Value Function Approximation

Michail G. Lagoudakis
Technical University of Crete, Chania, Greece

Abstract
The goal in sequential decision making under uncertainty is to find good or optimal policies for selecting actions in stochastic environments in order to achieve a long-term goal; such problems are typically modeled as Markov decision processes (MDPs). A key concept in MDPs is the *value function*, a real-valued function that summarizes the long-term goodness of a decision into a single number and allows the formulation of optimal decision making as an optimization problem. An exact representation of value functions in large real-world problems is infeasible; therefore, a large body of research has been devoted to *value-function approximation* methods, which sacrifice some representation accuracy for the sake of scalability. These approaches have delivered effective approaches to deriving good policies in hard decision problems and laid the foundation for efficient reinforcement learning algorithms, which learn good policies in unknown stochastic environments through interaction.

Synonyms

Approximate dynamic programming; Cost-to-go function approximation; Neuro-dynamic programming

Definition

Value Function Approximation is a collection of function approximation representations, techniques, and methods aiming at providing a scalable and effective approximation to an exact value function (a real-valued function indicating the long-term goodness of making decisions at any state within a sequential decision problem).

Motivation and Background

Markov Decision Processes

A *Markov decision process* (MDP) is a 6-tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma, \mathcal{D})$, where $\mathcal{S}$ is the state space of the process, $\mathcal{A}$ is a finite set of actions, $\mathcal{P}$ is a Markovian transition model ($\mathcal{P}(s'|s,a)$ denotes the probability of a transition to state s' when taking action a in state s), $\mathcal{R}$ is a reward function ($\mathcal{R}(s,a)$ is the reward for taking action a in state s), $\gamma \in (0,1]$ is the discount factor for future rewards (a reward received after t steps is weighted by γ^t), and $\mathcal{D}$ is the initial state distribution (Puterman 1994). MDPs are discrete-time processes. The process begins at time $t = 0$

C. Sammut, G.I. Webb (eds.), *Encyclopedia of Machine Learning and Data Mining*,
DOI 10.1007/978-1-4899-7687-1

in some state $s_0 \in \mathcal{S}$ drawn from $\mathcal{D}$. At each time step t, the decision maker observes the current state of the process $s_t \in \mathcal{S}$ and chooses an action $a_t \in \mathcal{A}$. The next state of the process s_{t+1} is drawn stochastically according to the transition model $\mathcal{P}(s_{t+1}|s_t, a_t)$, and the reward r_t at that time step is determined by the reward function $\mathcal{R}(s_t, a_t)$. The horizon h is the temporal extent of each run of the process and is typically infinite. A complete run of the process over its horizon is called an *episode* and consists of a long sequence of states, actions, and rewards:

$$s_0 \xrightarrow[r_0]{a_0} s_1 \xrightarrow[r_1]{a_1} s_2 \xrightarrow[r_2]{a_2} s_3 \xrightarrow[r_3]{a_3} s_4 \quad \ldots \quad s_{h-1} \xrightarrow[r_{h-1}]{a_{h-1}} s_h .$$

The quantity of interest is the *expected total discounted reward* from any state s:

$$E\Big(r_0 + \gamma r_1 + \gamma^2 r_2 + \gamma^3 r_3 + \cdots + \gamma^h r_h \;\Big|\; s_0 = s\Big) = E\left(\sum_{t=0}^{h} \gamma^t r_t \;\middle|\; s_0 = s\right),$$

where the expectation is taken with respect to all possible trajectories of the process in the state space under the decisions made and the transition model, assuming that the process is initialized in state s. The goal of the decision maker is to make decisions so that the expected total discounted reward, when s is drawn from $\mathcal{D}$, is optimized. (The optimization objective could be maximization or minimization depending on the problem. Here, we adopt a reward maximization viewpoint, but there are analogous definitions for cost minimization. There are also other popular optimality measures, such as maximization/minimization of the average reward/cost per step.)

Policies

A *policy* dictates how the decision maker chooses actions in each state. A *stationary, deterministic policy* π is a mapping $\pi : \mathcal{S} \mapsto \mathcal{A}$ from states to actions; $\pi(s)$ denotes the action the agent takes in state s. In this case, there is a single action choice for each state, and this choice does not change over time. In contrast, a *stationary, stochastic policy* π is a mapping $\pi : \mathcal{S} \mapsto \Omega(\mathcal{A})$, where $\Omega(\mathcal{A})$ is the set of all probability distributions over $\mathcal{A}$. Stochastic policies are also called *soft*, for they do not commit to a single action per state; $\pi(a|s)$ stands for the probability of choosing action a in state s under policy π. Each policy π (deterministic or stochastic) is characterized by the expected total discounted reward it accumulates during an episode. An *optimal policy* π^* for an MDP is a policy that maximizes the expected total discounted reward from any state $s \in \mathcal{S}$. It is well known that for every MDP, there exists at least one, not necessarily unique, optimal policy, which is stationary and deterministic.

Value Functions

The quality of any policy π can be quantified formally through a value function, which measures the expected return of the policy under different process initializations. For any MDP and any policy π, the *state value function* V assigns a numeric value to each state. The value $V^\pi(s)$ of a state s under a policy π is the expected return, when the process starts in state s and the decision maker follows policy π (all decisions at all steps are made according to π):

$$V^\pi(s) = E_{a_t \sim \pi \,;\, s_t \sim \mathcal{P} \,;\, r_t \sim \mathcal{R}} \left(\sum_{t=0}^{\infty} \gamma^t r_t \;\middle|\; s_0 = s\right).$$

Similarly, the *state-action value function* Q assigns a numeric value to each pair (s, a) of states and actions. The value $Q^\pi(s, a)$ of taking action a in state s under a policy π is the expected return when the process starts in state s, and the decision maker takes action a for the first step and follows policy π thereafter:

$$Q^{\pi}(s,a) = E_{a_t\sim\pi\,;\,s_t\sim\mathcal{P}\,;\,r_t\sim\mathcal{R}}\left(\sum_{t=0}^{\infty}\gamma^t r_t \,\middle|\, s_0=s,\, a_0=a\right).$$

The state and the state-action value functions for a deterministic policy π are related as follows:

$$V^{\pi}(s) = Q^{\pi}\big(s,\pi(s)\big).$$

For a stochastic policy π this relationship needs to take into account the probability distribution over actions:

$$V^{\pi}(s) = \sum_{a\in\mathcal{A}} \pi(a|s)Q^{\pi}(s,a).$$

The state-action value function of a policy π (either deterministic or stochastic) can also be expressed in terms of the state value function:

$$Q^{\pi}(s,a) = \mathcal{R}(s,a) + \gamma\sum_{s'\in\mathcal{S}}\mathcal{P}(s'|s,a)V^{\pi}(s').$$

The *optimal value functions* $V^* = V^{\pi^*}$ and $Q^* = Q^{\pi^*}$ are the state and the state-action value functions of any optimal policy π^*. Even if there are several distinct optimal policies, they all share the same unique optimal value functions.

Bellman Equations

Given the full MDP model, the state or the state-action value function of any given policy can be computed by solving a linear system formed using the linear Bellman equations. In general, the linear *Bellman equation* relates the value of the function at any point to the values of the function at several – in fact, all – other points. This is achieved by separating the first step of an episode from the rest and using the definition of the value function recursively in the next step. In particular, for any deterministic policy π, the linear Bellman equation for the state value function is

$$V^{\pi}(s) = \mathcal{R}(s,\pi(s)) + \gamma\sum_{s'\in\mathcal{S}}\mathcal{P}(s'|s,\pi(s))V^{\pi}(s'),$$

whereas for a stochastic policy π, it becomes

$$V^{\pi}(s) = \sum_{a\in\mathcal{A}}\pi(a|s)\Big(\mathcal{R}(s,a) + \gamma\sum_{s'\in\mathcal{S}}\mathcal{P}(s'|s,a)V^{\pi}(s')\Big).$$

The exact V^{π} values for all states can be found by solving the $(|\mathcal{S}| \times |\mathcal{S}|)$ linear system that results from writing down the linear Bellman equation for all states.

Similarly, the linear Bellman equation for the state-action value function given any deterministic policy π is

$$Q^{\pi}(s,a) = \mathcal{R}(s,a) + \gamma\sum_{s'\in\mathcal{S}}\mathcal{P}(s'|s,a)Q^{\pi}\big(s',\pi(s')\big),$$

whereas for a stochastic policy π, it becomes

$$Q^{\pi}(s,a) = \mathcal{R}(s,a) + \gamma\sum_{s'\in\mathcal{S}}\mathcal{P}(s'|s,a) \times \sum_{a'\in\mathcal{A}}\pi(a'|s')Q^{\pi}(s',a').$$

The exact Q^{π} values for all state-action pairs can be found by solving the $(|\mathcal{S}||\mathcal{A}| \times |\mathcal{S}||\mathcal{A}|)$ linear system that results from writing down the linear Bellman equation for all state-action pairs.

The unique optimal state or state-action value function can be computed even for an unknown optimal policy π^* using the nonlinear *Bellman optimality equation*, which relates values of the function at different points while exploiting the fact that there exists a deterministic optimal policy that achieves the maximum value at each point. In particular, the nonlinear Bellman optimality equation for the state value function is

$$V^*(s) = \max_{a\in\mathcal{A}}\Big\{\mathcal{R}(s,a) + \gamma\sum_{s'\in\mathcal{S}}\mathcal{P}(s'|s,a)V^*(s')\Big\},$$

whereas for the state-action value function is

$$Q^*(s,a) = \mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s,a) \max_{a' \in \mathcal{A}} \left\{ Q^*(s',a') \right\}.$$

The functions V^* and Q^* can be approximated arbitrarily closely by the iterative application of the operator formed by the right-hand side of the equations above (Bellman optimality operator). This iteration is a contraction with rate γ, so starting with any arbitrary initialization, it eventually converges to V^* or Q^*.

Significance of Value Functions

Value functions play a critical role in sequential decision making because they address two core problems: policy evaluation and policy improvement. Policy evaluation refers to the problem of quantifying the quality of any given policy π in a given MDP. Apparently, computing the value function V^π or Q^π using the Bellman equations provides the solution to this problem. Policy improvement, on the other hand, refers to the problem of deriving an improved policy π' given any base policy π, so that π' is at least as good as π and possibly better. The knowledge of V^π or Q^π allows for the derivation of an improved deterministic policy π' through a simple one-step look-ahead maximization procedure:

$$\pi'(s) = \arg\max_{a \in \mathcal{A}} \left\{ \mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s'|s,a) V^\pi(s') \right\}$$

$$\pi'(s) = \arg\max_{a \in \mathcal{A}} \left\{ Q^\pi(s,a) \right\}.$$

Note that this maximization does not need the MDP model when using the state-action value function. Once policy evaluation and policy improvement have been addressed, the derivation of an optimal policy for any MDP is straightforward. One can alternate between policy evaluation and policy improvement producing a sequence of improving policies until convergence to an optimal policy; this algorithm is known as policy iteration. Alternatively, one can iteratively compute an optimal value function V^* or Q^* and extract an optimal policy through a trivial step of policy improvement on top of V^* or Q^*; this algorithm is known as value iteration. In either case, value functions provide the means to the end.

The problem of deriving an optimal policy using the full MDP model is known as planning. Nevertheless, in many real-world sequential decision domains, the model is unknown. The problem of optimal decision making in an unknown stochastic environment is known as reinforcement learning, because the decision maker relies on the feedback received through interaction with the environment to reinforce or discourage past decisions. More specifically, the learner interacts with an unknown MDP and typically observes the state of the process and the immediate reward at every step; however, $\mathcal{P}$ and $\mathcal{R}$ are not accessible. At each step of interaction, the learner observes the current state s, chooses an action a, and observes the resulting next state s' and the reward received r, thus learning is based on (s, a, r, s') samples. The core problems in reinforcement learning are known as prediction and control. Prediction refers to the problem of learning the value function of a given policy π in an unknown MDP through interaction. Well-known algorithms for the prediction problem are Monte Carlo estimation and temporal difference (TD) learning. Control, on the other hand, refers to the problem of gradually learning a good or even optimal policy in an unknown MDP through interaction. Well-known algorithms for the control problem are SARSA and Q-learning. Again, value functions play a critical role in reinforcement learning; they are absolutely necessary for the prediction problem, and the majority of control approaches are value-function based.

Structure of Learning System

Value-Function Approximation

Most algorithms for planning or learning in MDPs rely on computing or learning a value function. However, if the state space of the process is fairly large, the exact (tabular) representation of a value function becomes problematic. Not only does memory space become insufficient very quickly, but also computing or learning accurately all the distinct entries of the function requires a tremendous amount of computation and data. This is known as the *curse of dimensionality*: the exponential growth of the state or action space as a function of the dimensionality of the state or action. The urgent need for solutions to large real-world sequential decision problems has drawn attention to approximate methods. These methods use function approximation techniques for approximating value functions; therefore, they sacrifice some representational accuracy in order to make the representation manageable in practice. Sacrificing accuracy in the representation of the value function is acceptable, since the ultimate goal is to find a good policy and not necessarily an accurate value function. As a result, value-function approximation methods cannot guarantee optimal solutions, but only good solutions. This is not to say that they are doomed to always finding suboptimal solutions; if an optimal solution lies within the space spanned by the value-function approximation scheme, it is possible that an optimal solution will be discovered.

Let $\hat{V}^{\pi}(s;w)$ be an approximation to the state value function $V^{\pi}(s)$ represented by a parametric approximation architecture with free parameters w. The key idea of value- function approximation is that the parameters w can be adjusted appropriately so that the approximate values are "close enough" to the original values,

$$\hat{V}^{\pi}(s;w) \approx V^{\pi}(s),$$

and, therefore, $\hat{V}^{\pi}$ can be used in place of the exact value function V^{π}. Similarly, let $\hat{Q}^{\pi}(s,a;w)$ be an approximation to the state-action value function $Q^{\pi}(s,a)$. Again, the goal is to adjust the parameters w so that

$$\hat{Q}^{\pi}(s,a;w) \approx Q^{\pi}(s,a),$$

and, therefore, $\hat{Q}^{\pi}$ can be used in place of the exact value function Q^{π}. Approximations $\hat{V}^{*}$ and $\hat{Q}^{*}$ of the optimal value functions V^{*} and Q^{*} are defined similarly. The characterization "close enough" ($\approx$) accepts a variety of interpretations in this context, and it does not necessarily refer to the minimization of some norm. Value-function approximation should be regarded as a *functional* approximation rather than as a pure *numerical* approximation, where "functional" refers to the ability of the approximation to play closely the functional role of the original value function within a decision making algorithm.

The benefits of value-function approximation are obvious. The storage requirements are much smaller compared to the tabular case, since only the parameters w need to be stored along with a compact description of the functional form of the architecture. In general, for most approximation architectures, the storage needs are independent of the size of the state space and/or the size of the action space. Furthermore, for most approximation architectures there is no restriction on the state space to be a finite set; it could be an infinite, or even a continuous, space. This flexibility nevertheless reveals the need for good generalization abilities on behalf of the architecture, since the approximate value function will have to provide good values over the entire state/state-action space, using data only from a limited subset of the space.

The main difficulty associated with value-function approximation, beyond the loss in accuracy, is the choice of the *projection method*, which is the method of finding appropriate parameters that maximize the accuracy of the approximation according to certain criteria and with respect to the target function. Typically, for ordinary function approximation, this is accomplished using a training set of examples of the form $\{s, V^{\pi}(s)\}$, $\{s, V^{*}(s)\}$, $\{(s,a), Q^{\pi}(s,a)\}$, or $\{(s,a), Q^{*}(s,a)\}$ that provide the true value of the target function at certain sample points s or (s,a) (supervised learning). Unfortunately, in the context of sequential decision making, the target value function is completely unknown. Had it been possible to compute it easily, value-function approximation would have been unnecessary. In fact, it is not possible to analytically compute the true value of the target value function even at certain isolated sample points due to interdependencies between the values at all points. The implication of this difficulty is that evaluation and projection to the approximation architecture must be blended together. This is usually achieved by trying to find values for the free parameters so that the approximate function retains some properties of the original exact value function. Another implication of using approximation for value functions is that all convergence properties of exact planning or learning algorithms are compromised. Therefore, significant attention must be paid to the choice of the approximation architecture and the evaluation and projection method to minimize the chances for divergence or oscillations.

Approximation Architectures

There are a variety of architectures available for value-function approximation: perceptrons, neural networks, splines, polynomials, radial basis functions, support vector machines, decision trees, CMACs, wavelets, etc. These architectures have diverse representational power and generalization abilities, and the most appropriate choice will heavily depend on the properties of the decision making problem at hand. The projection methods associated with these approximation architectures are typically designed for a supervised learning setting. For successful use in the context of decision making, combined evaluation and projection methods are necessary.

A broad categorization of approximation architectures distinguishes between nonlinear and linear architectures. The characterization "nonlinear" or "linear" refers to the way the free parameters enter into the architecture and not to the approximation ability of the architecture. Nonlinear architectures are usually more expressive than the linear ones, due to the complex interactions among their free parameters; however, tuning their parameters is a much more elaborate task compared to tuning the parameters of their linear counterparts. Linear architectures are perhaps the most popular choice for value-function approximation; interestingly, most theoretical results on convergence properties in the context of value-function approximation are restricted to linear architectures.

A *linear approximation architecture* approximates a function $V^\pi(s)$ or $Q^\pi(s,a)$ as a linear weighted combination of k *basis functions* (also called *features*):

$$\hat{V}^\pi(s;w) = \sum_{j=1}^{k} \phi_j(s) w_j = \phi(s)^\top w$$

$$\hat{Q}^\pi(s,a;w) = \sum_{j=1}^{k} \phi_j(s,a) w_j = \phi(s,a)^\top w.$$

The free *parameters* of the architecture are the coefficients w_j of the combination (also called *weights*). The basis functions ϕ_j are fixed, but arbitrary and, in general, nonlinear functions of s or (s,a). It is required that the basis functions ϕ_j are linearly independent to ensure that there are no redundant parameters and that the matrices involved in the computations are full rank. In general, $k \ll |\mathcal{S}|$ and $k \ll |\mathcal{S}||\mathcal{A}|$ and the basis functions ϕ_j have small compact descriptions. As a result, the storage requirements of a linear approximation architecture are much smaller than those of the tabular representation of a value function. There is a large variety of linear approximation architectures, and in fact, many common schemes for value-function approximation can be cast as linear architectures.

– *Lookup Table*. This is the exact tabular representation (There is no approximation under this scheme; it is included only to illustrate that exact representation belongs in the family of linear architectures.) suitable for problems with discrete state variables. Each basis function is an indicator function whose value is 1 only for a specific discrete input point (state or state-action) and 0 otherwise. Each parameter stores one value/entry of the table.
– *Discretization*. This is a common technique for turning a continuous space into discrete using a uniform- or variable-resolution grid. One indicator basis function is assigned to each cell of the discretization, and the corresponding parameter holds the value of that cell.

– *Tile Coding (CMAC)*. This scheme utilizes several overlapping discretizations (tilings) for better accuracy. It generates indicator basis functions for each cell of each tiling and concatenates the basis functions for all tilings. Each parameter corresponds to one cell in one tiling, but the value at each input point is computed additively from the values of all containing cells from all tilings.
– *State Aggregation*. This is a family of schemes where "similar" (by some metric) states are grouped together and are treated as one state. The similarity metric is usually formed through dimensionality reduction techniques for identifying the most significant dimensions in a high-dimensional input space, unlike conventional proximity measures in the same space. There is one indicator basis function for each group and a single value for all states in the group.
– *Polynomials*. This is a generic approximation scheme suitable for problems with several (continuous) state variables. Each basis function is a polynomial term composed of state variables up to a certain degree.
– *Radial Basis Functions (RBFs)*. This is another generic approximation scheme suitable for continuous state variables. Each basis function is a Gaussian with fixed mean and variance; the Gaussians are topologically arranged so that they cover the input space with some overlap.
– *Kernel Methods*. Kernels are symmetric functions between two points, and they are used to represent compactly dot products of feature vectors in high- or even infinite-dimensional spaces. The compactness of kernels allows for approximation schemes that essentially enjoy the flexibility provided by a huge or infinite number of basis functions. The basis functions, in this case, are implicitly defined through the choice of the kernel.
– *Partitioning*. This technique is used for constructing complex approximators by partitioning the state space in several subsets and using a different approximator in each one of them. If linear architectures are used in all partitions, then a set of basis functions for the global architecture can be constructed by concatenating the basis functions of the smaller linear architectures multiplying each subset with an indicator function for the corresponding partition.

Linear architectures offer several advantages: they are easy to implement and use, and their behavior is fairly transparent, both from an analysis standpoint and from a debugging and feature engineering standpoint. It is usually relatively easy to get some insight into the reasons for which a particular choice of features succeeds or fails. This is facilitated by the fact that the magnitude of each parameter is related to the importance of the corresponding feature in the approximation (assuming normalized features).

A *nonlinear approximation architecture* approximates a function $V^\pi(s)$ or $Q^\pi(s,a)$ using arbitrary parametric functions of s and (s,a), possibly in conjunction with some *features* ϕ computed over s and (s,a). The best-known representative of this category is the multilayer feed-forward neural networks, which use one or more layers of linear combinations followed by a nonlinear sigmoidal transformation (thresholding). In their simplest form (one layer), neural networks approximate value functions as

$$\begin{aligned}
\hat{V}^\pi(s;w,\theta) &= \sum_{i=1}^{m} \theta_i \sigma\left(\sum_{j=1}^{k} \phi_j(s) w_{ji}\right) \\
&= \sum_{i=1}^{m} \theta_i \sigma\Big(\phi(s)^\top w_i\Big) \\
\hat{Q}^\pi(s,a;w,\theta) &= \sum_{i=1}^{m} \theta_i \sigma\left(\sum_{j=1}^{k} \phi_j(s,a) w_{ji}\right) \\
&= \sum_{i=1}^{m} \theta_i \sigma\Big(\phi(s,a)^\top w_i\Big).
\end{aligned}$$

Common choices for the differentiable, bounded, and monotonically increasing function σ are the hyperbolic tangent function $\sigma(x) = \tanh(x) = (e^x - e^{-x})/(e^x + e^{-x})$ and the logistic function $\sigma(x) = 1/(1 + e^{-x})$. The free *parameters* of the architecture (also called *weights*) are the coefficients w_{ji} of the linear combinations of the inputs and the coefficients θ_i of the linear combination of the sigmoidal transformations for the output. Notice that the parameters w_{ji} enter nonlinearly into the approximation.

A key question in all approximation architectures is *how* features are generated and selected. The *feature generation and selection* problem is an open question that spans most of machine learning research and admits no easy and general answer. Prior domain-specific knowledge and experience can be very helpful in choosing appropriate features. Several recent studies also describe methods for automatically generating features targeted for value-function approximation (Menache et al. 2005; Mahadevan and Maggioni 2007; Parr et al. 2007).

Learning

Learning (or *training* or *parameter estimation*) in value-function approximation refers to parameter tuning methods that take as input a policy π, an approximation architecture for V^π/Q^π, and the full MDP model or samples of interaction with the process and output a set of parameters w^π such that $\hat{V}^\pi/\hat{Q}^\pi$ is a good approximation to V^π/Q^π. Learning also covers methods for the harder problem of taking an approximation architecture for V^*/Q^* and the model or samples and outputting a set of parameters w^* such that $\hat{V}^*/\hat{Q}^*$ is a good approximation to V^*/Q^*. The former problem is somewhat easier because the policy π, unlike an optimal policy π^*, is known, and therefore in the presence of a simulator of the process, the value function can be estimated at any isolated point using Monte Carlo estimation techniques based on repeated policy rollouts from that point. Each rollout is an execution of an episode starting from a state s (or state-action (s, a)) using policy π to obtain an unbiased estimate of the return of the policy from s (or (s, a)). In this case, value-function approximation can be cast as a classic supervised learning problem; the true value of V^π/Q^π is estimated at a subset of points to form a training set, which can be subsequently used to train the approximation architecture using supervised learning techniques. However, in the absence of a simulator or when seeking approximations to V^*/Q^*, evaluation and projection into the architecture have to be carried out simultaneously.

The true value function has two key properties: it satisfies the Bellman equations, and it is the fixed point of the Bellman operator. Learning in value-function approximation strives to find values for the free parameters so that the approximate function retains at least one of these properties to the extent this is possible. Learning methods that focus on satisfying the Bellman equations attempt to find an approximate function that minimizes the Bellman residual, the difference between the left- and the right-hand sides of the system of Bellman equations. On the other hand, learning methods that focus on the fixed point property attempt to find an approximate function that exhibits a fixed point behavior under the combined application of the Bellman operator and projection onto the space spanned by the basis functions. Experimental evidence suggests that it is really hard to satisfy both properties under approximation, and therefore these two approaches differ significantly in their solutions. The majority of existing learning methods focus on fixed point approximation, which experimentally has been shown to exhibit more stable behavior and delivers better policies. There are also proposals for combining the benefits of both approaches into a hybrid method (Johns et al. 2009).

The most widely used learning approach is based on gradient descent and is applicable to any approximation architecture that is differentiable with respect to its parameters. Any stochastic approximation learning method for tabular representations of value functions can be extended to approximate representations. For example, given any sample (s, a, r, s') of interaction with the process, the temporal difference (TD) learning update rule

$$V^\pi(s) \leftarrow V^\pi(s) + \alpha\Big(r + \gamma V^\pi(s') - V^\pi(s)\Big)$$

for tabular representations, where $\alpha \in (0,1]$ is the learning rate, becomes

$$w^{\pi} \leftarrow w^{\pi} + \alpha\Big(r + \gamma \hat{V}^{\pi}(s'; w^{\pi}) - \hat{V}^{\pi}(s; w^{\pi})\Big)\nabla_{w^{\pi}} \hat{V}^{\pi}(s; w^{\pi})$$

under an approximation scheme $\hat{V}^{\pi}$. Similarly, the SARSA update rule

$$Q^{\pi}(s,a) \leftarrow Q^{\pi}(s,a) + \alpha\Big(r + \gamma Q^{\pi}(s', \pi(s')) - Q^{\pi}(s,a)\Big)$$

for tabular representations becomes

$$w^{\pi} \leftarrow w^{\pi} + \alpha\Big(r + \gamma \hat{Q}^{\pi}(s', \pi(s'); w^{\pi}) - \hat{Q}^{\pi}(s,a; w^{\pi})\Big)\nabla_{w^{\pi}} \hat{Q}^{\pi}(s,a; w^{\pi})$$

under an approximation scheme $\hat{Q}^{\pi}$. Finally, the Q-learning update rule

$$Q^{*}(s,a) \leftarrow Q^{*}(s,a) + \alpha\Big(r + \gamma \max_{a' \in \mathcal{A}} \{Q^{*}(s',a')\} - Q^{*}(s,a)\Big)$$

for tabular representations under an approximation scheme $\hat{Q}^{*}$ becomes

$$w^{*} \leftarrow w^{*} + \alpha\Big(r + \gamma \max_{a' \in \mathcal{A}} \{\hat{Q}^{*}(s',a'; w^{*})\} - \hat{Q}^{*}(s,a; w^{*})\Big)\nabla_{w^{*}} \hat{Q}^{*}(s,a; w^{*}) \,.$$

These rules are applicable to any approximation architecture, linear or nonlinear. However, when using linear architectures they can be greatly simplified, because the gradient with respect to a parameter w_j is simply the corresponding basis function ϕ_j, for $j = 1, 2, \ldots, k$.

$$\text{TD: } w_j^{\pi} \leftarrow w_j^{\pi} + \alpha\Big(r + \gamma \phi(s')^{\top} w^{\pi} - \phi(s)^{\top} w^{\pi}\Big)\phi_j(s)$$

$$\text{SARSA: } w_j^{\pi} \leftarrow w_j^{\pi} + \alpha\Big(r + \gamma \phi(s', \pi(s'))^{\top} w^{\pi} - \phi(s,a)^{\top} w^{\pi}\Big)\phi_j(s,a)$$

$$Q\text{-learning: } w_j^{*} \leftarrow w_j^{*} + \alpha\Big(r + \gamma \max_{a' \in \mathcal{A}} \{\phi(s',a')^{\top} w^{*}\} - \phi(s,a)^{\top} w^{*}\Big)\phi_j(s,a)$$

More sophisticated learning approaches rely on retaining the desired value-function property in a batch manner by processing several samples collectively. A variety of least-squares techniques have been proposed for linear architectures giving rise to several least-squares reinforcement learning methods, such as least-squares temporal difference (LSTD) learning (Bradtke and Barto 1996), least-squares policy evaluation (LSPE) (Nedić and Bertsekas 2003), hybrid least-squares approximation (Johns et al. 2009), and least-squares policy iteration (LSPI) (Lagoudakis and Parr 2003). The parameters of a linear architecture can also be estimated using Linear Programming (de Farias and Van Roy 2003). Specialized learning algorithms have been proposed when using a kernel-based approximation architecture, based either on Gaussian process TD (GPTD) (Engel et al. 2003), Gaussian process SARSA (GPSARSA) (Engel et al. 2005), kernelized LSTD (KLSTD) and LSPI (KLSPI) (Xu et al. 2007), support vector regression (Bethke et al. 2008), or Gaussian process regression (Rasmussen and Kuss 2004; Bethke and How 2009). A unified view of kernelized value-function approximation is offered by Taylor and Parr (2009). On the other hand, boot-strapping – the updating of a value estimate based on other value estimates – is the main learning approach behind batch methods for nonlinear architectures, such as fitted Q-iteration (FQI) (Ernst et al. 2005).

Examples

Very close approximations of the state value function of optimal policies in two well-known problems are presented to illustrate the difficulty of value-function approximation. To obtain these close approximations, a fine discretization of the two-dimensional state space into a uniform grid of 250×250 was used for representation. The state-action value function Q was initially computed using approximate policy iteration (a sparse-matrix version of LSPI) with a set of indicator basis functions over the state grid and all actions and 500 (s, a, r, s') samples for each one of the 187,500 discrete cells (s, a), until convergence to a near-optimal policy; the state value function V was extracted from the Q values.

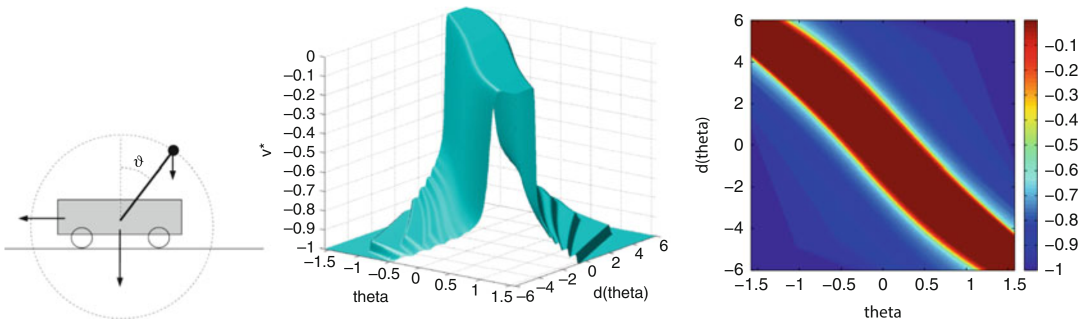
Inverted Pendulum

The *inverted pendulum* problem is to balance a pendulum of unknown length and mass at the upright position by applying forces to the cart it is attached to (Fig. 1, left). Three actions are allowed: left force LF (−50 Newtons), right force RF (+50 Newtons), or no force NF (0 Newtons). All three actions are noisy; Gaussian noise with $\mu = 0$ and $\sigma^2 = 10$ is added to the chosen action. The state space of the problem is continuous and consists of the vertical angle θ and the angular velocity $\dot{\theta}$ of the pendulum. The transitions are governed by the nonlinear dynamics of the system and depend on the current state and the current (noisy) control u:

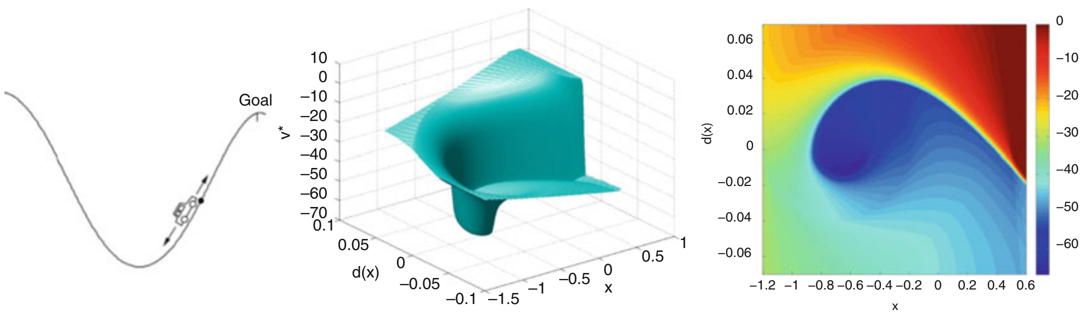
$$\ddot{\theta} = \frac{g\sin(\theta) - \alpha m l(\dot{\theta})^2 \sin(2\theta)/2 - \alpha\cos(\theta)u}{4l/3 - \alpha m l\cos^2(\theta)} \ ,$$

where g is the gravity constant ($g = 9.8\,\mathrm{m/s^2}$), m is the mass of the pendulum (default: $m = 2.0\,\mathrm{kg}$), M is the mass of the cart (default: $M = 8.0\,\mathrm{kg}$), l is the length of the pendulum (default: $l = 0.5\,\mathrm{m}$), and $\alpha = 1/(m + M)$. The simulation step is 0.1 s, thus the control input is given at a rate of 10 Hz, at the beginning of each time step, and is kept constant during any time step. A reward of 0 is given as long as the angle of the pendulum does not exceed $\pi/2$ in absolute value (the pendulum is above the horizontal line). An angle greater than $\pi/2$ in absolute value signals the end of the episode and a reward (penalty) of −1. The discount factor of the process is 0.95.

Figure 1 shows a close approximation to the state value function V^* of an optimal policy for the inverted pendulum domain over the two-dimensional state space $(\theta, \dot{\theta})$. The value function indicates that states which potentially offer high return are clustered within a zone where θ and $\dot{\theta}$ have different signs and therefore the gravity force can be counteracted. Notice the nonlinearity of the function and the difficult approximation problem it presents.



Value Function Approximation, Fig. 1 Inverted pendulum: state value function of an optimal policy (3D and 2D) (Courtesy of Ioannis Rexakis)



Value Function Approximation, Fig. 2 Mountain car: state value function of an optimal policy (3D and 2D) (Courtesy of Ioannis Rexakis)

Mountain Car

The *mountain car* problem is to drive an underpowered car from the bottom of a valley between two mountains to the top of the mountain on the right (Fig. 2, left). The car is not powerful enough to climb any of the hills directly from the bottom of the valley even at full throttle; it must build some energy by climbing first to the left (moving away from the goal) and then to the right. Three actions are allowed: forward throttle FT (+1), reverse throttle RT (−1), or no throttle NT (0). All three actions are noisy; Gaussian noise with $\mu = 0$ and $\sigma^2 = 0.2$ is added to the chosen action. The state space of the problem is continuous and consists of the position x and the velocity $\dot{x}$ of the car along the horizontal axis. The transitions are governed by the nonlinear dynamics of the system and depend on the current state $(x(t), \dot{x}(t))$ and the current (noisy) control $u(t)$:

$$\begin{aligned} x(t+1) &= \textsc{Bound}_x[x(t) + \dot{x}(t+1)] \\ \dot{x}(t+1) &= \textsc{Bound}_{\dot{x}}[\dot{x}(t) \\ &\qquad +0.001u(t) - 0.0025\cos(3x(t))] \,, \end{aligned}$$

where $\textsc{Bound}_x$ is a function that keeps x within $[-1.2, 0.5]$, while $\textsc{Bound}_{\dot{x}}$ keeps $\dot{x}$ within $[-0.07, 0.07]$. If the car hits the bounds of the position x, the velocity $\dot{x}$ is set to zero. A penalty of -1 is given at each step as long as the position of the car is below the right bound (0.5). As soon as the car position hits the right bound of the position, it has reached the goal; the episode ends successfully and a reward of 0 is given. The discount factor of the process is 0.99.

Figure 2 shows a close approximation to the state value function V^* of an optimal policy for the mountain car domain over the two-dimensional state space $(x, \dot{x})$. The value function indicates that in order to gain high return, the car has to follow a spiral in the state space that goes through states with progressively higher value. In practice, this means that the car has to move back and forth between the two mountains until sufficient energy is built to escape from the valley. Again, notice the high nonlinearity of the function and the hard approximation problem it presents.

Notation

The table summarizes the differences in names and symbols between the common notation (adopted here) and the alternative notation used in the literature.

Common notation		Alternative notation	
Name	**Symbol**	**Symbol**	**Name**
State space	$\mathcal{S}$	S	States
State	s, s'	i, j	State
Action space	$\mathcal{A}$	U	Controls
Action	a	u	Control
Transition model	$\mathcal{P}(s'\|s, a)$	$p_{ij}(u)$	Transition probabilities
Reward function	$\mathcal{R}$	g	Cost function
Discount factor	γ	α	Discount factor
Policy	π	μ	Policy
State value function	V	J	Cost-to-go function
State-action value function	Q	Q	Q-factors
Parameters/weights	w	r	Parameters
Learning rate	α	γ	Step size

Cross-References

▶ Curse of Dimensionality
▶ Dynamic Programming
▶ Feature Selection
▶ Gaussian Process Reinforcement Learning
▶ Least-Squares Reinforcement Learning Methods
▶ Q-Learning
▶ Radial Basis Function Approximation
▶ Reinforcement Learning
▶ Relational Value Iteration
▶ Temporal Difference Learning

Recommended Reading

Bertsekas DP, Tsitsiklis JN (1996) Neuro-dynamic programming. Athena Scientific, Belmont

Bethke B, How JP (2009) Approximate dynamic programming using Bellman residual elimination and Gaussian process regression. In: Proceedings of the American control conference, St. Louis, pp 745–750

Bethke B, How JP, Ozdaglar A (2008) Approximate dynamic programming using support vector regression. In: Proceedings of the IEEE conference on decision and control, Cancun, pp 745–750

Bradtke SJ, Barto AG (1996) Linear least-squares algorithms for temporal difference learning. Mach Learn 22(1–3):33–57

Buşoniu L, Babuška R, Schutter BD, Ernst D (2010) Reinforcement learning and dynamic programming using functions approximators. CRC, Boca Raton

de Farias DP, Van Roy B (2003) The linear programming approach to approximate dynamic programming. Oper Res 51(6):850–865

Engel Y, Mannor S, Meir R (2003) Bayes meets Bellman: the Gaussian process approach to temporal difference learning. In: Proceedings of the international conference on machine learning (ICML), Washington, DC, pp 154–161

Engel Y, Mannor S, Meir R (2005) Reinforcement learning with Gaussian processes. In: Proceedings of the international conference on machine learning (ICML), Bonn, pp 201–208

Ernst D, Geurts P, Wehenkel L (2005) Tree-based batch mode reinforcement learning. J Mach Learn Res 6:503–556

Johns J, Petrik M, Mahadevan S (2009) Hybrid least-squares algorithms for approximate policy evaluation. Mach Learn 76(2–3):243–256

Lagoudakis MG, Parr R (2003) Least-squares policy iteration. J Mach Learn Res 4:1107–1149

Mahadevan S, Maggioni M (2007) Proto-value functions: a Laplacian framework for learning representation and control in Markov decision processes. J Mach Learn Res 8:2169–2231

Menache I, Mannor S, Shimkin N (2005) Basis function adaptation in temporal difference reinforcement learning. Ann Oper Res 134(1):215–238

Nedić A, Bertsekas DP (2003) Least-squares policy evaluation algorithms with linear function approximation. Discret Event Dyn Syst Theory Appl 13(1–2):79–110

Parr R, Painter-Wakefield C, Li L, Littman M (2007) Analyzing feature generation for value-function approximation. In: Proceedings of the international conference on machine learning (ICML), Corvallis, pp 449–456

Puterman ML (1994) Markov decision processes: discrete stochastic dynamic programming. Wiley, New York

Rasmussen CE, Kuss M (2004) Gaussian processes in reinforcement learning. In: Thrun S, Saul LK, Scholkopf B (eds) Advances in neural information

processing systems (NIPS). MIT Press, Cambridge pp 751–759
Sutton R, Barto A (1998) Reinforcement learning: an introduction. MIT, Cambridge
Taylor G, Parr R (2009) Kernelized value function approximation for reinforcement learning. In: Proceedings of the international conference on machine learning (ICML), Toronto, pp 1017–1024
Xu X, Hu D, Lu X (2007) Kernel-based least-squares policy iteration for reinforcement learning. IEEE Trans Neural Netw 18(4):973–992

Variance Hint

▶ Inductive Bias

VC Dimension

Thomas Zeugmann
Hokkaido University, Sapporo, Japan

Motivation and Background

We define an important combinatorial parameter that measures the combinatorial complexity of a family of subsets taken from a given universe (learning domain) X. This parameter was originally defined by Vapnik and Chervonenkis (1971) and is thus commonly referred to as Vapnik-Chervonenkis dimension, commonly abbreviated as VC *dimension*. Subsequently, Dudley (1978, 1979) generalized Vapnik and Chervonenkis' (1971) results. The reader is also referred to Vapnik's (2000) book in which he greatly extends the original ideas. This results in a theory which is called structural risk minimization.

The importance of the VC dimension for ▶ PAC learning was discovered by Blumer et al. (1989) who introduced the notion to computational learning theory.

As Anthony and Biggs (1992, Page 71) have put it, "*The development of this notion is probably the most significant contribution that mathematics has made to Computational Learning Theory.*"

Recall that we use $|S|$ and $\wp(S)$ to denote the cardinality and the power set of any set S, respectively. We first define the VC dimension and provide a short explanation of its importance for ▶ PAC learning. Then we present some examples.

Definition

Let $X \neq \emptyset$ be any learning domain, let $\mathcal{C} \subseteq \wp(X)$ be any nonempty concept class, and let $S \subseteq X$ be any finite set. We set

$$\Pi_{\mathcal{C}}(S) = \{S \cap c \mid c \in \mathcal{C}\} .$$

1. S is said to be *shattered* by $\mathcal{C}$ iff $\Pi_{\mathcal{C}}(S) = \wp(S)$.
2. The VC *dimension* of $\mathcal{C}$ is the cardinality of the largest finite set $S \subseteq X$ that is shattered by $\mathcal{C}$.

   If arbitrary large finite sets S are shattered by $\mathcal{C}$, then the VC dimension of $\mathcal{C}$ is defined to be infinite.

Notation: By VC($\mathcal{C}$) we denote the VC *dimension* of $\mathcal{C}$.

Remarks

As far as ▶ PAC learning is concerned, for a sample set S, the notion $\Pi_{\mathcal{C}}(S)$ has the following meaning. Essentially, $\Pi_{\mathcal{C}}(S)$ collects the set of *all subsets* of the sample set S which are made positive by some concept $c \in \mathcal{C}$. Consequently, $S \cap c$ represents the elements of S that are labeled as to be positive by the concept c. Hence, $\Pi_{\mathcal{C}}(S)$ is the collection of all such subsets taken over all $c \in \mathcal{C}$. If *every* subset of S can be labeled as to be positive by some concept $c \in \mathcal{C}$ and c does not make any other element of S positive, then S is shattered.

If VC($\mathcal{C}$) $= d$ then there *exists* a finite set $S \subseteq X$ such that $|S| = d$, and S is shattered by $\mathcal{C}$. Moreover, *every* set $S \subseteq X$ with $|S| > d$ is *not* shattered by $\mathcal{C}$.

It is intuitively clear that an infinite VC dimension might enormously complicate learning. On the other hand, it is by no means obvious that a finite VC dimension may always guarantee the learnability of the corresponding concept class. However, this is a central theorem of the ▸ PAC learning theory. Moreover, the value of the VC dimension is a measure of the sample complexity. This holds for PAC learning and beyond. Further models where this is true comprise the ▸ online learning models (cf. Haussler et al. 1994; Maass and Turán 1990; Littlestone 1988), models of ▸ query-based learning (cf. Maass and Turán 1990), and others.

Examples

First, let $\mathcal{C}$ be any finite concept class. Then, since it requires 2^d distinct concepts to shatter a set of cardinality d, no set of cardinality larger than $\log|\mathcal{C}|$ can be shattered. Thus, $\log|\mathcal{C}|$ is always an upper bound for the VC dimension of finite concept classes. Here log denotes the logarithm to the base 2.

However, if the VC dimension can be determined, it usually gives a better bound than $\log|\mathcal{C}|$. To see this, let $\mathcal{L}_n = \{x_1, \bar{x}_1, x_2, \bar{x}_2, \ldots, x_n, \bar{x}_n\}$, $n \geq 1$ be a set of literals and let $X = \{0,1\}^n$ be the n-dimensional Boolean learning domain. Furthermore, let $\mathcal{C}_n \subseteq \wp(X)$ be the class of all concepts describable by a monomial including the empty monomial (representing $\{0,1\}^n$) and the conjunction of all literals (representing $\emptyset$). Then $|\mathcal{C}_n| = 3^n + 1$ and thus $\mathrm{VC}(\mathcal{C}) \leq n(\log 3) + 1$. But $\mathrm{VC}(\mathcal{C}_n) = n$ for all $n \geq 2$ and $\mathrm{VC}(\mathcal{C}_1) = 2$ as shown by Natschläger and Schmitt (1996). Note that the same is true for the class of all concepts describable by *monotone monomials*, i.e., monomials containing only non-negated literals.

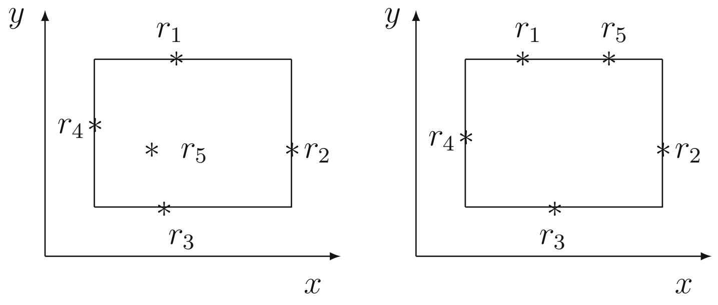
Next, we consider the concept class $\mathcal{C}$ of all axis-parallel rectangles. So let $X = \mathbb{E}^2$ be the two-dimensional Euclidean space and $\mathcal{C} \subseteq \wp(\mathbb{E}^2)$ be the set of all axis-parallel rectangles, i.e., products of intervals on the x-axis with intervals on the y-axis. Then, it is not hard to see that $\mathrm{VC}(\mathcal{C}) = 4$.

Clearly, we can shatter the empty set and sets of cardinality 1, 2, and 3. Now, let $S = \{r_1, r_2, r_3, r_4\}$ be such that r_1, r_2, r_3, r_4 are the middle points of the sides of some square. Then it is not hard to see that there are 16 concepts c_i, $1 \leq i \leq 16$, in $\mathcal{C}$ such that $\wp(S) = \{S \cap c_i \mid 1 \leq i \leq 16\}$. Hence, $\mathrm{VC}(\mathcal{C}) \geq 4$.

Next, let $S = \{r_1, r_2, r_3, r_4, r_5\}$ be any set of 5 pairwise different points. Let c be the smallest closed axis-parallel rectangle containing the points of S. Since c has only four sides, there must be some point $r \in S$, say r_5, such that r_5 lies either in the interior of c or r_5 lies on some side of c along with another point of S (cf. Fig. 1). Suppose S is shattered by $\mathcal{C}$. Then, there has to be a concept $c \in \mathcal{C}$ such that $\{r_1, r_2, r_3, r_4\} = S \cap c$. However, by construction we obtain that $\{r_1, r_2, r_3, r_4\} = S \cap c$ implies $r_5 \in S \cap c$, a contradiction. Thus, *no* set of cardinality 5 is shattered. Hence, $\mathrm{VC}(\mathcal{C}) = 4$.

The latter result can be easily generalized. Let $X = \mathbb{E}^n$, and let $\mathcal{C}$ be the set of all axis-parallel parallelepipeds in $\mathbb{E}^n$. Then $\mathrm{VC}(\mathcal{C}) = 2n$.

A further generalization is as follows. Let X be the real line (one-dimensional Euclidean



VC Dimension, Fig. 1 No set of cardinality 5 can be shattered by axis-parallel rectangles space), i.e., $X = \mathbb{E}$, and let $\mathcal{C}$ be the set of all unions of at most s (closed or open) intervals for some fixed constant $s \geq 1$. Let $S = \{x_i \mid 1 \leq i \leq 2s,\ x_i < x_{i+1}$ for all $1 \leq i < 2s\}$. Then one easily verifies that S is shattered by $\mathcal{C}$. Hence, $\mathrm{VC}(\mathcal{C}) \geq 2s$. On the other hand, if S is any set of $2s + 1$ pairwise different points with $x_i < x_{i+1}$ for all $1 \leq i \leq 2s$, then no concept in $\mathcal{C}$ contains $x_1, x_3, \ldots, x_{2s+1}$ without also containing a point in $x_2, x_4, \ldots, x_{2s}$. Thus, no such S is shattered. Consequently, we have $\mathrm{VC}(\mathcal{C}) = 2s$.

Furthermore, we can generalize the observations made above by deriving some rules that turn out to be very useful to estimate the VC dimension of more complicated concept classes provided they can be constructed from simpler classes.

First, let $\mathcal{C}_1$ and $\mathcal{C}_2$ be concept classes such that $\mathcal{C}_1 \subseteq \mathcal{C}_2$. Then we clearly have

$$\mathrm{VC}(\mathcal{C}_1) \leq \mathrm{VC}(\mathcal{C}_2)\ .$$

Second, let X be any learning domain, let $\mathcal{C} \subseteq \wp(X)$ and define the complement of $\mathcal{C}$ to be $\overline{\mathcal{C}} = \{X \setminus c \mid c \in \mathcal{C}\}$. Then we have

$$\mathrm{VC}(\overline{\mathcal{C}}) = \mathrm{VC}(\mathcal{C})\ .$$

Third, consider two concept classes $\mathcal{C}_1$ and $\mathcal{C}_2$ defined over the same learning domain X. Let $\mathcal{C} = \mathcal{C}_1 \cup \mathcal{C}_2$ be the union of $\mathcal{C}_1$ and $\mathcal{C}_2$. Then,

$$\mathrm{VC}(\mathcal{C}) \leq \mathrm{VC}(\mathcal{C}_1) + \mathrm{VC}(\mathcal{C}_2) + 1\ .$$

Fourth, let $\mathcal{C}$ be any concept class such that $\mathrm{VC}(\mathcal{C}) = d$. Consider the $\mathcal{C}_s$ union (or intersection) of at most s concepts from $\mathcal{C}$, where $s \geq 1$ is any fixed constant, i.e., $\mathcal{C}_s = \{c \mid c = \bigcup_{1 \leq i \leq s} c_i\ ,\ c_i \in \mathcal{C}\}$ (or $\mathcal{C}_s = \{c \mid c = \bigcap_{1 \leq i \leq s} c_i\ ,\ c_i \in \mathcal{C}\}$). Then one can show that

$$\mathrm{VC}(\mathcal{C}_s) \leq 2ds \cdot \log(3s)\ .$$

Numerous further examples can be found in, e.g., Vapnik and Chervonenkis (1974), Haussler and Welz (1987), Anthony and Bartlett (1999), Wenocur and Dudley (1981), Karpinski and Werther (1994), Karpinski and Macintyre (1995), Sakurai (1995), and Mitchell et al. (1999).

Applications

Let us return to the notion $\Pi_{\mathcal{C}}(S)$ and let us generalize it a bit as follows. For any natural number $m \in \mathbb{N}$ and any nonempty concept class $\mathcal{C} \subseteq \wp(S)$, we set

$$\Pi_{\mathcal{C}}(m) = \max\{|\Pi_{\mathcal{C}}(S)| \mid S \subseteq X,\ |S| = m\}\ .$$

We can use the new notion to give an equivalent definition of the VC dimension of a concept class $\mathcal{C}$, i.e.,

$$\mathrm{VC}(\mathcal{C}) = \max\{d \mid d \in \mathbb{N},\ \Pi_{\mathcal{C}}(d) = 2^d\}\ .$$

Looking at $\Pi_{\mathcal{C}}(m)$ from the perspective of learning, we see the following. The argument m refers to the sample size. $\Pi_{\mathcal{C}}(m)$ is describing the maximum number of ways a sample of size m can be labeled by concepts taken from $\mathcal{C}$. Hence, the number $\Pi_{\mathcal{C}}(m)$ behaves as a measure of concept class complexity. What can be said about $\Pi_{\mathcal{C}}(m)$? Suppose $d = VC(\mathcal{C})$; then $m \leq d$ implies $\Pi_{\mathcal{C}}(m) = 2^m$. On the other hand, $m > d$ directly implies $\Pi_{\mathcal{C}}(m) < 2^m$. Therefore, we are interested in learning how fast $\Pi_{\mathcal{C}}(m)$ really grows provided $m > d$. The key ingredient to obtain the desired information is usually referred to as Sauer's Lemma (cf. Sauer 1972). Under the assumptions made above, it states that

$$\Pi_{\mathcal{C}}(m) \leq \sum_{i=0}^{d} \binom{m}{i},$$

$$\text{where} \quad \binom{m}{i} = 0 \quad \text{if } i > m\ .$$

Like many important results, Sauer's Lemma (cf. Sauer 1972) has several proofs and generalizations have been studied, too. We refer the reader to Anthony and Biggs (1992), Kearns and Vazirani (1994), and Gurvits (1997) for a more detailed exposition.

Let us first look at the case $m \leq d$ already considered. For this case Sauer's Lemma is telling us that

$$\Pi_{\mathcal{C}}(m) \leq \sum_{i=0}^{d} \binom{m}{i} = 2^m,$$

and thus, we get an exponential bound. The interesting aspect is that in the remaining cases, the bound is *polynomial*. For simplifying notation, we set

$$\Phi(d,m) = \sum_{i=0}^{d} \binom{m}{i}.$$

Using combinatorial arguments and Stirling approximation, one can show that

1. $\Phi(0,m) = \binom{m}{0} = 1$ for all $m \in \mathbb{N}$.
2. $\Phi(d,1) = \binom{1}{0} + \binom{1}{1} = 2$ for all $d \in \mathbb{N}, d \geq 1$.
3. $\Phi(d,m) = \Phi(d,m-1) + \Phi(d-1,m-1)$ for all $d,m \in \mathbb{N}$, $d \geq 1$, $m \geq 2$.
4. $\Phi(d,m) \leq m^d + 1$ for all $d \geq 0, m \geq 0$.
5. $\Phi(d,m) \leq m^d$ for all $d \geq 2, m \geq 2$.
6. $\Phi(d,m) \leq (\frac{em}{d})^d$ for all $m \geq d \geq 1$.

That is, (4) through (6) provide a bound polynomial in m for $\Pi_{\mathcal{C}}(m)$ whenever $\mathrm{VC}(\mathcal{C})$ is finite. This insight is fundamental for ▶ PAC learning and other learning models.

Linial et al. (1991) initiated the study of the complexity problem of computing the VC dimension of a finite family of concepts defined over a finite learning domain. Given any finite learning domain X of cardinality n and any concept class $\mathcal{C} \subseteq \wp(X)$ of cardinality r, one can represent the concept class $\mathcal{C}$ by an $r \times n$ matrix M such that $M_{ij} = 1$ iff $x_j \in c_i$. Then each row of M represents a concept $c \in \mathcal{C}$ and each column represents a point in X. The *discrete VC dimension decision problem* is then, given a $\{0,1\}$-valued matrix M and an integer $d \geq 1$ as input, to decide whether or not $VC(\mathcal{C}) \leq d$, and the *discrete VC dimension problem* is, given a $\{0,1\}$-valued matrix M as input, to determine $VC(\mathcal{C})$.

Linial et al. (1991) showed that the discrete VC dimension decision problem to be solvable in time $O(rn^d)$ and that the discrete VC dimension problem can be solved in time $O(rn^{\log r})$. Further progress was made by Shinohara (1995) who showed that the discrete VC dimension decision problem is in the complexity class $\mathrm{SAT}_{\log^2 n}$ and hard for the complexity class $\mathrm{SAT}^{\mathrm{CNF}}_{\log^2 n}$, where $\mathcal{P} \subseteq \mathrm{SAT}^{\mathrm{CNF}}_{\log^2 n} \subseteq \mathrm{SAT}_{\log^2 n} \subseteq \mathcal{NP}$ (see Shinohara (1995) for details). Moreover, Papadimitriou and Yannakakis (1996) defined a new complexity class $\mathcal{LOGNP}$ and showed the VC dimension decision problem to be complete for this class.

However, the matrix representation of a concept class may be exponentially larger than a parameterized representation of it, e.g., the concept class may be generated by a circuit. Representing concept classes by circuits, Schaefer (1999) showed the discrete VC dimension problem (modified in the canonical way) to be Σ_3^p complete. For a definition of the complexity class Σ_3^p, we refer to Arora and Barak (2009).

Furthermore, we refer the reader to Goldberg and Jerrum (1995) who succeeded in bounding the VC dimension of concept classes parameterized by real numbers.

Finally, the notion of the VC dimension can be generalized to sets of indicator functions and to sets of real functions (cf. Vapnik 2000, Section 3.6). These generalizations play an important role in statistical learning theory.

Cross-References

▶ Epsilon Nets
▶ PAC Learning
▶ Statistical Machine Translation
▶ Structural Risk Minimization

Recommended Reading

Anthony M, Bartlett PL (1999) Neural network learning: theoretical foundations. Cambridge University Press, Cambridge

Anthony M, Biggs N (1992) Computational learning theory. Cambridge tracts in theoretical computer science, Vol 30. Cambridge University Press, Cambridge

Arora S, Barak B (2009) Computational complexity: A Modern approach. Cambridge University Press, Cambridge

Blumer A, Ehrenfeucht A, Haussler D, Warmuth MK (1989) Learnability and the Vapnik-Chervonenkis dimension. J ACM 36(4):929–965

Dudley RM (1978) Central limit theorems for empirical measures. Ann Probab 6(6):899–929

Dudley RM (1979) Corrections to "Central limit theorems for empirical measures". Ann Probab 7(5):909–911

Goldberg PW, Jerrum MR (1995) Bounding the Vapnik-Chervonenkis dimension of concept classes parameterized by real numbers. Mach Learn 18(2-3):131–148

Gurvits L (1997) Linear algebraic proofs of VC-dimension based inequalities. In: Ben-David S (ed) Proceedings of the third european conference on computational learning theory, EuroCOLT '97, Jerusalem, Israel, March 1997, Lecture notes in artificial Intelligence, vol 1208. Springer, pp 238–250

Haussler D, Littlestone N, Warmuth MK (1994) Predicting $\{0, 1\}$ functions on randomly drawn points. Info Comput 115(2):248–292

Haussler D, Welz E (1987) Epsilon nets and simplex range queries. Discret Comput Geom 2:127–151

Karpinski M, Macintyre A (1995) Polynomial bounds for VC dimension of sigmoidal neural networks. In: Proceedings of the 27th annual ACM symposium on theory of computing, ACM Press, New York, pp 200–208

Karpinski M, Werther T (1994) VC dimension and sampling complexity of learning sparse polynomials and rational functions. In: Hanson SJ, Drastal GA, Rivest RL (eds) Computational learning theory and natural learning systems. Constraints and prospects, vol I, chap. 11. MIT Press, pp 331–354

Kearns MJ, Vazirani UV (1994) An Introduction to computational learning theory. The MIT Press, Cambridge, Massachusetts

Linial N, Mansour Y, Rivest RL (1991) Results on learnability and the Vapnik-Chervonenkis dimension. Inform Comput 90(1):33–49

Littlestone N (1988) Learning quickly when irrelevant attributes abound: A new linear-threshold algorithm. Mach Learn 2(4):285–318

Maass W, Turán G (1990) On the complexity of learning from counterexamples and membership queries. In: Proceedings of the 31st annual symposium on foundations of computer science (FOCS 1990), St. Louis, 22-24 October 1990. IEEE Computer Society Press, Los Alamitos, pp 203–210

Mitchell A, Scheffer T, Sharma A, Stephan F (1999) The VC-dimension of subclasses of pattern languages. In: Watanabe O, Yokomori T (eds) Proceedings of the 10th international conference on algorithmic learning theory, ALT '99, Tokyo, Dec 1999, Lecture notes in artificial intelligence, vol 1720. Springer, pp 93–105.

Natschläger T, Schmitt M (1996) Exact VC-dimension of Boolean monomials. Infor Process Lett 59(1): 19–20

Papadimitriou CH, Yannakakis M (1996) On limited nondeterminism and the complexity of the V-C dimension. J Comput Syst Sci 53(2):161–170

Sakurai A (1995) On the VC-dimension of depth four threshold circuits and the complexity of Boolean-valued functions. Theoret Comput Sci 137(1):109–127

Sauer N (1972) On the density of families of sets. J Comb Theory (A) 13(1):145–147

Schaefer M (1999) Deciding the Vapnik-Červonenkis dimension is Σ_3^p-complete. J Comput Syst Sci 58(1): 177–182

Shinohara A (1995) Complexity of computing Vapnik-Chervonenkis dimension and some generalized dimensions. Theoret Comput Sci 137(1):129–144

Vapnik VN (2000) The nature of statistical learning theory, 2nd edn. Springer, Berlin

Vapnik VN, Chervonenkis AY (1971) On the uniform convergence of relative frequencies of events to their probabilities. Theory Probab Appl 16(2):264–280

Vapnik VN, Chervonenkis AY (1974) Theory of pattern recognition. Nauka, Moskwa (In Russian)

Wenocur RS, Dudley RM (1981) Some special Vapnik-Chervonenkis classes. Discret Math 33:313–318

Vector Optimization

▶ Multi-objective Optimization

Version Space

Claude Sammut
The University of New South Wales, Sydney, NSW, Australia

Definition

Mitchell (1977, 1982) defines the *version space* for a learning algorithm as the subset of hypotheses consistent with the training examples. That is, the ▶ hypothesis language is capable of describing a large, possibly infinite, number of concepts. When searching for the target concept, we are only interested in the subset of sentences in the hypothesis language that are consistent with the training examples, where consistent means that the examples are correctly classified (assuming deterministic concepts and no ▶ noise in the data). While the version space may be infinite, it can often be represented in a compact manner by maintaining only its *bounds*, the most specific (▶ Most Specific Hypothesis) and ▶ most general hypotheses. Any hypothesis that is more general than a hypothesis in the most specific bound and more specific than a hypothesis in the most general bound is in the version space.

Cross-References

▶ Learning as Search
▶ Noise

Recommended Reading

Mitchell TM (1977) Version spaces: a candidate elimination approach to rule-learning. In: Proceedings of the fifth international joint conference on artificial intelligence, Cambridge, pp 305–310

Mitchell TM (1982) Generalization as search. Artif Intell 18(2):203–226

Viterbi Algorithm

A dynamic programming algorithm for finding the most likely sequence of hidden states resulting in an observed sequence of output events. The most likely sequence is called the Viterbi path. The Viterbi algorithm was popularized due to its usability in Hidden Markov models (HMM).

The Viterbi algorithm was initially proposed by Andrew Viterbi as an error-correction scheme for noisy digital communication links. It is now also commonly used in speech recognition, natural language processing, and bioinformatics.

Recommended Reading

Viterbi AJ (1967) Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. IEEE Trans Inf Theory 3(2):260–269