# On-Policy Prediction with Approximation

Saturday, June 20, 2020 6:19 PM

In function approximation approaches to reinforcement learning, the state-value function is written as a function of the state and a weight vector $\left(\hat{v}(s,\vec{w}) \approx v_\pi(s)\right)$

Note that this weight vector could represent all of the weights in a neural network

Using function approximation for reinforcement learning makes reinforcement learning applicable to partially observable states (which is equivalent to the functional form not being parameterized for certain aspects of the state)

Function approximation methods can't augment the state representation with memories of past observations

In function approximation many more states exist than weights so making one state's estimate more accurate invariably makes other states estimates less accurate

Mean Squared Value Error:

$$\overline{VE} \equiv \sum_{s\in S} \mu(s)\left[v_\pi(s) - \hat{v}(s,\vec{w})\right]^2$$

Where $\mu(s)$ is the on-policy distribution

On-Policy Distribution:

$$\sum_s \mu(s) = 1$$

This represents how much we care about the error in each state

This is often chosen to be the fraction of total time spent in that state

For Episodic Problems:

$$\mu(s) = \frac{\eta(s)}{\sum_{s'} \eta(s')}$$

$$\eta(s) = h(s) + \sum_{\bar{s}} \eta(\bar{s}) \sum_a \pi(a|\bar{s}) p(s|\bar{s},a)$$

Where $h(s)$ is the probability that an episode starts in state $s$ and $\eta(s)$ represents the average number of time steps spent in state $s$

If there is discounting a factor of $\gamma$ should be added (multiplied) to the second term of $\eta(s)$

Global Optimum:

$$\overline{VE}(\vec{w}^*) \leq \overline{VE}(\vec{w})$$

Where $\vec{w}$ is all possible weight vectors and $\vec{w}^*$ is the optimum weight vector

Local Optimum:

$$\overline{VE}(\vec{w}^*) \leq \overline{VE}(\vec{w})$$

Where $\vec{w}$ is all weight vectors in some neighborhood of $\vec{w}^*$ and $\vec{w}^*$ is the optimum weight vector

Stochastic Gradient Descent (SGD):

$$\vec{w}_{t+1} \equiv \vec{w}_t - \frac{1}{2}\alpha\nabla\left[v_\pi(S_t) - \hat{v}(S_t,\vec{w}_t)\right]^2 = \vec{w}_t + \alpha\left[v_\pi(S_t) - \hat{v}(S_t,\vec{w}_t)\right]\nabla\hat{v}(S_t,\vec{w}_t)$$

Where $\mu$ is a positive step-size parameter (the learning rate) and we have assumed that states appear with the same distribution $\mu$ over which we are trying to minimize the value error and that mean squared value error is being used

This is called stochastic since the update is done on a single example, which may have been selected stochastically

This is guaranteed to converge to a local optimum if the standard stochastic approximation conditions are met

Note that this is optimizing the inputs to the function (the weights) and not the function iteself

Vector Gradient:

$$\nabla f(\vec{w}) \equiv \left(\frac{\partial f(\vec{w})}{\partial w_1}, \frac{\partial f(\vec{w})}{\partial w_2}, \cdots, \frac{\partial f(\vec{w})}{\partial w_d}\right)^T$$

Where $d$ is the dimension of the weight vector

General SGD for State-Value Prediction:

$$\vec{w}_{t+1} \equiv \vec{w}_t + \alpha\left[U_t - \hat{v}(S_t, \vec{w}_t)\right]\nabla\hat{v}(S_t, \vec{w}_t)$$

Where $U_t$ (the ouput of the training example) is an estimate of the true value instead of the true value (possibly due to noise or bootstrapping)

Note that the true value is by definition the expected value of the return following that state

If $U_t$ is an unbiased estimate then $\vec{w}_t$ is still guaranteed to converge under the stochastic approximation conditions

Note that MC methods provide one such unbiased estimate

Semi-Gradient Methods:

Gradient descent methods that rely on bootstrapping

These aren't true gradient descent methods since they take into account the effect of changing the weight vector on the estimate, but not the target, meaning they include only a part of the gradient

To see this note that going from the first form of the SGD expression above to the second form involves assuming that the gradient doesn't depend on $v_\pi(S_t)$ (i.e $\nabla v_\pi(S_t) = 0$)

Note that these don't converge as robustly as true gradient descent methods, but often enable significantly faster learning and enable learning to be asynchronous

Semi-Gradient TD(0):

$$U_t \equiv R_{t+1} + \gamma\hat{v}(S_{t+1}, \vec{w})$$

State Aggregation:

A form of generalizing function approximation through grouping states together with one estimated value (component of the weight vector) for each group and estimating the value of a state as the value of the component for the group

When the value of a state is updated, only the corresponding component of the weight vector is updated, not the whole weight vector

Note that this is a special case of SGD where $\nabla\hat{v}(S_t, \vec{w}_t)$ is 1 for the group's component and 0 for all other components of the weight vector

Linear Function Approximation:

$$\hat{v} \equiv \vec{w}^T\vec{x}(s) = \sum_{i=1}^{d} w_i x_i(s)$$

Where $\vec{x}(s)$ is a feature vector representing the state with the same dimension as the weight vector

The feature vector consists of $d$ features, which are functions mapping from state space to the real numbers

For linear methods the features are basis functions forming a basis for the space of approximate functions

Linear SGD:

$$\vec{w}_{t+1} \equiv \vec{w}_t + \alpha\left[U_t - \hat{v}(S_t, \vec{w}_t)\right]\vec{x}(S_t)$$

Note that this uses that $\nabla\hat{v}(s, \vec{w}) = \vec{x}(s)$ for linear function approximation

In the linear case there is only one optimum (or a set of equally good optima) so any method that converges to a local optimum converges to the global optimum (assuming $\alpha$ decreases over time appropriately)

Linear Semi-Gradient TD(0):

$$\vec{w}_{t+1} \equiv \vec{w} + \alpha(R_{t+1} + \gamma\vec{w}_t^T x_{t+1} - \vec{w}_t^T\vec{x}_t)\vec{x}_t = \vec{w}_t + \alpha\left(R_{t+1}\vec{x}_t + \vec{x}_t(\vec{x}_t - \gamma\vec{x}_{t+1})^T\vec{w}\right)$$

Where $x_t$ is shorthand for $x(S_t)$

This converges to the TD fixed point

In Steady State:

$E[\vec{w}_{t+1}|\vec{w}_t] = \vec{w}_t + \alpha\left(\vec{b} - \vec{A}\vec{w}_t\right)$

Where $\vec{A} \equiv E\left[\vec{x}_t\left(\vec{x}_t - \gamma\vec{x}_{t+1}\right)^T\right]$, $\vec{b} \equiv E[R_{t+1}\vec{x}_t]$

The algorithm will converge when $\vec{A}$ is positive definite ($\vec{y}^T\vec{A}\vec{y} > 0$ for any real vector $\vec{y} \neq 0$)

TD Fixed Point:

$\vec{w}_{TD} \equiv \vec{A}^{-1}\vec{b}$

Note that this point is not the local optimum but a point near the local optimum

This comes from substituting in the semi-gradient $U_t$ into the linear SGD update expression and finding the condition such that the expected update is 0 (defining $\vec{A}$ and $\vec{b}$ for convenience along the way)

TD Fixed Point Error:

$\overline{VE}(\vec{w}_{TD}) \leq \frac{1}{1-\gamma}\min_{\vec{w}}\overline{VE}(\vec{w})$

This says that the error at the fixed point is no more than $\frac{1}{1-\gamma}$ times the smallest possible error (that obtained in the MC limit)

Note that this has only been proven in the continuous limit

$n$-Step Semi-Gradient TD:

$\vec{w}_{t+n} \equiv \vec{w}_{t+n+1} + \alpha[G_{t:t+n} - \hat{v}(S_t, \vec{w}_{t+n-1})]\nabla\hat{v}(S_t\vec{w}_{t+n-1})$

Generalized $n$-Step Return:

$G_{t:t+n} \equiv \sum_{k=0}^{n-1}\gamma^k R_{t+k+1} + \gamma^n\hat{v}(S_{t+n}, \vec{w}_{t+n-1})$

$= R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1}R_{t+n} + \gamma^n\hat{v}(S_{t+n}, \vec{w}_{t+n-1})$

Where $n \geq 1$ and $0 \leq t \leq T - n$

Choosing features appropriate to the task is a way to add prior domain knowledge to the reinforcement learning system

However, note that linear systems can't take into account interactions between features (e.g. feature $x$ being present is good only if feature $y$ is also present)

This can be fixed with features composed of combinations of other features

Polynomial Basis Features:

$x_i(s) = \prod_{j=1}^{k} s_j^{c_{i,j}}$

Where $k$ is the dimension of the state and each $c_{i,j}$ is an integer in the set $\{0,1,\cdots n\}$

These states form a complete polynomial basis for dimension $k$

Note that there are $(n+1)^k$ different features

One-Dimensional Fourier Cosine Basis Features:

$x_i(s) = \cos(i\pi s)$

With the definition that $x_0$ is a constant function

Note that these can always be used if we restrict $s \in [0,1]$

Multi-Dimensional Fourier Cosine Basis Features:

$x_i(s) = \cos(\pi\vec{s}^T\vec{c}^i)$

Where each state corresponds to a vector of $k$ numbers (i.e. $\vec{s} = (s_1, s_2, \cdots, s_k)$) with the restriction $s_i \in [0,1]$ and $\vec{c}^i = (c_1^i, c_2^i, \cdots, c_k^i)$ with $c_k^i \in \{0, \cdots, n\}$ for $j = 1, \cdots, k$ and $i = 1, \cdots, (n+1)^k$

Note that again there are $(n+1)^k$ different features

When using cosine features with learning algorithms it may be helpful to use a different step-size parameter for each feature

One suggestion based on a basic step-size parameter $\alpha$:

$$\frac{\alpha}{\sqrt{\left(c_1^i\right)^2 + \cdots + \left(c_k^i\right)^2}}$$

With the definition that if each $c_j^i = 0$ then $\alpha_i = 0$

These features tend to have problems with discontinuities (specifically "ringing" around discontinuities)

Binary Coding:

A feature that can only take the values 0 or 1

This can be used to represent some potential feature being present or not present

Coarse Coding:

Representing a state with overlapping features

The name comes from these features coarsely coding for the location of the state in state-space

Tile Coding:

Coarse coding with the receptive fields grouped into partitions of the state space (e.g. a grid)

Multiple tilings are used, each offset by a fraction of a tile width

This may be the most practical feature representation for modern computers

Note that tile coding with just one tiling is state aggregation

Note that exactly one feature is present in each tiling so the total number of features present is the same as the number of tilings

Choosing $\alpha = \frac{1}{n}$ where $n$ is the number of tilings results in one-trial learning (the estimate will be correct after one training example)

A constant can be multiplied with $n$ to get training slower than this

Asymmetrically offset tilings are best since they avoid an effect along the diagonal that occurs with symmetric tilings

One recommendation is to use an offset displacement vector of the first odd integers (e.g. first four tilings start at $(0,0,0)$, $(1,3,5)$, $(2,6,10)$, and $(3,9,15)$)

Note that $\frac{w}{n}$ where $w$ is the width of the tiling and $n$ is the number of tiles, is a fundamental unit in that if a state moves this distance in the direction of any basis vector the feature representation will change by one component per tile

Note that the size, shape, and orientation of the tiles are arbitrary, even weird things like a log scale or fractals can be used

Hashing can also be used as a form of tiling, which has memory benefits

Radial Basis Features (RBF):

Course coding but with each feature being anywhere in the range $[0,1]$

Gaussian RBF:

$$x_i(s) \equiv e^{-\frac{\|s-c_i\|^2}{2\sigma_i^2}}$$

Where $\sigma_i$ is the feature's width and $c_i$ is the feature's prototypical or center state (with the norm or distance metric chosen appropriately for the feature)

Using these basis functions for tile coding results in much more computational complexity and potentially reduced performance

Tabular Method Step-Size:

Setting $\alpha = \frac{1}{\tau}$ means that it takes around $\tau$ experiences for the estimate to approach the mean of the experience targets

Linear SGD Step-Size Rule of Thumb:

$$\alpha \equiv \left(\tau E\left[\vec{x}^T \vec{x}\right]\right)^{-1}$$

Where $\vec{x}$ is a random feature vector chosen from the same distribution as input vectors will be in the SGD

Training the hidden layers of a neural network can be thought of as automatically creating features appropriate to the given problem

Deep neural networks often have the problem of partial derivatives either growing or decaying rapidly

towards the input side of the network
Overfitting is especially problematic for deep neural networks
Batch Normalization:
Normalizing the output of deep layers before they feed into the next layer
Residual Learning:
Learning how a function differs from the identity function rather than learning the function itself
In practice this is done by adding the input of a layer to the output
Least-Squares TD:

$$\vec{w}_t \equiv \hat{\hat{A}}_t^{-1}\hat{\vec{b}}_t$$

$$\hat{\hat{A}}_t \equiv \sum_{k=0}^{t-1} \vec{x}_k\left(\vec{x}_k - \gamma\vec{x}_{k+1}\right)^T + \varepsilon\vec{I}$$

$$\hat{\vec{b}}_t \equiv \sum_{k=0}^{t-1} R_{k+1}\vec{x}_k$$

Where $\vec{w}_t$ is an estimate of the TD fixed point, $\vec{I}$ is the identity matrix, and $\varepsilon$ is some small constant
This is the most data efficient form of linear TD(0), but is expensive computationally
Sherman-Morrison Formula (Incremental Inverse Update):

$$\hat{\hat{A}}_t^{-1} = \left(\hat{\hat{A}}_{t-1} + \vec{x}_{t-1}\left(\vec{x}_{t-1} - \gamma\vec{x}_t\right)^T\right)^{-1} = \hat{\hat{A}}_{t-1}^{-1} - \frac{\hat{\hat{A}}_{t-1}^{-1}\vec{x}_{t-1}\left(\vec{x}_{t-1} - \gamma\vec{x}_t\right)^T\hat{\hat{A}}_{t-1}^{-1}}{1 + \left(\vec{x}_{t-1} - \gamma\vec{x}_t\right)^T\hat{\hat{A}}(t-1)^{-1}\vec{x}_{t-1}}$$

With the definition that $\hat{\hat{A}}_0 \equiv \varepsilon\vec{I}$
This applies to matrices that are a sum of outer products
Note that this algorithm will never "forget" and is $O\left(d^2\right)$ to compute
It also doesn't require a step-size parameter, but does require $\varepsilon$
Memory-Based Function Approximation:
Function approximation methods that save training examples in memory and compute a value estimate for a state only when needed
This is also called lazy learning
These are nonparametric methods since they don't require a parametrized form of the function they are approximating (although they do require some method of combining training examples into a value estimate)
These include nearest neighbor methods
Advantages:
They can focus functional approximation on local neighborhoods of states visited in real or simulated trajectories, global approximation may be unnecessary
They allow experience to have an immediate effect on value estimates without being watered down by trying to move a global function approximation
Speed can be an issue for these methods as they generally have to search over all examples stored in memory, although intelligent data structure design can help with this (specifically $k$-d trees)
Local Learning:
Methods that approximate a value function only locally in the neighborhood of the current query state
Weighted Average Methods ($k$-nearest neighbors):
Methods that approximate values with a weighted average of some number ($k$) of the nearest neighbors to the current state (weighted by relevance, usually distance from the current state)
Locally Weighted Regression Methods:
Methods that fit a surface to the values of a set of nearest states by means of a parametric approximation method, returning the value found by evaluating the surface at the current query state
Kernel Function:

The function that assigns a relevance score to pairs of states

Relevance can be viewed as the strength of generalization between the two states

Kernel Regression:

$$\hat{v}(s, D) = \sum_{s' \in D} k(s, s') g(s')$$

Where $D$ is the set of stored examples, $k$ is the kernel function and $g$ is the target for the example being considered

A common kernel is the Gaussian Radial Basis Function above

Any linear parametric regression method can be recast as kernel regression with the kernel function being the inner product between the feature vectors of the states being compared

This is sometimes called the "kernel trick" since it allows the feature vectors to be ignored and to effectively work in a high dimensional feature space while only actually working with stored examples

Interest:

$I_t$

A non-negative scalar measure of the degree to which we are interested in accurately valuing the state or state-action pair at time $t$

This is applied by weighting $\mu$

Emphasis:

$M_t$

A non-negative scalar that multiplies the learning rate and therefore emphasizes or de-emphasizes the learning done at time $t$

$n$-Step Learning with Emphasis:

$$\vec{w}_{t+n} \equiv \vec{w}_{t+n-1} + \alpha M_t \left[ G_{t:t+n} - \hat{v}\left(S_t, \vec{w}_{t+n-1}\right) \right] \nabla \hat{v}\left(S_t, \vec{w}_{t+n-1}\right)$$

Relation Between Interest and Emphasis:

$$M_t = I_t + \gamma^n M_{t-n}$$

With the definition $M_t \equiv 0$