Modeling the Student with Reinforcement Learning

Joseph Beck
Department of Computer Science
University of Massachusetts
Amherst, MA, U.S.A.
beck@cs.umass.edu

Abstract

We describe a methodology for enabling an intelligent teaching system to make high level strategy decisions on the basis of low level student modeling information. This framework is less costly to construct, and superior to hand coding teaching strategies as it is more responsive to the learner's needs. In order to accomplish this, reinforcement learning is used to learn to associate superior teaching actions with certain states of the student's knowledge. Reinforcement learning (RL) has been shown to be flexible in handling noisy data, and does not need expert domain knowledge. A drawback of RL is that it often needs a significant number of trials for learning. We propose an off-line learning methodology using sample data, simulated students, and small amounts of expert knowledge to bypass this problem.

1 Introduction

Within the realm of intelligent computer based instruction, student models are frequently collected to aid in pedagogical decision making. The most frequent modeling approach taken is to attempt to measure the degree of the student's ability in a "topic". This measurement commonly takes the form of the probability a student has mastered a topic. This suffices for basic reasoning such as whether to promote the student to a more complex topic, or to control the activity's level of difficulty [1]. However, frequently more complex types of reasoning are needed. For example, should the system review a previously learned topic, should it give the student a problem to solve, or should it first present another example of the process involved?

Ideally, the answers to such questions would be based on the system's model of the student, and use principled methods to evaluate these complex decisions. The reality is that most teaching mechanisms are rather ad-hoc. At best it is difficult to attempt to reason about whether a student is confused about topic X or has forgotten topic Y when the only data available are estimates of his ability. Yet the ability to answer such complex questions is a characteristic of a good student model.

In order to support such functionality, we propose constructing a machine learning (ML) element on top of a traditional student model. Building the ML element on top of the student model makes it easier to provide inputs that speed learning and generalize more easily, as the student model can be considered to be constructed expert features for the learning algorithm. ML algorithms typically exhibit a tradeoff in flexibility and training data requirements. That is, the more flexible (i.e. less assumptions) a particular framework is, the more data it needs to reason effectively. Therefore, having a model that tries to reason about what a student knows at a fine-grained level of detail is likely to be less effective than trying to capture more coarse-gained features. Fortunately, much progress has been made on the former via Bayesian Networks [2] and other formal reasoning techniques while less progress has been made on the latter.

2 Motivation

This framework is being designed for use in the MFD (Mixed Number, Fractions, Decimals) system [1]. The goal of this system is to both to help grade school students learn fractional and decimal arithmetic, and to

increase the confidence of students using the system. This system has some heuristics that help determine if a student needs remediation in a topic, has forgotten a topic, needs to be shown an example, etc. Creating a sound foundation for this reasoning is problematic. Most teachers do not think about individual students at this level of detail, so interviewing instructors is not necessarily the best course of action. It is certainly possible to empirically derive a set of teaching rules by running several studies and trying to fine tune the parameters. However, in addition to being expensive, this does not customize interactions to individual students who vary in ways other than the predictor variables gathered. From the student model's standpoint two students could be identical, yet one may learn best from examples while the other prefers to puzzle out problems himself. It is not possible to disambiguate these students without the system learning about the student while it is being used.

Therefore, we are moving away from hand coding routines that evaluate the student's state and beginning to investigate ML methods. We are currently considering placing a learning agent on top of the student model, and having it learn the probable long term benefit to the student for each teaching interaction when used in the current situation. This permits a significant amount of data to be gathered, as the system is not trying to learn about the student's performance in a narrow area, but rather it generalizes across all of his interactions. To accomplish this, it is necessary to carefully construct a state representation that is neutral across topics. That is, features given to the machine learner should be phrased in terms like "knowledge about this skill", or "recent performance at prerequisite knowledge", not "misconceptions in finding least common multiples".

3 The Learning Methodology

There are several features a learning framework must have to be useful to us. First it must be possible to incrementally update the system in real time (on a classroom personal computer). This excludes many neural network algorithms, as they tend to work best with batch training. There must also be some method for handling stochastic data as students do not exhibit constant behavior by forgetting knowledge, making typos, etc. For these reasons we have chosen to use reinforcement learning as a general framework.

3.1 Reinforcement Learning

Reinforcement learning (RL) is a system for providing target values for a functional approximator [7] [9]. It best thought of as a combination of dynamic programming (DP) and monte carlo (MC) methods. While the RL system is solving a problem it is learning approximate values for each state that it encounters, and in many ways gains the best of both DP and MC. Similar to MC, RL learns based on actual experience. This is unlike DP, which in computing "sweeps" may derive values that are never actually encountered. Like DP, it does not need to know the eventual outcome of an action to start adjusting the estimated values of states. DP requires a strong model to perform such calculations, while RL simply uses its actual experience as a best estimate. That is, it guesses the value of the current state from its guess about the value of adjacent states.

Systems built with RL have outperformed expert humans players in backgammon [10], and have demonstrated superior performance in elevator scheduling compared to commercial, hand crafted algorithms [3]. These results suggest the potential for impressive performance by these systems. An important point is that RL is not a learning algorithm per se, but rather a method of determining the value of being in a certain state. This value is then used as the target for a machine learner. Briefly, RL provides the machine learner with "rewards" for good performance and no reward for poor performance (alternately, no reward for good performance and a negative reward for poor performance). The goal of the learner is to maximize the amount of rewards received over the long term.

RL is agnostic on which ML algorithm should be used to compute the value of a state. Successful systems have been built using neural networks [10], linear function approximators (LFA), and lookup tables [9]. Given that the amount of training data is not likely to be large, using a simpler scheme such as an LFA seems prudent. If the problem is small enough, a lookup table that stores learned values may be applicable.

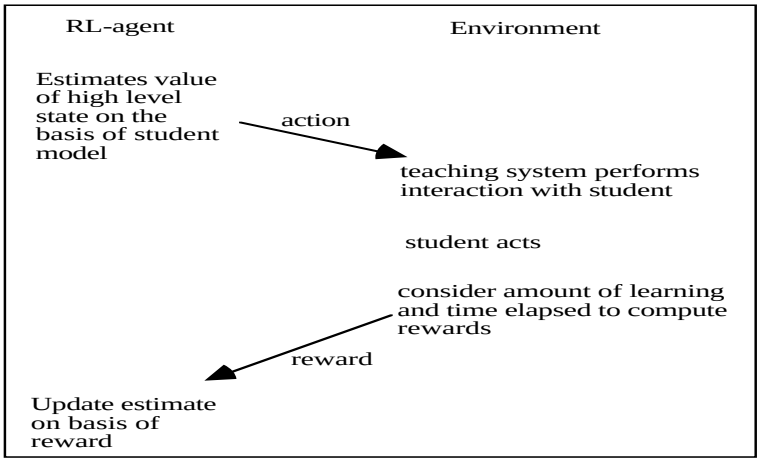

Figure 1: RL agent interacting with a student.

There are two components in an RL system: the environment and the actions. The environment is whatever is beyond the direct control of the learner, while actions are items the agent can select. Note that actions themselves may be stochastic (e.g. actions can be performed incorrectly). The agent examines the current state in which it is, and then selects an action to perform. At this point the environment takes over and the effects of the action performed are observed. Based on the new state in which the agent ends up it is given a reward based on previous estimates of this state's value.

Once the reward is received some method of moving the estimate of the previous state towards the reward is needed. The simplest such method is simply to use the Widrow-Hoff rule [12]: $\Delta w_t = \alpha(z - w^T x_t)x_t$, where α is the step size (or learning rate) parameter. The main difference between RL and other systems that use this update rule is that RL uses the next state as its target value, while most other algorithms use the result in the final state of the "trial". Waiting until the end to reward the learner has several drawbacks. First many tasks (such as this one) have no clear notion of what constitutes a trial. Additionally, delaying the reward until the last state produces much more variance in the values during learning, and consequently such systems tend to take longer to converge to optimal values.

3.2 General Framework

For our task of determining "high level" information about the student, we use the methodology shown in Figure 1. The system constructs its state representation by examining the individual's student model. As in Figure 2 it computes the expected utility from using each teaching action the system supports (present an example, remediate a misconception, teach a topic, etc.). The reward mechanism can calculate the effect the action has on the student by using the student model to calculate the amount of learning, and discounting this for how much time the teaching action required. The reward is given the RL agent, and the value of performing the action in the current context is moved towards this reward.

Some actions such as presenting an example will clearly not have immediate, noticeable effects but are useful in the long run. Fortunately, such problems have been examined in the context of RL. First, since the system is learning to associate the value of a state-action pair with the value of the succeeding state, eventually values will propagate back to the current state. Unfortunately, this process can be slow. Therefore, a set of learning procedures known as TD(λ) [7] have been developed. This adjusts the value of a state-action pair to be equal to a combination on the values of the next several states visited, giving decreasing weight to successive states. Drawbacks include having to keep track of what states have been visited. This TD framework is a generalized form of other learning algorithms. TD(0) is equivalent to only considering the next state, while TD(1) is equiva-

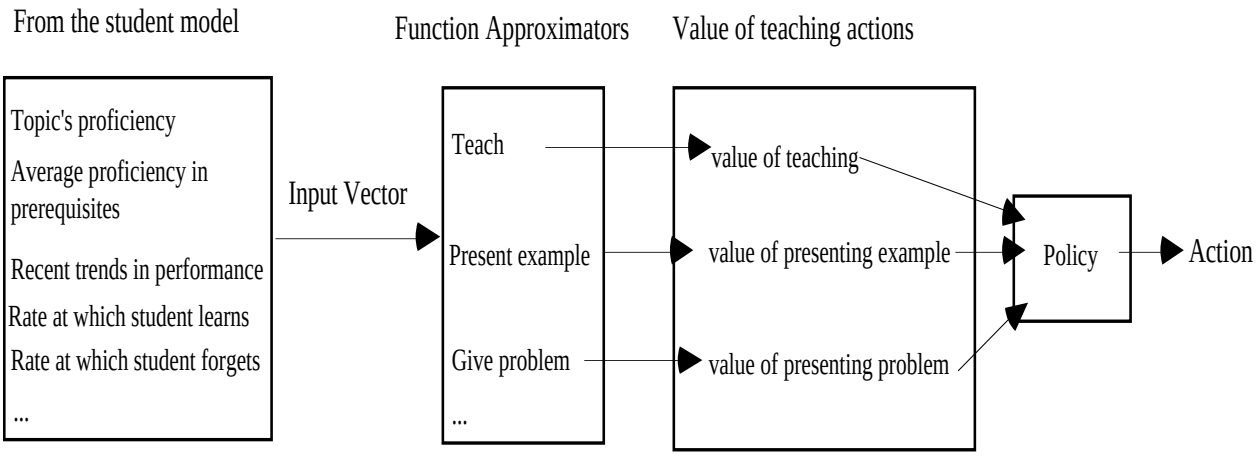

Figure 2: State construction and evaluation of teaching actions.

lent to monte carlo learning, which uses the final outcome in a learning trajectory as a target. Using intermediate values of Lambda controls how far to propagate values, and has been shown to be somewhat better than TD(0) and much better than TD(1) [8].

By learning the value of performing teaching actions in a state, the system is learning to customize its behavior to the student. Once the systems learns how to compute the expected gain for using each teaching interaction, a teaching policy can be constructed. A first attempt might be to select the action with the highest expected reward. However, this has a large drawback that little exploration is performed. Once an action is found that is slightly better than the others, it will be performed exclusively. A solution is to perform action selection via a Boltzmann machine: initially the temperature is set high, but as the system becomes more accustomed to working with a student the temperature is decreased. This results in more variation in action selection initially, which allows for exploration, but over time the system becomes more greedy in maximizing its reward.

3.3 Strengths

As previously mentioned, RL is robust with respect to noisy data. This is due to its learning the value of the current state from the value of the next state (i.e. a "guess from a guess"). In spite of being counterintuitive, this has been shown to give good performance in a variety of learning tasks. An additional advantage is that substantial domain knowledge is not required to construct an RL system, and it easily incorporates the existing knowledge in the student model. Obviously using domain knowledge makes any learner's job easier, but a simple simulation (or observing the actual task occurring) is sufficient for RL. It is possible to use additional domain knowledge to provide hints to the learner. Constructing a good state representation is an example. A program learning to play chess would certainly learn more quickly if it was given features that described "in check", "can capture", etc. Additionally, knowledge can be used to bias the initial evaluation function or to allow the system to construct simulations of what might happen, enabling it to learn without making mistakes on the actual task.

4 Learning Procedure

Thus far, we have outlined a procedure for how a learning system can adapt its actions to fit individual students' needs. However, if the system starts with no knowledge, its initial performance will be random. Thus some method of providing the system with starting knowledge is required. Once this step is accomplished, the system will have a general model of how to teach a domain, but will not take into account idiosyncrasies of the

student currently using the system. Given these facts, we propose a two-phase learning procedure with the first phase being done off-line, and the second occurring in real time as the student uses the system.

4.1 Off-line learning

There are a variety of approaches for training such a system off-line. The most obvious is to take records of students using a similar system (that does not use ML to select teaching actions), and use these interactions as training data. While straightforward, this has a large cost of running a substantial experiment whenever a new system is being designed.

An alternate course is to use simulated learners [5][11]. This would entail constructing a weak model of how "typical" students would interact with the system. In order to represent the variance in student learners, several such learners could be constructed. The system would learn from interacting with the simulated learners (simulees). This is tractable problem, as the models do not have to be extremely accurate.

Finally, the system could be biased towards performing certain actions in some states by altering the function approximator's initial coefficients (this is simpler if an LFA is used). The result is that the system will initially perform according to the constructed policy, but will learn when to alter from this path.

In all of these cases, the learner takes a student model as input and decides what teaching action is appropriate; it does not simply learn a model of how to teach that is independent of the learner's level of knowledge. It is likely that some combination of the three techniques listed above will be the most efficient method to obtain good performance. For example, it may be best to build some weak simulee models to get a good first approximation, and then to gather a small amount of training data. In either event, the goal of this is to produce a system that can interact with students and learn from them on-line, and has performance that is good enough to be used by students. By providing these weak initial methods, the cost of determining pedagogical behavior is decreased (as opposed to constructing a policy by hand).

Given the high cost of hand-coding teaching action selection, Quafafou's NeTutor [6] proposes providing the system with examples, and using machine learning to learn teaching actions applicable to a specific student. A difference between our approach and the one outlined there is that they were concerned with learning symbolic rules while we are interested in numerical techniques. The advantage of using a function approximator is that more data can be considered (only 3 items were used to describe a state in NeTutor) and the system can generalize results more quickly.

4.2 On-line learning

Once the system has been trained to have adequate performance, it can be used with actual students. The system (hopefully) has a good general model for how teach students (with their student model as input to the system), but has yet to learn about idiosyncrasies that the student using the system may have. In order to account for this, the system learns while the student is using it. The system selects an action, the student acts, and a reward is received. If the student learns differently from the standard model, the rewards received will be less than (or greater than) expected, and the system will alter its teaching strategy selection accordingly.

Conceptually, the result of this is a layering of the knowledge about how to teach a student of this type. At the bottom are the raw data recording student actions, above this the standard student model data, then a general notion of how to teach such a student, and finally the knowledge of how to teach this particular student.

In order to gather the most from each item of student datum, replay [4] can be used. This technique presents several examples of useful trials. If an action receives a particularly large or a particularly small reward then it is selected to be presented to the system several more times (i.e. the state, action, reward sequence is given to the function approximator several times). Lin [4] has shown that this results in faster learning.

5 Conclusions and Future Work

Using a learning algorithm in the method we describe has clear advantages over hand coding teaching strategies. First, it has the potential to be less costly than spending time constructing and empirically validating a pedagogical module. Second, it has the ability to customize its interactions to individual students. Two students can have identical student models but require very different teaching actions. A hand crafted strategy that "simply" takes the current model as input has no way of knowing this.

The major difficulties in using such a framework is the selection of features to provide to the learner (this is true for any task, but is somewhat more difficult here), and providing the system with its initial training data. Simulees seem to hold the most promise as they can potentially be modified and used to train widely different systems. There is a considerable fixed cost for building the first RL student modeler, but the marginal cost for similar future systems is greatly reduced.

We are currently planning a next generation version of our mathematics tutor (MFD) and are exploring how to use this proposed methodology in customizing instruction to the student.

References

[1] J. Beck, M. Stern, and B. Woolf. Using the student model to control problem difficulty. In *Proceedings of the Seventh International Conference on User Modeling*, 1997.

[2] J. Collins, J. Greer, and S. Huang. Adaptive assessment of using granularity hierarchies and Bayesien nets. In *Proceedings of Intelligent Tutoring Systems*, pages 569–577, 1996.

[3] R. Crites and R. Sutton. Improving elevator performance using reinforcement learning. *Advances in Neural Information Processing*, 1996.

[4] L.-J. Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8:293–321, 1992.

[5] J. Mertz jr. Using a simulated student for instructional design. In *Proceedings of Seventh World Conference on Artificial Intelligence in Education*, 1995.

[6] M. Quafafou, A. Mekaouche, and N. H.S. Multiviews learning and intelligent tutoring systems. In *Proceedings of Seventh World Conference on Artificial Intelligence in Education*, 1995.

[7] R. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

[8] R. Sutton. Generalization in reinforcement learning: Successful examples using sparse coarse coding. *Advances in Neural Information Processing Systems*, 8:1038–1044, 1996.

[9] R. Sutton and A. Barto. *An Introduction to Reinforcement Learning*. MIT Press, 1997.

[10] G. Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3), 1995.

[11] K. VanLehn, S. Ohlsson, and R. Nason. Applications of simulated students: An exploration. *Journal of Artificial Intelligence in Education*, 5(2):135–175, 1996.

[12] B. Widrow and M. Hoff. *Adaptive Signal Processing*. Prentice-Hall, Englewood Cliffs, N.J., 1960.