(s) + \alpha(\mathbf{R}(s) + \gamma\, \mathbf{U}^{\pi}(s') - \mathbf{U}^{\pi}(s))$$

Where α is the learning rate parameter. Both **ADP** and **TD** try to make local adjustments to the utility estimates.

Active Reinforcement Learning

Unlike passive learning agents, an active agent doesn't have a fixed policy and should decide what actions to take. Generally, the agent must, first, learn a complete model with outcome probabilities of all actions and then use it to learn the *optimal policy.* A **greedy agent** always tries to stick with the current optimal actions, which very seldom converges to an actual optimal policy. Consequently, the agent must make a tradeoff between **exploitation** to maximize its reward and **exploration** to maximize its long-term well-being. Also, we introduce *two* important learning methods :

1) **Q-Learning :** learns an action-utility representation instead of learning utilities. This representation is formulated by **Q-function** *Q(s, a)*, representing the value of doing action *a* in state *s*. Since the **Q-function** doesn't need a model of the environment *P(s' | s, a)*, Q-learning is called a **model-free** method. The following equation represents the update equation for **TD Q-Learning** :

$$Q(s, a) \leftarrow Q(s, a) + \alpha(R(s) + \gamma \max_{a'} Q(s', a') - Q(s, a))$$

2) **SARSA (State-Action-Reward-State-Action) :** very similar to that of **Q-Learning**, however the update rule is applied at the end of each *s, a, r, s', a'* quintuplet. The main difference is that **Q-Learning** backs up the best Q-Value from the state reached in the observed transition (**off-policy** learning algorithm), however **SARSA** waits until an action is actually taken and backs up the Q-Value for this action (**on-policy** learning algorithm).

Generalization in Reinforcement Learning

Previously, we assumed a **tabular** representation with one output value for each input tuple. Such an approach works well for small state spaces, however it becomes *computationally expensive* for large state spaces. One way to handle this problem is to use **function approximation**, which involves **parametrization** of **Q-function** into a set of parameters less than that of a lookup table. A function approximator isn't just important for large state spaces, it also allows the agent to **generalize** from visited states to states it has not visited. The parameters of the function approximator is updated using **Widrow-Hoff** update rule (similar to *neural network learning*) :

$$\theta_i \leftarrow \theta_i - \alpha\,(\partial E_j / \partial \theta_i)$$

Policy Search

The idea is simply to keep twiddling the policy as long as its performance improves, then stop. The policy is represented as follows :

$$\pi(s) = \max_a \hat{Q}_\theta(s, a)$$

We can follow the policy gradient, if the policy is differentiable. Otherwise, we can follow the empirical gradient by hill climbing, i.e. evaluating the change in policy value for small increments in each parameter.

Applications of Reinforcement Learning

Reinforcement Learning has a lot of large-scale applications. We mention two of them, which are :

1) **Game playing :** TD-Gammon.
2) **Robot Control :** cart-pole.