On the Numeric Stability of Gaussian Processes Regression for Relational Reinforcement Learning

Jan Ramon
Kurt Driessens
Department of Computer Science, K.U.Leuven, Celestijnenlaan 200A, B-3001 Leuven, Belgium

Abstract

In this work we investigate the behavior of Gaussian processes as a regression technique for reinforcement learning. When confronted with too many “mutually dependent” learning examples, the matrix inversion needed for prediction of a new target value becomes numerically unstable. More needed...

1. Introduction

The RRL system (Dzeroski et al., 2001) uses a combination of a standard Q-learning algorithm and a relational regression algorithm to make reinforcement learning possible in relational domains. Through the use of relational representation for states, actions and the eventual learned Q-function, RRL makes use of structural information about the learning environment and can be applied to larger problems than standard Q-learning approaches.

The RRL system makes use of an incremental and relational regression algorithm to build Q-function generalization. Three different regression algorithms have been used in this context so far: an incremental regression tree algorithm called RQ (Driessens et al., 2001), a relational instance based algorithm RIB (Driessens & Ramon, 2003) and a regression algorithm based on Gaussian Processes and kernels for structural domains called RNR (Gärtner et al., 2003).

So far however, although the RQ algorithm has been successfully applied to the Digger computer game, these algorithms have mainly been tested on different tasks in the Blocks World. In this work, we investigate the performance of the RNR regression algorithm using the Tetris game as a driving force.

2. The Tetris Game

Tetris\(^1\) is probably one of the most famous computer games around. Designed by Alexey Pajitnov in 1985 and has been ported to almost any platform available, including most consoles. We therefore consider the dynamics and rules of the Tetris game known to the reader.

We will only consider the task of deciding the placement of the next block. Given the shape of the dropping block, one has to decide on the optimal orientation and location of the block in the game field. This can be seen as the strategic part of the game and deals with the uncertainty about the shape of the blocks that will follow the present one.

There exist very good artificial Tetris players, most of which are hand built. The best of these algorithms score about 500,000 lines on average when they only include information about the falling block and more than 5 million lines when the next block is also considered. The results discussed here will be nowhere near this high and will be even low for human standards. However, the experiments shown will illustrate the capabilities of and the difficulties still faced by the RRL system, and possibly of Q-learning algorithms in general.

The stochastic nature of the game (i.e. the unknown shape of the next falling block) in combination with the chaotic nature of the Tetris dynamics make it very hard to connect a Q-value to a given \((\text{state}, \text{action})\) pair. It is very hard to predict the future rewards starting from a given Tetris state. The state that can quickly lead to a reward for a given block, can be completely unsuited to deal with other blocks. Also, for a given state, two different actions can lead to completely different resulting states (for example, creating a deep canyon in one case and deleting 3 lines in the other).

\(^1\)Tetris is owned by The Tetris Company and Blue Planet Software.
One feature that is of specific importance for this work is that the beginstate for each learning episode is identical (except for maybe the falling block). Still, all states during the first few turns of the Tetris game are very similar in subsequent games. This will cause these states to be visited quite frequently.

3. The KBR regression algorithm

The KBR algorithm applies Gaussian processes to predict the Q-values of unknown (state, action) pairs. Gaussian processes are a non-parametric Bayesian regression technique. Where most (parametric) Bayesian regression techniques assume a prior distribution over the parameter vector and calculate a posterior distribution over parameter vectors using Bayes rule and the available learning data, Gaussian processes instead assume a prior assumed over the target function itself.

Given a set of data points \([\{x_i, t_i\}]_{i=1}^{N}\), with \(x_i\) the description of the example and \(t_i\) the target value, Gaussian processes assume the target values \(t_N = [t_1 \cdots t_N]\) to have a joint distribution (Gibbs, 1997):

\[
P(t_N|\{x_1 \cdots x_N\}, C_N) = \frac{1}{Z} \exp \left( -\frac{1}{2} (t_N - \mu)^T C_N^{-1} (t_N - \mu) \right)
\]

where \(\mu\) is the mean vector of the target values, \(C\) is a covariance matrix \((C_{ij} = C(x_i, x_j), 1 \leq i, j \leq N)\) and \(Z\) is an appropriate normalization constant. For simplicity reasons\(^2\), we assume the mean vector \(\mu = 0\).

The regression task in a Bayesian approach is to find the predictive distribution of the value \(t_{N+1}\) given the example description \(x_{N+1}\). I.e.:

\[
P(t_{N+1}|\{x_1 \cdots x_N\}, \{t_1 \cdots t_N\}, x_{N+1})
\]

Using the assumption made when using Gaussian processes, this distribution can be written as:

\[
P(t_{N+1}|\{x_1 \cdots x_N\}, t_N; x_{N+1}, C_{N+1}) = \frac{Z_{N+1}}{Z_N} \exp \left( -\frac{1}{2} (t_{N+1} C_{N+1}^{-1} t_{N+1} - t_N C_N^{-1} t_N) \right)
\]

with \(Z_N\) and \(Z_{N+1}\) appropriate normalizing constants and \(C_N\) and \(C_{N+1}\) as in Figure 1. The vector \(k_{N+1}\) and scalar \(\kappa\) are defined as:

\[
k_{N+1} = [C(x_1, x_{N+1}) \cdots C(x_N, x_{N+1})]
\]

\[
\kappa = C(x_{N+1}, x_{N+1})
\]

\(^2\)Although this may seem as a leap of faith, assuming 0 as an apriori Q-value is standard practice in Q-learning.

\[
C_{N+1} = \begin{pmatrix}
C_N & k_{N+1} \\
\kappa & \kappa
\end{pmatrix}
\]

Figure 1. The relationship between the covariance matrices \(C_N\) and \(C_{N+1}\).

By grouping the terms that depend on \(t_{N+1}\) (Gibbs, 1997) Equation 1 can be rewritten as:

\[
P(t_{N+1}|x_1 \cdots x_N, t_N, x_{N+1}, C_{N+1}) = \frac{1}{Z} \exp \left( -\frac{(t_{N+1} - \hat{t}_{N+1})^2}{2\sigma_{\hat{t}_{N+1}}^2} \right)
\]

with

\[
\hat{t}_{N+1} = k_{N+1}^T C_N^{-1} t_N \quad (1)
\]

\[
\sigma_{\hat{t}_{N+1}}^2 = \kappa - k_{N+1}^T C_N^{-1} k_{N+1} \quad (2)
\]

and \(k_{N+1}\) and \(\kappa\) as previously defined. This expression maximizes at \(\hat{t}_{N+1}\), and therefore the value \(\hat{t}_{N+1}\) is the one that will be predicted by the regression algorithm. \(\sigma_{\hat{t}_{N+1}}\) gives the standard deviation on the predicted value. Note that, to make predictions, \(C_N^{-1}\) is used, so there is no need to invert the new matrix \(C_{N+1}\) for each prediction.

3.1. The Covariance Function and Kernels

The choice of covariance functions is restricted to positive definite kernel functions. In general, kernel methods work by embedding the data into a vector space and then looking for (often linear) relations between the data in that space. If the mapping to the vector space is well chosen, complex relations can be simplified and more easily discovered. These relations can then be used for classification, regression, etc.

The kernel function is employed to avoid the need for an explicit mapping to the (often high dimensional) vector space. Technically, a kernel \(k\) computes an inner product in some feature space which is, in general, different from the representation space of the instances. The computational attractiveness of kernel methods comes from the fact that quite often a closed form of these ‘feature space inner products’ exists. Instead of performing the expensive transformation step
\( \phi \) explicitly, a kernel \( k(x, x') = \langle \phi(x), \phi(x') \rangle \) computes the inner product directly and performs the feature transformation only implicitly.

For a given function \( k: \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R} \), a feature transformation \( \phi: \mathcal{X} \rightarrow \mathcal{H} \) into some Hilbert space \( \mathcal{H} \) such that \( k(x, x') = \langle \phi(x), \phi(x') \rangle \) for all \( x, x' \in \mathcal{X} \) exists, if the function is positive definite (Aronszajn 1950).

4. Numeric Instability

To predict the value of a new example, the covariance matrix \( C \) needs to be inverted. In the past, we have computed this inverse incrementally, using the partitioned inverse equations of (Barnett 1979):

\[
C^{-1}_{N+1} = \begin{bmatrix} M & m \\ m^T & \mu \end{bmatrix}
\]

with

\[
M = C^{-1}_N - \mu k_{N+1} k_{N+1}^T \\
m = -\mu C^{-1}_N k_{N+1} \\
\mu = (k - k_{N+1}^T C^{-1}_N k_{N+1})^{-1}
\]

While matrix inversion is of cubic complexity, computing the inverse of a matrix incrementally after expansion is only of quadratic time complexity. As stated before, no additional inversions need to be performed to make multiple predictions.

However, this inversion can become numerically unstable, and actually even impossible, as there is no guarantee that the covariance matrix is non-singular. This is in particular a problem when many examples are in a low-dimensional sub-space of the feature space induced by the kernel, which is likely in a reinforcement learning context as the same states are often visited repeatedly, especially when a lot of training episodes are allowed. In the Tetris application, the feature space has a rather low dimension and hence the problem is unavoidable.

4.1. Possible Solutions

A first solution is to observe that the covariance matrix of the examples is positive semi-definite (which follows from the fact that its components are values of the kernel \( k \)). Hence one can make it of full rank by adding a multiple of the identity matrix to it. This means we use \( C + \epsilon I \) in the computations instead of \( C \).\footnote{This solution was applied in previous applications of the kern system, mainly in the blocks world} Unfortunately, this is not always sufficient: if \( \epsilon \) is large, the matrix \( C + \epsilon I \) differs too much from the real covariance matrix \( C \) while if \( \epsilon \) is small, computing \( (C + \epsilon I)^{-1} \) gets numerically very unstable.

Therefore, we should treat the numerical instability problems more fundamentally. Let us first see where \( C^{-1} \) is needed. Let \( x_1 = \phi(e_1) \) be the representation of example \( e_1 \) in feature space. We want to minimize the error \( E = \sum_i \| x_i^T w - y_i \|^2 \), where \( w \) is the normal vector to the regression plane in feature space. We can express this as \( E = (w^T X^T - Y^T)(X w - Y) \) where \( X \) is a matrix containing the vectors \( x_i \) as rows and \( Y \) is a vector containing the \( y_i \). Setting \( \nabla E = 0 \) we get \( X^T X w - X^T Y = 0 \) or \( w = (X^T X)^{-1} X^T Y = X^T (X^T X)^{-1} Y = X^T C^{-1} Y \).

We first discuss a naive attempt. We determined for every example whether it is in feature space a multiple of an already stored example (by checking \( k(e_{new}, e_i) = \sqrt{k(e_i, e_i) \cdot k(e_{new}, e_{new})} \)). If it is a multiple, we adapt the value of the already stored example using a linear regression technique and do not store the new example. If the new example is not a multiple of a previous example, we just store it as before. Our experiments show that this already greatly improves the numerical stability for Tetris; the algorithm now quickly learns to delete around 40 lines, where the original KRR algorithm only achieved around 5. Still, after a while the system un-learns this good behavior as an increasing number of examples which are linear combinations of (more than one) other examples in feature space still cause numerical problems. We note that increasing the floating-point precision in our implementation barely helps.

A better approach is to handle the problem is to use a QR-factorization of the examples. This allows for a numerically stable algorithm while still using all available information from the examples. Let \( X = QR \) where \( X \) is again a matrix with the \( x_i \) as rows, \( R \) is a square, non-singular upper triangular matrix and \( Q \) is an orthonormal matrix \( (Q^T Q = I) \), so we get \( w = (R^T Q R)^{-1} R^T Q^T Y \) and \( w = R^{-1} Q^T Y \) or

\[
R w = Q^T Y,
\]

from which \( w \) can be easily computed with back substitution. The problem is now to do this implicitly using a kernel as we do not know the vectors \( x_i \) (the representations of the examples in feature space). One can do this by using a set of linearly independent examples as a base. In particular, we keep in memory the matrix \( X \) representing the examples as rows \( x_i = (x_{i1}, \ldots, x_{ik}) \) and a matrix \( U \) representing our orthonormal base vectors \( u_i \) with

\[
u_{ij} = \sum_{j=0}^{i} u_{ij} e_{ij}
\]
where the $b_i$ are indices of examples. We update these two matrices incrementally. When receiving a new example $e_n$, we first decompose it in terms of the base set $\{u_1, \ldots, u_k\}$ where $k \leq n - 1$ is the dimension of the base: $x_{ni} = k(e_n, u_i)$. This kernel value can be computed using (4): $k(e_n, u_i) = \sum_{j=0}^{i} u_{ij}^2$. In a second step we determine whether this example is linearly dependent on the previous examples. Therefore, we compute the distance from the current example to the hyperplane formed by the base vectors $u_i$, which is

$$k(e_n, e_n) - \sum_{i=0}^{k} x_{ni}^2.$$  

If this value is larger than some threshold, we choose $e_n$ as a new base vector:

$$u_{k+1} = \frac{e_n - \sum_{i=0}^{k} x_{ni} u_i}{\sqrt{k(e_n, e_n) - \sum_{i=0}^{k} x_{ni}^2}}$$

$b_{k+1}$ is set to $n$, $k$ is incremented by one, and all vectors $x_i$ get an extra coordinate 0 (except for $e_n$), such that the set of base vectors has a new element.

So now we have at every stage an explicit representation of our examples, stored in a $n \times k$ matrix $X$ where $k$ may be smaller than $n$. In order to be able to use (3) one should write this $X$ in the form $X = QR$. After computing $X$, one can incrementally update $Q$ and $R$ in time $O(nk)$ (see Golub & Van Loan, 1996) for a more detailed explanation. This can be seen as follows

$$[X \quad x_{n+1}] = [Q \quad 0] [0 \quad 1] [R \quad 0]$$

to make the extended R-matrix upper-triangular again, $k$ Givens rotations are required to zero the $k$ first elements of the bottom line of this matrix. The extended $Q$ matrix should be transformed accordingly.

After computing $w$ using (3), our prediction for a new example $e$ can be computed by decomposing $e$ w.r.t. the base vectors $u_i$ and by computing $w \cdot x$.

In summary, we have described a way to update our kernel-based theory avoiding both matrix inversion and changing the data. Note that our method has complexity $O(nk)$, so when $k \ll n$ (most examples are (approximately) linearly dependent in feature space) this method is also faster than the previously used method.

5. Experiments

Experimental results are at present still quite unstable for the discussed reasons. However, preliminary results show that RRL very quickly learns to delete around 40 lines. While this is still very low compared to other approaches, it is a lot better than previously reported results of the RRL-TG system on the Tetris Game.

In the figure, average over 10 test runs.

As one can see:

- QR is better
- std starts with learning but un-learns this very fast due to numerical problems.

In fact, for other kernels with more features, std sometimes performs well for a few more episodes.

6. Conclusions

In relational reinforcement learning, examples are often heterogeneously spread in feature space, making our original kernel based regression algorithm numerically unstable due to the problem of inverting a singular covariance matrix. Paying attention to using a suitable numerical technique improves experimental results significantly.

Acknowledgements

Jan Ramon is a post-doctoral fellow of the Fund for Scientific Research (FWO) of Flanders.

References


