Q-learning using state and action space reduction transformations

CAP 6671 - Final Project

Anthony Wertz
Engineering and Computer Science
University of Central Florida
4000 Central Florida Blvd
Orlando, FL
awertz@knights.ucf.edu

ABSTRACT

Without using modified methods, the vanilla Q-learning algorithm is not capable of handling policy construction for even simple problems as soon as the state space becomes too large. This paper introduces a method of Q-learning policy generation for large state spaces and dynamic environments by proposing a simple state space reduction transformation model. The algorithm is based on the unmodified Q-learning implementation but designed such that it can seamlessly benefit from many already existing Q-learning improvements.

1. INTRODUCTION

Q-learning is a useful method for learning policies. However, without using modified versions of the algorithm, the state space for real-world problems is typically much too large for Q-learning to handle. This may require practitioners to use extensively simplified models potentially yielding less than optimal policies or making the problem unsolvable (for problems that require more fine granularity). One may be forced to use another learning method entirely.

This paper hopes to combat this issue by proposing a simple model by which a large Q-learning problem can be decomposed into orders of magnitude smaller Q-learning problems with little added overhead. This method also shows how the transformation functions used to decompose the problem may also be used to mask some elements of a dynamic environment so that generic policies may be learned and applied in many situations. This may also prove to be a useful model for transfer learning, as learned policies may be shared by other systems where an appropriate transformation function may be designed or learned.

2. RELATED WORK

This method is based on the basic Q-learning algorithm, an iterative global policy search using reinforcement learning. It's described in more detail in [2]. The method attempts to find an optimal policy for every state and action pair which quickly becomes intractable for problems with large state spaces. In most cases, without large alterations to the Q-learning operation it becomes unusable for learning policies in large problems and instead is relegated to use in small subproblems, as in [1] where Q-learning is used in small groups of soccer playing agents to learn simple skills. However, this marginalizes the usefulness of Q-learning. This work hopes to correct that issue.

3. METHODOLOGY

3.1 State and Action Space Transformations

Given the state space S and the action set A, basic Q-learning would require enough space to store policies for all possible states and actions, the set $S \times A$ which can require significant space to store and also significant time to calculate the Q-table. However, we can define a set of transformations to reduce the size of the state space to generalize the problem, allowing the use of a number of much smaller Q-tables to solve the same problem. The transformations can be represented as

$$tr(x; S, A) \rightarrow x_t; S_t, A_t \tag{1}$$

A good transformation function should be defined in a way that enables it to use relative measures in the state space instead of absolutes. Thus, it does not have to represent the state space in its entirety but can instead generalize a specific scenario that may occur anywhere. The dimensionality of S_t and A_t are, ideally, much less than those of S and A.

We can represent some number n of transformations using the matrix notation

$$Tr(x; S, A) = \begin{bmatrix} tr_1(x; S, A) \\ tr_2(x; S, A) \\ \vdots \\ tr_n(x; S, A) \end{bmatrix} \tag{2}$$

For each transformation the vanilla Q-learning algorithm is applied to the reduced state and action space. The Q-values can be represented for each transformation as

$$Q(x;S,A) = \begin{bmatrix} q_1(tr_1(x);S_{t1},A_{t2}) \\ q_2(tr_2(x);S_{t2},A_{t2}) \\ \vdots \\ q_n(tr_n(x);S_{tn},A_{tn}) \end{bmatrix} \quad (3)$$

For each Q-table we also wish to provide some tunable model parameters, W:

$$W = \begin{bmatrix} w_1 \\ w_2 \\ \vdots \\ w_n \end{bmatrix} \quad (4)$$

To determine at any given state which transformation to use we can calculate the maximum scaled Q-value in the given state:

$$i = \arg\max_i (w_i q_i(x;S,A)) \quad (5)$$

The transformation function alone is not enough to use Q-learning to find an optimal policy for a reduced state since, in general, the world model does not know how to execute actions in the reduced state. One simple option to combat this issue is to redefine the action handlers for use in the reduced state space. This is fairly simple and intuitive but it requires repeating work and may open room for inconsistencies between the models and bugs. Another option is to define a small training world (which must be done anyway) and a training world specific inverse transformation function which will allow the use of the original world model's action handling routines. For purposes of demonstration, this is the method chosen in this paper.

In general, a good transformation function will lose information, ideally a lot of it, so a one to one inverse transformation function is usually impossible. However, given a cleverly crafted training world representation, a training world specific inverse transition function, represented as

$$tr^{-1}(x_t;S_t,A_t) \rightarrow x*;S,A \quad (6)$$

can be designed, where $x*$ represents the full state vector generated by adding some scenario-specific information to the reduced state vector x_t. Thus, it is not relevant to general problems but it is designed to be sufficient for the purpose of learning the transformation policy. How this function and the appropriate training world are devised is very application specific and requires some expertise on the part of the designer. Some examples are given later on in this paper.

3.2 A toy world example

Now a simple deterministic world will be described which will be used to build a model as discussed in the previous section. There exists an agent who operates in a grid world with dimensions W cells wide by H cells high. Each cell can be either unusable or one of two useful sites: a mineral deposit or ground with good soil. The agent's objective is to collect enough mineral resources and bamboo to construct arrows. This is complicated by the fact that at each site the agent must perform some number of tasks before the resources can be collected. At the mineral deposit, the agent must first dig into the ground, then separate the minerals from other, less useful materials, and finally extract them from the site. At the sites with good soil, the land must be prepared for planting by tilling. Then, bamboo must be planted and some time must pass before bamboo will be ready to harvest. Once it is, the bamboo needs to be extracted before it's ready for use. The agent needs to move around the map collecting resources until he has enough to build his arrows. Once he has enough materials, he will build his arrows.

This world presents some problems for the standard Q-learning implementation: First, the world is dynamic, as extracting resources modifies the state of the given cell. This means policies would have to be learned either on the fly, which is probably much too slow depending on the application, or it must be learned for all or at least most of the likely scenarios, which may be too costly in a world with a large state space. Second, even if speed is not a concern, the memory required to store and compute a policy for such a large state space may be impractical.

The state vector for this world can be described as

$$s = \begin{pmatrix} site[W][H] \\ x \\ y \\ mineral \\ bamboo \\ arrows \end{pmatrix} \quad (7)$$

Where x and y are valid integers within the bounds of the grid world ($x \in [0,W)$, $y \in [0,H)$), *mineral* and *bamboo* are positive integers representing the quantity of mineral and bamboo resources (respectively), and *site* is an array with each element representing the state of the given grid world position which can be one of the elements in the set

$$\forall x_i \in [0..W), y_i \in [0..H) : \\ site_{x_i,y_i} \in \begin{pmatrix} useless, & mineral\_deposit, \\ minerals\_exposed, & minerals\_separated, \\ good\_soil, & tilled, \\ planted, & sprouted \end{pmatrix} \quad (8)$$

The *arrows* element represents a state vector annotating the current step in the arrow making process. It may be one of the following values:

$$arrows \in \begin{pmatrix} not\_started, \\ tips\_formed, \\ fins\_formed, \\ shafts\_formed, \\ arrows\_complete \end{pmatrix} \quad (9)$$

If the *mineral* and *bamboo* elements were assumed to be in the range of $[0, needed\_minerals]$ and $[0, needed\_bamboo]$ respectively, the size of the state space is effectively

$$8^{WH} 5WH(needed\_minerals+1)(needed\_bamboo+1) \quad (10)$$

This becomes very large very quickly as the size of the grid world increases. If the agent needs 99 mineral and bamboo

elements each in a ten by ten grid world, the state space size is 1×10^{97}. Using a Q-table to represent this state space would add the action dimension, making the total number of states to explore quite large for this very simple example. In fact, it's entirely impossible to make this calculation based on the spacial requirements alone. It is also highly inefficient to calculate policies for this entire state space even if it were possible since the usefulness of the actions is quite sparse (for example, you would only ever try to perform the action "dig" on top of a mineral deposit, so if there is one deposit in the entire grid then in 99 of 100 grid cells you would be calculating Q-table state values that you know don't matter).

Because of this, Q-learning is not even applicable to this problem. Another method would have to be selected to overcome these huge constraints. However, it is possible to get around these hurdles by using the model proposed earlier. In this example, four transition functions will be defined: one for world motion, one for handling mineral deposits, one for handling good soil, and one for building arrows. Effectively the problem will be reduced to a combination of four smaller Q-learning problems with much smaller state spaces.

3.2.1 Movement Transformation Function

The first transition function should deal with agent movement around the map. Without knowing in advance the exact map state, a transition function $tr_M(x; S, A)$ can be designed to reduce this model to a relative state space and action subset defined as

$$A_M = \begin{pmatrix} move\_left \\ move\_up \\ move\_right \\ move\_down \end{pmatrix} \tag{11}$$

$$S_M = \begin{pmatrix} rel_x \in (left\_of, at, right\_of) \\ rel_y \in (above, at, below) \end{pmatrix} \tag{12}$$

The transformation itself can be described as follows:

$$x_t = \begin{pmatrix} rel_x(x - x_r) \\ rel_y(y - y_r) \end{pmatrix} \tag{13}$$

$$rel_x(\Delta x) = \begin{cases} at & \text{if } \Delta x = 0; \\ left\_of & \text{if } \Delta x < 0; \\ right\_of & \text{if } \Delta x > 0; \end{cases} \tag{14}$$

$$rel_y(\Delta y) = \begin{cases} at & \text{if } \Delta y = 0; \\ above & \text{if } \Delta y < 0; \\ below & \text{if } \Delta y > 0; \end{cases} \tag{15}$$

x_r and y_r can be chosen as the resource position parameters that minimize the Euclidean distance to the agent location.

$$x_r, y_r = \arg \min_{x_r, y_r \in \Psi} \left(\sqrt{(x - x_r)^2 + (y - y_r)^2} \right) \tag{16}$$

The goal of this function is to determine where the agent is relative to the closest useful resource. Only the resource positions with useful resources are evaluated. This set is represented by Ψ which can be generated using the following criteria:

$$\psi(s, x_i, y_i) = \begin{cases} true & \text{if } s.site_{x_i, y_i} \in (mineral*) \\ & \text{and } s.minerals < needed\_mineral; \\ true & \text{if } s.site_{x_i, y_i} \in (good\_soil, tilled, sprouted) \\ & \text{and } s.bamboo < needed\_bamboo; \\ false & \text{otherwise;} \end{cases} \tag{17}$$

$$\forall x_i \in [0..W), y_i \in [0..H) : \psi(s, x_i, y_i) \rightarrow (x_i, y_i) \in \Psi \tag{18}$$

Effectively the ψ function is just determining whether or not a state contains a useful resource. All sites in a given state that have useful resources are part of the set Ψ.

3.2.2 Mineral Extraction Transformation Function

The mineral extraction transformation function, represented as $tr_{ME}(x; S, A)$, will just look at the site state. The transformation will work within the following state and action space:

$$A_{ME} = \begin{pmatrix} dig \\ separate \\ extract \end{pmatrix} \tag{19}$$

$$S_{ME} = \begin{pmatrix} site \in \begin{pmatrix} useless, \\ mineral\_deposit, \\ minerals\_exposed, \\ minerals\_separated \end{pmatrix} \end{pmatrix} \tag{20}$$

Simply put, any site state not enumerated in S_{ME} is automatically mapped to the *useless* state. Also, if minerals are not needed, then any state will be mapped to *useless* in this function.

3.2.3 Bamboo Extraction Transformation Function

The bamboo extraction transformation function, represented as $tr_{BE}(x; S, A)$, will just look at the site state. It's almost identical to the mineral extraction transformation function. The transformation will work within the following state and action space:

$$A_{BE} = \begin{pmatrix} till \\ plant \\ harvest \end{pmatrix} \tag{21}$$

$$S_{BE} = \begin{pmatrix} site \in (useless, good\_soil, tilled, sprouted) \end{pmatrix} \tag{22}$$

Like the mineral extraction transformation function, any state not represented in S_{BE} is mapped to *useless*, as is any state if bamboo is not not needed. What's noteworthy is that the *planted* state is not represented here so it's mapped to the *useless* state. This will allow the agent to move on and conduct other useful tasks while the bamboo is growing.

3.2.4 Arrow Construction Transformation Function

The arrow construction transformation function, represented as $tr_{AC}(x; S, A)$, needs to know only whether or not the agent has accumulated enough resources to make his arrows and what the progress of the arrow construction is. The transformation will look at the following state and action space:

$$A_{AC} = \begin{pmatrix} form\_tips \\ form\_fins \\ form\_shafts \\ connect\_parts \end{pmatrix} \tag{23}$$

$$S_{AC} = \begin{pmatrix} arrows \\ enough\_resources \end{pmatrix} \tag{24}$$

The *arrows* state parameter contains all the states of the original *arrows* parameter in the full state space.

4. EVALUATION

In this section the Q-learning transformation method will be evaluated relative to the vanilla Q-learning algorithm. The decision was made to disregard all of the more complicated Q-learning extensions and optimizations because those same alterations can be applied to the subtasks generated by this transform method as well, allowing the altered methods to still benefit from the same complexity reduction. By substituting vanilla Q-learning with a more capable algorithm, the transform method can be given the power to handle more complicated subproblems. Given that, it's more useful to compare the baseline techniques.

Two methods will be used to evaluate the transformation approach: first the time and space complexity of the methods will be compared in general and for the previously described application. It can be shown that given appropriate transformations the complexity of a task can be reduced significantly. After that, the problem discussed in the last section is implemented and used to compare the operations empirically. It is shown here that the unmodified Q-learning algorithm is completely incapable of handling this very simple world because the dynamics increase the size of the state space immensely, making the generation of an entire policy difficult and, in many cases, impossible.

4.1 Complexity Analysis

As equation 10 showed, the space complexity of the vanilla Q-learning task is unrealistically large for this application. In fact, it's not even possible to compute the policy for a small ten by ten grid as mentioned previously. The world is reduced to a much smaller world in the next section for comparison, but for this section I'll continue to focus on the original problem description.

In general, the space complexity for a discrete Q-learning task can be described as proportional to the cross of the state and action space, so $\propto |S| \times |A|$. In general, other terms in the computation are insignificant in comparison. If l_i represents the number of possible values for a given state element and l_a represents the number of actions possible, then the space complexity is the product over all i:

$$O\left(l_a \prod_i l_i\right) \tag{25}$$

Since the transformation operation is really just a number of n independent Q-learning subtasks, the space complexity for the transform Q-learning algorithm is represented as

$$O\left(\sum_n \left[l_{a,n} \prod_i l_{n,i}\right]\right) \tag{26}$$

It's obvious from this that poorly chosen transformations have the potential of making the complexity worse, but as seen in this example it can also make it much better. It's easy to see for the movement transform the space required for a Q-learning task is

$$m_M = 3 * 3 * 4 \tag{27}$$

$$= 36 \tag{28}$$

For the mineral extraction transformation the space required is

$$m_{ME} = 4 * 3 \tag{29}$$

$$= 12 \tag{30}$$

For the bamboo extraction transformation the space required is the same, so $m_{BE} = m_{ME}$. Finally, for the arrow construction task, the space required is

$$m_{AC} = 5 * 2 * 4 \tag{31}$$

$$= 40 \tag{32}$$

Added together, the total space complexity is a mere 100 entries. This is fairly insignificant and is many orders of magnitude smaller than the original problem size of about 1×10^{97}. The latter is impossible to solve with Q-learning, the former is trivial.

In terms of time complexity, the Q-learning implementation is proportional to the size of the state action space, $O(Q()) \propto |S| \times |A|$. This means it is also proportional to the space complexity, so $O_{time}(Q()) \propto O_{space}(Q())$. As such, the time complexity is proportional to the sum of the space complexities for each transformation:

$$O_{time}(Q) \propto O\left(\sum_n \left[l_{a,n} \prod_i l_{n,i}\right]\right) \tag{33}$$

From this it is obvious given the space complexity reduction in this example that there are proportionally large time complexity reductions in the Q-learning transform approach. Again, though, poorly chosen transformations are able to increase the time and space complexities. Additionally, the time and space complexities of the transformation functions themselves must also be considered because poorly designed algorithms can contribute to significant performance degredation.

4.2 Implementation

The toy world described above was implemented using Python 2.7.2, numpy 1.5.1 for some array manipulations, and pygame 1.9.1 for the visual presentation. The work was run on a laptop with an Intel i7 2.3GHz quad-core processor, although it was not parallelized for this comparison, so a single-core assumption could be made.

In order to execute Q-learning on the transformed state and action spaces, a specific training world and corresponding inverse transformation function were devised for each transformation to be used in learning. After describing the training worlds and inverse transformation functions, the Q-learning performance in the test world will be evaluated. For all tests presented here, the Q-learning parameters used

were $\alpha = 0.1$ for the learning rate and $\gamma = 0.6$ for the discount factor. The number of iterations and episodes ran vary from transform to transform. In all cases, a random exploration policy is utilized. The Q-table update function used is represented in equation 34.

$$Q(s,a) = Q(s,a) + \alpha\left(R(s) + \gamma \max Q(s\prime, a\prime) - Q(s,a)\right) \tag{34}$$

4.2.1 Movement Transformation

Around the movement transformation function a very simple world was built. In the movement transform, every agent position can be represented in terms of essentially one of nine different states representing the the relative x and y normalized offset from the closest resource. Therefore, a training grid of three by three squares can be used with one useful resource in the middle. The useful resource is the goal state for the Q-learning task and the movement operators of the world can be used to reach the goal. Essentially, you're learning a movement policy for a very small map. For a real-world problem, optimal path-finding would probably be better served using another more specialized planning algorithm, but this provides a simple example of using a transformation to build a static representation for a dynamic problem (i.e., in the real problem resources may change or disappear, and may be varying distances from the agent).

The inverse transformation function merely maps the normalized offset to an absolute position on the map. In this case there is an easy one to one mapping for the positions. Other parameters of the state vector are not important to the transformation, so they are initialized to null data. The parameters of 300 episodes with a maximum of 50 actions per episode were chosen empirically, as the Q-learning task rarely took more than fifty steps before reaching the goal, and the policy converged relatively quickly, closer to about one hundred episodes. Given the speed of the calculation, 300 was chosen to move marginally closer to the optimal policy reward values. With this configuration, the policy takes about $50ms$ of CPU time to execute on a single $2.3Ghz$ processor.

4.2.2 Mineral Extraction Transformation

The mineral extraction transformation was configured as a single grid world. The goal state is actually set to the *useless* state as that is what the site will transition to after the minerals have been extracted. The inverse transform method then simply sets the reduced world state's site parameter to the full state's site parameter. The other elements are irrelevant as long as they're valid (e.g. the agent position had to be set to $(0,0)$ since there is only one location on the grid). Using the same empirical process, the Q-learning task was set to perform 300 episodes with a maximum of 30 actions per episode. The policy takes about $20ms$ to execute.

4.2.3 Bamboo Extraction Transformation

The bamboo extraction transformation is essentially the same as the for the mineral extraction transformation except in this case there are two goal states: one at the *useless* state (since, again, that is the resulting state after the bamboo is harvested), and one at the *bamboo_planted* state. In the actual problem, there is a time delay between planting the bamboo and it being full-grown, ready for harvest. Modeling time in the Q-learner might confuse the policy generation, especially if the time delay is unknown or varies, so instead two separate goals were defined. This gave good results, as expected. For this case, 250 episodes with 15 actions maximum per episode worked well, finding a solution in around $10ms$. This is quicker than the mineral extraction case since there exist the same number of states but now twice as many goal states.

4.2.4 Arrow Construction Transformation

The arrow construction transformation is essentially the same as the mineral extraction transformation with just one additional state representing whether or not the agent has enough resources to construct the arrows. Again, the training world is configured as a one by one grid and the inverse transformation function essentially just sets the agent's arrow state (the particular site state now does not matter). In addition, the parameter *enough_resources* is mapped over by setting the expanded state's *minerals* and *bamboo* parameters to zero if the reduced state's *enough_resources* parameter is $false$ and they're set to the minimum needed resources, *needed_minerals* and *needed_bamboo* respectively, when *enough_resources* is *true*. This works well enough to learn a good policy. For the arrow construction learning, 300 episodes with a maximum action count of 30 is used and it runs in about $80ms$. Note that the time is higher than for the mineral extraction transformation. In this transformation, if a random state in the reduced state space is chosen such that the *enough_resources* parameter is $false$ there is no valid path to the goal, so every episode will take the maximum number of steps.

4.2.5 Full State Space Transformation

In order to compare the results of the policy generated by learning in the full state space, I did have to make a few concessions. First, I discretized the resource counts to values from [0..150] in increments of 50, the amount given for extracting resources in this implementation (where 150 is the number of both resources needed to build arrows). That is a pretty trivial state space reduction. The second is more substantial: since it is impossible to calculate the complete policy in a ten by ten grid (i.e. for every state representation to support a dynamic state space), the size of the map is reduced to a two by one cell grid. The time required for the previously discussed transformations does not change as their state spaces are not altered by this modification. The entire state space in this example is reduced to

$$8^{WH}5WH * 4 * 4 = 10240 \tag{35}$$

Where $(needed\_minerals + 1)$ and $(needed\_bamboo + 1)$ from equation 10 are replaced with $four$, the number of elements in the set $(0, 50, 100, 150)$, $W = 2$, and $H = 1$. That's certainly better than the original, but it's a pretty useless scenario. Goal states were put at all the valid goal locations (2048 in total). The Q-learning test was set to execute for 20000 episodes with a maximum of 50 actions per episode. Without a better reward function and with a random action selection method, it is very easy to choose a virtually unlimited number of useless actions. In such a high state space this was causing many timing issues using large numbers of actions per episode. I realized that, in general,

if a goal node was found on that episode, it usually took less than 50 actions. In almost all other cases, the maximum number of actions were taken to no avail so to expedite the Q-table learning process the number of actions was set to a smaller maximum and more episodes were added to sample a greater number of starting states.

With these settings, it takes about 15 minutes to execute the learning process, and the resulting policy is not very good. The major problem is that the Q-table remains sparse, with an occupancy rate (i.e. the fraction of non-zero Q-table entries) of only 0.67% . In this example, the sparseness of applicable actions at any given state make the expected occupancy rate very small anyway, estimated at around 6% (i.e. only six percent of Q-table entries in this example should be non-zero, what a waste). Twenty minutes of policy generation resulted in less than 1% occupancy, and that is not useful. The number of episodes was increased twenty-five fold to 500000 resulting in just over five and a half hours of computation and roughly 4.82% occupancy, which still did not perform well as quite a few states are still left with no policies. Clearly this strategy is not useful without either more reward states (which is difficult given the size of the state space) or the incorporation of a heuristic in the Q-learning algorithm. For the vanilla implementation, a good policy could not be generated in a reasonable amount of time, even for a very small grid world.

4.3 Transformation Weights

As mentioned, for each transformation function there exists an associated weight. This effectively raises or lowers the value of the best Q-value returned by the policies. In a sense, it can also be used to normalize the learned policies if they were generated using different scales of rewards. In this case, all goal states for every learned policy had the maximum value of 100, so the Q-values of all policies were in the same range distribution, thus no weighting was required. Additionally, for this example the set of applicable actions on any given state was very sparse so the transformation functions were, in general, not competing with other transformation functions, so weighting was not required. This is not always expected to be the case, but it does provide a good example of why this is very useful in sparse task worlds like this one. For this implementation, all weights were left at unity.

4.4 Comparison

It's difficult to directly compare the results of the two strategies since a sufficient policy could not be generated for the full state space transformation in a reasonable amount of time, even for the vastly reduced grid world. However, one can note that in this example the transformation decomposition approach is vastly superior. A problem impossibly large for Q-learning was easily decomposed into four trivially small subproblems easily solved using the very same Q-learning algorithm with minimal overhead. The policies generated were reloaded (without being regenerated) into a number of different worlds of varying sizes and states and the correct actions were taken every time. The most complete policy generated for the full state space condition was rarely able to compute correct actions for a given scenario on a the small two-cell map.

5. CONCLUSION AND FUTURE WORK

This paper introduces a simple concept of state and action space transformation that allows reinforcement learning to be successfully applied to very large problems where the full state space is otherwise much too large. The concept was demonstrated to work very well in testing world model where for even tiny worlds a full state space policy could not be generated. This model opens the door to applying Q-learning to large domains potentially without losing any resiliency and it provides a useful framework for transfer learning, as the policies generated can be used in any application with a valid state and action transformation function.

References

[1] K. Tuyls, S. Maes, and B. Manderick. Q-learning in simulated robotic soccer large state spaces and incomplete information. *Environment*, page 2427, 2006.

[2] C. Watkins and P. Dayan. Technical note: Q-learning. *Machine Learning*, 8:279–292, 1992.