Noname manuscript No.
(will be inserted by the editor)

On the Analysis and Design of Software for Reinforcement Learning, with a Survey of Existing Systems

Tim Kovacs · Robert Egginton

Received: date / Accepted: date

Abstract Reinforcement Learning (RL) is a very complex domain and software for RL is correspondingly complex. We analyse the scope, requirements, and potential for RL software, discuss relevant design issues, survey existing software, and make recommendations for designers. We argue that broad and flexible libraries of reusable software components are valuable from a scientific, as well as practical, perspective, as they allow precise control over experimental conditions, encourage comparison of alternative methods, and allow a fuller exploration of the RL domain.

1 Introduction

Reinforcement Learning (RL) has both theoretical and empirical sides, and the latter typically involves writing code and running experiments with it. We argue that writing RL code is a complex subject because RL itself is a very complex domain. Although there are a few well-known and heavily used RL algorithms (such as Q-learning and Sarsa) that an agent can use to learn a policy, they are just points in a space of possible RL algorithms, which, as Sutton and Barto (1998) explain, has many dimensions. As examples, agents may use value functions or direct policy search, be model-based or model-free, on-policy or off-policy, use 1-step or many-step backups, use a wide range of exploration control methods and a wide range of Supervised Learning (SL) function approximators.

This space of agents maps onto the space of environments in a complex way; for one thing, there is no one RL algorithm which is best for all environments. Furthermore, the space of RL environments has its own structure. Although the paradigm of iterating through states, actions and rewards might appear to define a set of very similar learning problems, these problems can differ significantly depending on, for example, whether the state is Markov, fully observable, continuous, metric, or whether there is some delay in observing it.

Department of Computer Science
University of Bristol
Tel.: +44 117-954-5145
Fax: +44 117-954-5208
E-mail: {kovacs,egginton}@cs.bris.ac.uk

In addition to the complexity of RL itself, we must consider the complexity of related areas. There are often alternatives to RL, for example, dynamic programming and planning methods can often be applied to the same problem, and comparisons *should* be made between alternative approaches. In addition, RL agents typically incorporate supervised function approximators to represent their value functions, so, in practice, RL experiments often involve SL.

In addition to environments and agents, there is great diversity in RL experiments. For one thing they may generate a wide range of statistics; as examples we might ask an agent how much reward it obtains over time, or ask an environment how many steps are taken to reach a goal each episode, or ask a Q-function to write itself to a file. Experiments may also differ in terms of how many agents are interacting, whether a Graphical User Interface (GUI) must be updated, whether there are separate training and testing phases and so on.

The aims of this work are to analyse the scope, requirements and potential for software which can support the complex domain just described, to discuss relevant design issues, to survey existing software, and to make recommendations for future work. In section 2 we distinguish different kinds of RL software, and discuss what each kind provides and how useful each is. We argue that libraries of reusable code are not only of more practical use than repositories of agents, but make for better science. In section 3 we present a fictional story about a researcher's experience implementing RL software, in order to illustrate the difficulties involved and to motivate a discussion of requirements for RL software. Section 4 discusses the different users of RL software and the purposes for which they use it, lists requirements and optional features for such software, and argues that analysis and design of RL software can reveal insights into RL. Section 5 discusses a range of design issues of particular relevance to RL, including how software constitutes a model of the RL domain, and how to select the scope of an RL system. In section 6 we survey existing RL software and in section 7 we conclude and give three recommendations for designers of RL software.

2 On Software for Reinforcement Learning

In this section we distinguish different kinds of RL software, discuss the utility of the different kinds, briefly mention what systems currently exist, and, finally, discuss the challenge of designing and implementing such complex software.

2.1 What RL software can do for you.

Software written for research, whether for RL or for any other area, may be used in a number of ways, including: i) as an executable, ii) as a specification (e.g. of an algorithm), iii) as a source of design ideas and iv) as a source of code to reuse. We now distinguish various kinds of RL software, in no particular order, and briefly summarise what they provide.

- *Ready-made agents and environments*, which allow the user to try them without implementing them. This may be a single agent/environment pair, perhaps made available by the author of a paper. The source code may or may not be available; if it is, this can be a way of specifying the details of published work. Ready-made

agents and environments may be collected and made available in a *repository*. If the contributions are by different authors they will tend to be mutually incompatible: each agent may only work with its own environment. Ready-made agents and environments may also be provided by a library of reusable code (see below), in which case they should be mutually compatible.

- *Protocols for communication* between agents and environments (or other subsystems). In sample-based RL, on each time step the environment generates a state, the agent chooses an action, and the environment responds with a successor state and reward. A protocol defines the order of these events, the names of the functions called and the type of the data passed on each call. (In software engineering terms, the protocol consists of the public interfaces of the agents and environments, and information on the sequence of events.) If a repository contains pairs of RL agents and environments, and each pair uses a different protocol, then we cannot mix and match agents and environments. In contrast, if a community of researchers adopts a common protocol then their software can interoperate, which makes repositories much more useful. The fictional case study in section 3 illustrates in detail the barriers posed by incompatible protocols.
- Libraries of *reusable software components*, which the user can take as building blocks for writing their own agents and environments, or other subsystems. For example, a library might contain a range of action selection methods (ε-greedy, softmax, and so on) which the user can call. In section 5.2 we distinguish a special kind of library called a framework, which guides the flow of control and provides more of the architecture of the application eventually written by an end-user than a regular library. All but one of the libraries we survey in section 6 are frameworks. Many libraries which are not specifically intended for RL provide things such as function approximators and GUIs. Note the distinction between a library, which is designed to help write software, and a repository, which provides complete software which is ready to use.
- *Experimental platforms*. Empirical research involves more than running an agent in an environment and the platform is whatever handles issues such as repeating experiments, averaging results, varying parameter settings over a series of experiments, plotting data, statistical tests, and so on. The platform may be compositional: composed of independent or loosely-connected tools. (An example compositional system is the tool chain we used to create this document: LaTeX to compile it to a DVI file, BibTeX to add citations, dvips to convert the DVI file to postscript and finally ps2pdf to produce a PDF file.) Alternatively, the experimental platform may be integrated, such as the supervised learning systems WEKA (Witten and Frank, 2005; Hall et al, 2009) and RapidMiner (Mierswa et al, 2006), or the numerical environment MATLAB (Moler et al, 1987). (There are also integrated environments for LaTeX; both compositional and integrated approaches have their supporters.)

All of the above are useful on their own and none are mutually exclusive; an RL library, for example, typically contains a protocol and provides a set of ready-made agents. The design of experimental platforms is beyond the scope of this paper, though we will mention them occasionally to illustrate possibilities.

2.2 Current RL software.

Of the systems we review in section 6, all are at once both libraries and protocols, except RL-Glue, which is a protocol only. There are currently no fully-featured experimental platforms such as WEKA dedicated to RL, although the MDP library we cover in section 6.7 is part of the MATLAB platform. Of the other systems we survey, MMLF has probably the best integrated experimental platform: it allows plotting and logging of data, choice of agent and environment, parameter tuning, and support for setting up simple experiments to compare two different agents.

A number of other systems were not included in the survey, for reasons given in section 6. These include the RL Repository,[1] a repository of agents and environments written by diverse authors (as well as papers and other resources). There are also two projects associated with RL-Glue: the RL-Logbook (Tanner, 2009b) and the RL-Library (Tanner, 2010). The RL-Logbook is an experiment database of the kind described in section 4.4, and could be considered an aspect of an experimental platform. It is currently not functional and as of January 2011 its site had not been updated since September 2009. The RL-Library is, in our terminology, a repository, and is specifically for RL-Glue compatible systems. As of January 2011 it has 3 agents, 10 environments, 2 experiments and 4 tutorials.

Some evidence for the use of RL-Glue comes from its role in recent annual RL competitions,[2] but in general we lack evidence on the usage of various systems. Despite a lack of data, however, it is clear that neither is the state of RL software as mature as it might be, nor is the take-up of existing RL systems what it might be. First, there is no one system which is universally recognised as the standard in the community. We do not mention this to propose it as an ideal towards which the community should work, but only to point out that if it were the case, it would demonstrate a much greater take-up of software, and convergence of the field, than there is now. Second, in SL, comprehensive, integrated systems such as WEKA and RapidMiner have been available for years. These are major undertakings (Bouckaert et al, 2010), and nothing comparable has, to our knowledge, even been attempted in RL, let alone released. (One reason may be that RL is a more difficult domain for which to produce software; see e.g. the discussion on environments as code vs. environments as data in section 3.2.) Third, of the systems we survey, only RL-Glue provides a protocol and not a library. While this protocol specifies interactions between agents and environments, it does not cover the interaction between an agent and its value function. Thus, more comprehensive protocol-only systems could be introduced than we have at present. Fourth, we are not aware of large communities actively developing RL software. If, for example, there was a website showing weekly progress being made by dozens of developers, we would have a different impression of the state of RL software than we do. In summary, we have little evidence for the use of various RL systems, and our impression is based largely on a failure to observe more activity, which is admittedly not the best form of evidence. It would be very useful to survey RL researchers to ask which systems they have evaluated or used, and why.

Given that a number of useful RL systems have been available for years, why have they not seen greater take-up? A lengthy discussion is beyond the scope of this work, but one factor is undoubtedly that established RL researchers have their own code and

[1] `http://www-anw.cs.umass.edu/rlr/`

[2] See (Whiteson et al, 2010) or `http://www.rl-competition.org/`.

do not relish the overhead involved in adopting a system written by others, even if it would save time in the long term. Designers of RL software should take this into account and, for example, make their systems as easy to use as possible. Another factor is that researchers often enjoy implementing their own software (Stodden, 2010).

2.3 The utility of different types of software.

In this section we argue that repositories of ready-made agents and environments are tremendously useful as, for one thing, specifications of experiments. Nonetheless, we argue that libraries of reusable software components have very important advantages over repositories.

Sonnenburg et. al (2007) present many arguments in favour of open-source machine learning software over non-public software. Most of their arguments are beyond our scope but to give a sense of their position we quote their list of 11 advantages of open-source ML software:

"1. Reproducibility of scientific research is increased
2. Algorithms implemented in same framework facilitate fair comparisons
3. Problems can be uncovered much faster
4. Bug fixes and extensions from external sources
5. Methods are more quickly adopted by others
6. Efficient algorithms become available
7. Leverage existing resources to aid new research
8. Wider use leads to wider recognition
9. More complex machine learning algorithms can be developed
10. Accelerates research
11. Benefits newcomers and smaller research groups"

The futility of exact replication. In response to point 1, and to similar points made elsewhere, Drummond (2009) distinguishes between replication, by which he means repeating *exactly* the same experiment, and reproduction, by which, in the context of ML, he means repeating an experiment under different conditions. He claims the value placed on reproducibility comes from the physical sciences, where exactly repeating an experiment is impossible, and hence replication is necessarily reproduction. In contrast, in computer science we often *can* replicate exactly (including using the same pseudo-random numbers), but, as he points out, true replication is pointless since we must get the same result.

The utility of reproduction with variations. Drummond suggests that in the context of ML, reproducibility should mean reproducing the result of an earlier experiment with a *different* experiment. That is, we should seek to show that the same result occurs even when some of the conditions have changed, because this allows us to refine our hypotheses about the relationship between experimental conditions and results. We will now expand on Drummond's point to show that obtaining different results is also useful, and link this discussion to the version space framework.

Let us begin with an abstract example. Suppose, for simplicity, that the result of an experiment is a deterministic function of its parameters, and that all parameters

are binary. If we run one experiment, we know the result for a given list of parameter settings. If we replicate this experiment exactly we gain no new information, but suppose we change exactly one parameter value. If we obtain the same result, we can remove that parameter from the list because the result is invariant to that parameter's value.

A more concrete example may help. Suppose a first experiment shows that, when using tabular Q-functions, Q-learning outperforms Sarsa in a given environment. We might be tempted to hypothesise that Q-learning is better than Sarsa in this environment. That hypothesis is consistent with the results, but it is a rather general hypothesis, and perhaps over-general: it could be that the results depend on the type of function approximator used. To avoid the risk of overgeneralising, we could adopt the most specific hypothesis consistent with the results: that *tabular* Q-learning outperforms *tabular* Sarsa on this environment (and given whatever other parameter settings were used). If a second experiment again shows Q-learning outperforming Sarsa, but this time both use tile coding, we can generalise our hypothesis to be: Q-learning is better regardless of which of the two function approximators is used. Further experiments might show the same result with other function approximators, and at some point we might feel confident enough to adopt the general hypothesise that the result holds regardless of the kind of function approximator used. (We can only be certain that hypothesis is true if we actually evaluate all possible function approximators, but that is the nature of induction.)

The point of our examples is to illustrate that science is a search process and that search progresses by varying experimental conditions. Incidentally, search falls within the remit of Machine Learning and hence, to the extent that science is a process of search, so does science. In our examples, a series of experiments refines the set of hypotheses which are consistent with experimental results. Since the experiments are deterministic, our examples fit the version space framework of concept learning (Mitchell, 1997), which clearly describes the potential of logical induction in the absence of noise.

Libraries are more valuable than repositories. We agree with Drummond that reproducibility should involve variations, but add that it has implications for the utility of different types of software. If a repository contains agents written by different people at different times, we are unlikely to have fine control over their differences (unless we modify their code extensively, but that reduces the value of having the repository). For example, both the Q-learner and Sarsa implementations in a repository may be tabular. Do both support tile coding? If not, we cannot generalise our hypothesis as in the example above. Worse, they may – in fact, probably do – have many minor differences, such as the action selection method used. This makes it hard to attribute the outcome to any one difference. It is worse still if one supports only a tabular Q-function and the other only tile coding. What can we conclude in this case? Is the difference due to the learning algorithm or the function approximator? Drawing precise conclusions from experiments requires precise control over experimental conditions, or, put another way, precise control over the search process that is science. A repository of inconsistently designed agents does not facilitate precise control, but a good library does. Hence, we argue that libraries make for better science than repositories.

The scientific value of broad libraries. The scientific value of a library is related to its breadth, by which we mean the range of features it has, for example, the range of agents

it provides. Broad libraries are particularly valuable for several reasons. First, they allow better control over experiments than more limited ones. Note that this scientific value is distinct from, though based on, the more practical value of a library as a time-saving source of reusable code. Second, researchers tend to restrict their comparisons to their favourite algorithms or approaches. So, for example, RL algorithms are often not compared to planning methods, even when both can be applied to the same problem. A broad library reduces the barriers to wider comparisons by making it easier for a researcher to make these comparisons, and easier for others to make them if the first researcher does not.

Sonnenburg et al. (2007, section 3.5) give a third reason, which they call "combination of advances". They point out that, in any given area, researchers are likely to be simultaneously studying a range of improvements on a basic strategy. This is certainly the case in RL. The difficulty is that a set of improvements may interact (that is, they may be epistatic), which means we cannot assume that using them all is the best solution, or that they are all necessary. It may be that two improvements exploit the same underlying principle and that while either alone results in an improvement, using both is no better. Worse, it is possible that they conflict and that using both results in worse performance than using neither. The point is that since they interact we cannot evaluate them independently, and to evaluate them simultaneously we need a system which implements them all; such a system is necessarily broad.

Sonnenburg et al.'s argument can be taken farther: it is not just potential improvements which may interact, but many aspects of the experiment. As noted in the introduction, RL has diverse agents, environments and experimental set-ups. We could ask how suitable a given agent is for any given experimental set-up and environment and the answer would give us, to borrow a concept from biology, a *fitness landscape* for the agent. Since different niches (parts of the landscape) will impose different requirements, the agent will perform better on some than on others; sadly, the conservation law of generalisation (Schaffer, 1994) and No Free Lunch theorem (Wolpert, 1996) tell us that no one agent will be best for all niches. Indeed, the mapping between agents, environments, experimental set-ups and fitness is likely to be a complex one (all the more so since agents are not atomic but themselves have internal structure).[3] A long-term goal of RL research is to determine which agents are best for which niches, and in order to do so we need software broad enough to allow the mapping between agent, environment, and experimental set-up to be explored thoroughly.

Repositories are also valuable. We agreed with Drummond on the value of reproduction over replication, but we diverge from him on the value of repositories. Although we have argued that libraries are much more valuable, we believe repositories nonetheless have much value. Drummond states: "I accept that there may be other virtues for having repositories of software from various sources. My claim here, though, is that scientific reproducibility is not one of them". First, we are convinced that repositories have great value: many papers lack sufficient detail for replication (Sonnenburg et al, 2007), and repositories can fill in these missing details. Second, Drummond encourages reproduction of results with different experiments, and the best way to know what you are trying to reproduce is to have the source code. In short, while rerunning deterministic programs is pointless, source code has tremendous value as a specification.

[3] See Sloman (1994) on the complex mappings which often arise between a space of designs and a space of requirements.

Having the original source code makes replication trivial, while lacking it can make replication impossible. We believe Sonnenburg et. al were pointing out the value of source code as specification when they said that open source code has an advantage in terms of reproducibility.

The cost of writing public software. Drummond points out there is a considerable cost to making software suitable for public use, but we note that when built using a library, the cost of writing the software in the first place, as well as that of preparing it for public consumption, can be much reduced. This is significant. Stodden conducted a survey on the sharing of code and data, using 134 researchers drawn from the registration lists of the NIPS conference up to and including 2008 (Stodden, 2010). The top two reasons given for not sharing code were "The time it takes to clean up and document for release", cited by 55.64% of researchers, and "Dealing with questions from users about the code", cited by 43.61% of researchers. Both contribute to the time cost of releasing code, which Stodden noted was by far the biggest disincentive. Good libraries can significantly reduce both problems.

Conclusion. We have argued for the scientific value of code repositories, which fully specify experiments, unlike many papers. We pointed out the utility of protocols in section 2.1: they allow interoperability within a repository, and make it much more useful. However, libraries of reusable code are even more useful as they also allow precise control over experiments, and encourage comparison of alternatives and hence a fuller exploration of the space of agents, environments and experiments.

2.4 The design hurdle.

The existence of good RL software solves design problems for those who use it and hence makes it easier to implement RL experiments. However, designing a good system in the first place is a major undertaking (see section 5.2), and is perhaps even more difficult than implementing it once it has been designed, in terms of both the time and expertise required. We address various aspects of this design hurdle in this paper.

Once the design hurdle has been surmounted, the core system has been implemented, and the system has some minimal functionality, then much less expertise and commitment is needed by any one individual in order to make progress. For example, although the full design hurdle would be too great a challenge for most undergraduate students to take on as a project, once the hurdle has been passed it should be possible for a good student to contribute to the growing functionality of the core implementation by adding e.g. a GUI. It should be *easy* for a student to add new RL algorithms and environments; good software should make such extensions easy. Thus open-source systems which have passed the design hurdle have the potential to grow through relatively small contributions from many individuals.

3 Alfred's Experience

In order to explore issues concerning designs and requirements we present a fictional yet plausible story about Alfred's experience writing RL software. We take this approach as a story is more concrete and hopefully clearer than an abstract discussion. The story

also provides structure: the complexity of the scenario grows incrementally. It should, to some extent, mirror the experience of readers who have undertaken the implementation of RL software. However, while many researchers will have addressed the same issues Alfred faces early on, his later work, on more complex and comprehensive systems, takes him into less well-known territory. In a very few places it will help make our discussion concrete to assume that Alfred is working in Java, but for the most part the discussion is language-neutral.

3.1 Alfred decides to write his own system

Alfred is a great programmer and is familiar with Supervised Learning (SL) but new to RL. After reading Sutton & Barto's text (1998) he becomes interested in applying Q-learning to the game of blackjack. Alfred is familiar with WEKA, an open-source system for SL which can be used through alternative graphical user interfaces, from the command line, or as a library for user-developed applications. With the help of a data-mining book based around WEKA (Witten and Frank, 2005), Alfred has used WEKA to visualise data, preprocess it, apply a set of supervised learners to it, and then postprocess and visualise the results. He has even set up pipelines of events, such as the process just described, using WEKA's graphical interface. Alfred searches in vain for a comparable tool for RL. He comes across a number of relevant systems but nothing as comprehensive, easy to use or well-documented as WEKA. In the end he decides to implement Q-learning and blackjack from scratch. His reasons are many: he cannot find a suitable library in his favourite language, he does not anticipate it being very difficult, he thinks he will learn more this way than by adapting an existing library (although it may take longer), he has had bad experiences with poorly-documented libraries in the past, and, finally, he likes programming.

3.2 Environments as code vs. environments as data

In supervised learning, learning problems are normally specified by datasets and a great many are freely available on the web, for example from the UCI repository (Asuncion and Newman, 2010). Datasets can be encoded in many formats. A simple one is the Comma Separated Value (CSV) file, which is human-readable and widely understood by e.g. spreadsheets. Alfred's instinct is to download a dataset of blackjack games to use as the basis for learning. This is not an unreasonable approach, especially if he wanted to learn about the way humans play blackjack and could find transcripts of games played by people. However, this is not his aim; instead he wants a Q-learner to learn to play blackjack from scratch. He quickly realises that in RL the interaction of the learning agent and environment generates the data that drives learning. Although this data can be captured in a static file and RL agents can learn from it in an off-line manner, it is far more common for RL agents to learn on-line from their own interactions with the environment. Alfred has come to appreciate a significant (and widely-understood) difference between SL and RL. In terms of software, this presents a requirement that environments can be expressed through executable code, rather than by a static dataset.

The obvious approach is therefore to write a blackjack simulator. However, Alfred knows that code and data are in a sense the same – ultimately both are just sequences of

```
Initialise Q arbitrarily
Repeat (for each episode):
    Initialise s
    Repeat (for each step of episode):
        Choose a using policy derived from Q
        Take action a, observe r, s'
        Q(s,a) = Q(s,a) + α[r + γ max_a' Q(s',a') − Q(s,a)]
        s = s'
```

Fig. 1 Q-learning

bits – and he wonders whether blackjack can be conveniently specified by a combination of a data file and code. He knows that RL problems are often formalised as Markov Decision Processes (MDPs) and realises he can formalise blackjack as a finite MDP, that is, as a set of states, actions, a transition function and reward function. The last two can be represented as matrices and all are static and can be stored in files (Comma Separated Value files, for example). To create a blackjack environment for an RL agent he would only need to add code to "animate" the MDP specified by the files; that is, to define the current state and sample from the transition and reward functions. Furthermore, the same MDP-running code would work for any MDP, so to apply Q-learning to other environments he would only need to load other datasets – just as in SL. He notes that although different researchers might use different data file formats, code can be in different languages and follow different specifications. Consequently, he expects researchers would find it easier to exchange data files than code.

While specifying MDPs in files has some appeal, with a little more thought Alfred uncovers some drawbacks. First, the dataset would require much more memory than would be needed to implement blackjack only in code. (In Sutton and Barto's simplified formulation the player has 200 states and 2 actions so the state transition matrix alone would have 80,000 real values.) While this is not a problem for blackjack, it is easy to find environments which are too large to ever store; consider chess, for example. More importantly, Alfred sees that creating the files specifying the MDP is non-trivial. In fact, he cannot think of an easier way than writing code to model blackjack and extracting the relevant statistics from it, which he anticipates will actually be more work than writing a blackjack simulator. He also foresees that while analysing the data files (with visualisation methods, statistical analysis, and data mining) might be interesting, when debugging or attempting to understanding the process it would be far easier to work with the code. Eventually, on rereading section 5.1 of Sutton & Barto, he realises the authors do make the point that writing simulations is often much easier than explicitly writing out the figures for the equivalent probabilistic process. Alfred decides to write a blackjack simulator entirely in code (although he revisits this decision in section 3.6).

3.3 Customising experiments

Alfred implements a blackjack simulator and tabular Q-learning. For reference, figure 1 shows the high-level pseudocode for Q-learning, based on that in Sutton & Barto (section 6.5). By 'high-level' we mean that each line may abstract over and hide a great many details. To the basic algorithm Alfred adds code to print the number of

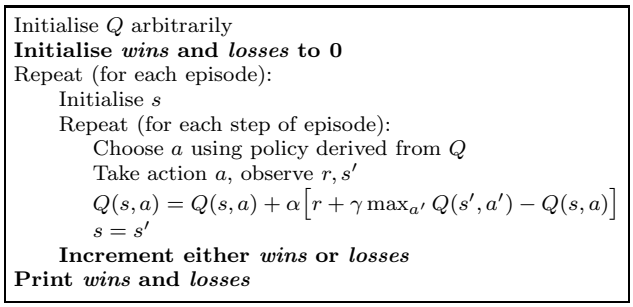

Fig. 2 Q-learning with additions (in bold) to report the number of wins and losses

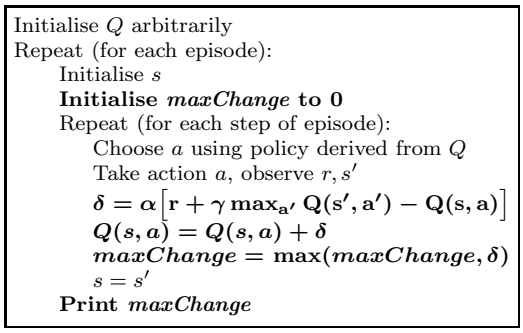

Fig. 3 Q-learning altered (in bold) to report the maximum change in Q-value each episode

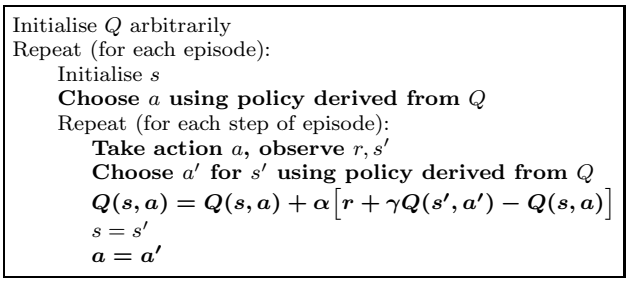

Fig. 4 Sarsa, with differences from Q-learning in bold

games won and lost, shown in bold in figure 2. Note how the code which generates and reports statistics is embedded within the high-level Q-learning algorithm at more than one point. This is typical of the statistics-generating code in an RL experiment. Alfred becomes interested in how the Q-values converge over time and decides to plot the maximum change in a Q-value on each episode. To do so he modifies his code to that shown in figure 3. Again, this requires modification of the high-level pseudocode, but in different places. Alfred now has three versions of his code, each with a slightly different high-level algorithm. Of course he could combine them but evidently there are many other forms of output he might want to generate and continuing to add them all to the

same code is not a good option: he will need a way to organise his variations on the basic experiment. Next Alfred implements Sarsa for comparison with Q-learning and notes that despite a strong overall similarity, about half the code is subtly different. (See figure 4. This arises partly because the Sarsa update is different, but mostly because Q-learning can update the previous Q-value before selecting a successor action while Sarsa cannot.)

3.4 Reusing experiment code

By now it is clear that a great many experiments and a great range of statistics are possible for one pairing of agent and environment, let alone a range of agents. Each experiment requires a variation on the code. Alfred can see that if each experiment is implemented independently (as in figures 1 to 4) he will soon have a lot of *code duplication*. Code duplication refers to having the same segment of code appear in different parts of a program (for example, the same segment of five lines might appear in different methods).

Although not all instances of code duplication are problematic, there are a number of reasons why code duplication in general is considered undesirable (Roy and Cordy, 2007; Koschke, 2007; Juergens et al, 2009). First, an update to one instance must usually be repeated for all instances, which costs time. Second, if updates are not made to all instances a bug may be introduced. Third, duplication makes code longer and hence more difficult to understand. Finally, a fourth and related problem is that it tends to make the code's purpose less clear. For example, the differences between Q-learning and Sarsa would be easier to understand if the duplicated parts were moved into subroutines. (We have shown the differences in bold in the Sarsa code.)

Code duplication is generally a sign that the software could be better designed, so Alfred considers whether any *code reuse* is possible. Although there are strong similarities between the experiments, there are also many differences. Furthermore, the differences occur throughout the code and interleave with the similarities, which makes code reuse somewhat more difficult. Alfred ponders how he can implement a set of related experiments, such as those in figures 1 to 4, in a way which exploits their similarities but allows arbitrary variations.

Code reuse is an important topic in software engineering and there are different approaches, but the basic idea is to factor out the duplicated code into *modules* which can be reused. "Module" is a generic term for some reusable segment of code. Different languages support different kinds of modules, and often have different names for them, including subroutines, functions, methods, classes, closures and S-expressions. Some are more powerful than others.

Software engineering note. A very basic way to reduce duplication is to move duplicated code into a method which can be called from different places. In many languages code can also be passed as a parameter to a method, which allows very flexible reuse. Object-oriented programming provides another form of reuse: we can factor the duplicated code into a class and then use inheritance to extend and specialise this class, producing alternate versions. Whatever the approach, the key point is that by *modularising* the code we make it more flexible and easier to reuse. This comes at a cost, however, as there is an overhead to defining the modules:

it makes the code longer and more complex. As an example, contrast i) putting an entire program in one method with ii) splitting it into several classes. The first approach does not scale well, but does have some advantages. For one, no code is needed to define the classes, their names, and how they relate to each other. For another, reading the code does not require understanding of how classes work (or inheritance, polymorphism, static and dynamic types, and so on). Nonetheless, except for very short programs, the benefits of modularising code (and other techniques for structuring it) tend to outweigh the costs.

Modularisation is one of the most fundamental and widely-used techniques in software engineering, but the difficulty of applying it appropriately and the complexity which can result should not be underestimated. We next look at some reasonably detailed examples in the hope they will stimulate the reader's intuition.

Alfred looks at the Q-learner in figure 1 again and considers how to modularise it. He realises that it already treats the environment as a separate module: it does not define the environment, it simply calls it as needed. In fact, the same Q-learning code could be used with different environments, as long as they all follow the same *protocol*; that is, use the same function names, and pass the same information back and forth in the same sequence. (We will discuss protocols in more detail in section 3.5.) Although this code does not define the environment, it defines the protocol that the environment must support in order for Q-learning to use it. This in turn implies that the environment must support learning from sample experience, as that is what Q-learning does. (Note that, because figures 1 to 4 are pseudocode, they do not actually specify function names.)

Alfred decides to design a new, modular system which will support code reuse. He decides he should begin with some software engineering. He writes a paragraph in English which specifies requirements for a Q-learning system and notes that it refers to an environment, a learning agent, and experiments. He decides that each of these will be a module, and, since he will be working in Java, each will be a class.

Looking at figure 2, Alfred decides that the added statistics-generating code in bold belongs in the experiment class since he would like to be able to alter it without touching the agent or environment classes. Figure 3 is a more complex case. It adds statistics-generating code to Q-learning but in the process also spreads the calculation of Q-values across two lines of code. Some of the code in bold therefore belongs in the experiment class but other parts belong in the agent. Sarsa, in Figure 4, is an interesting case. Although there are substantial differences from Q-learning, much of the code is nonetheless identical. Alfred realises that Q-learning and Sarsa are closely related. Both are one-step temporal difference control algorithms, and the code they share is common to this class of algorithm. Alfred makes a method called `episodicRL` with this code (figure 5). To further modularisation, he puts the initialisation code in other methods (`initialiseExperiment` and `initialiseEpisode`, neither of which are shown), and the details of the update in yet another method (`oneStep`). Each of these methods can be redefined as needed to produce variations. For example, figure 6 shows a version of `oneStep` suitable for Q-learning while figure 7 shows a version for Sarsa. The `oneStep` methods are themselves composed of other methods, which can in turn be redefined as needed.

```
initialiseExperiment()
Repeat (for each episode):
    initialiseEpisode()
    Repeat (for each step of episode):
        oneStep()
```

Fig. 5 The `episodicRL` method

a = chooseAction(s)
s', r = observeTransitionAndReward(s, a)
a^* = getMaxAction(s')
updateQValue(s, a, r, s', a^*)
$s = s'$

Fig. 6 A version of the `oneStep` method called by `episodicRL` (figure 5). This version is suitable for Q-learning. The initial s is set by `initialiseEpisode`

s', r = observeTransitionAndReward(s, a)
a' = chooseAction(s')
updateQValue(s, a, r, s', a')
$s = s', a = a'$

Fig. 7 A version of the `oneStep` method for Sarsa. The initial s and a are set by `initialiseEpisode`

Discussion. As the number of variations of Alfred's code increased, so did the need to manage them with modularisation and code reuse. (Note that the unmodular versions such as figure 2 are pseudocode and hence likely to be shorter and more manageable than the real versions.) Alfred achieved a substantial amount of code reuse using modularisation. For example, the `episodicRL` method in figure 5 can be used in all the experiments he has considered so far, and many more. However, modularisation also makes the code more complex (and hence harder to design and harder for someone else to understand). A key difficulty in producing an appropriate modularisation is producing a good model of the domain. In the case of RL, this includes an understanding of what types of agents, environments, and experiments need to be handled, and how they can vary. This is quite a challenge with a domain as complex as RL, as we discuss in section 5.5.

3.5 Same experiment, different protocols

Alfred does not think his Q-learner is learning as well as it should and suspects a bug. Luckily his friend Imi has also just implemented Q-learning based on the pseudocode by Sutton & Barto (1998) and in the same language as Alfred (Java), although she applied it to playing tic-tac-toe. She sends it to him for comparison. Alfred intends to interface Imi's Q-learner with his blackjack simulation to see whether it learns better than his Q-learner. However, he discovers they have chosen different function names and different function signatures (parameter lists); in other words their *protocols* – the specification of how to interact with the code – differs. Alfred realises that two

people may implement the very same pseudocode in incompatible ways. This is possible because the pseudocode is, by definition, not a fully detailed specification, and each implementor fills in the missing details as they see fit.

Alfred notes the protocols for the two systems he is working with are defined only by their code. Since neither Alfred nor Imi intended their code to interface with anyone else's, neither gave any consideration to the issue when implementing their system, and neither wrote any documentation on how to do so.[4]

Alfred sets out to write an *adapter* from Imi's protocol to his (see section 5.1). That is, he wants to write code which will sit between Imi's Q-learner and his blackjack simulator, and impersonate each to the other. That way, neither the simulator nor the Q-learner will need modification. However, he quickly realises that adapting one protocol to the other is not a simple matter. He finds the following incompatibilities:[5]

State representation.
: In Imi's tic-tac-toe system a state is a vector of characters (Java chars) describing the board. In contrast, in Alfred's blackjack system a state is an integer representing the sum of a set of cards.

Action representation.
: Imi's tic-tac-toe system represents actions as an array of two characters which specify a coordinate on the board. In contrast, blackjack represents actions with a Java enum (a nominal value).

Alfred expected he would need to adapt his Q-learner to the state and action representation used in tic-tac-toe; they are very different environments, after all. However, it occurs to him that some representations would be general enough for many environments. For example, both blackjack and tic-tac-toe states could be specified by an array of integers (of length 1 in the case of blackjack). Similarly, in both cases actions could be specified by an array of integers. In fact, real-valued arrays would also work for these two environments and are more general than integer arrays.[6]

Handling of terminal states.
: The value of a terminal state is 0 by definition and it does not therefore strictly need to be estimated by a Q-table. Imi's Q-learner detects terminal states by querying the tic-tac-toe environment and so avoids querying her Q-table about them. However, Alfred's Q-learner does not check whether states are terminal and its Q-table does maintain estimates for them (which are initialised to a default value of 0 and never updated).

Alfred wants to avoid modifying blackjack to answer queries on whether states are terminal or not, so he changes tack. Instead of applying Imi's Q-learner to his blackjack, he considers applying his Q-learner to her tic-tac-toe; if her Q-learner outperforms his it will suggest a bug in his version. However, he quickly runs into another problem.

[4] Software engineering note: in essence, private implementation details became incompatible public interfaces when Alfred tried to use the code in an unintended way.

[5] The following discusses two agents and two environments written by two people and it may be difficult to keep track of what each of them did. Fortunately, this is irrelevant: what matters is that the systems are incompatible in the ways outlined.

[6] WEKA represents all data, whether nominal, integer or real with an array of reals and a second array to specify each value's intended type.

Action space query.

Alfred's system assumes all states have the same number of actions whereas Imi's queries the environment. Each approach suits the problem addressed: in Sutton and Barto's simplified formulation of blackjack the player always either hits or sticks, whereas in tic-tac-toe, depending on the state, the player has a choice of between 1 and 9 empty positions in which to place their symbol. Alfred accepts that in order to play tic-tac-toe he will have to modify something. Perhaps his agent should have been designed to handle a variable number of actions from the start, since that is the more general case, but on the other hand, he initially only wanted to play blackjack.

State size query.

In order to allocate enough memory when initialising itself, Alfred's Q-learner asks his blackjack simulator how many state/action pairs are possible (by calling a method called `getMaxQTableSize` in the blackjack environment). In contrast, Imi's Q-learner uses a variable-size datastructure: when it is asked to read or write a Q-value it does not contain, it creates a new entry for the Q-value, which it initialises to a default value. Consequently, it does not need an estimate of the state/action space size at any point and Imi's tic-tac-toe environment does not provide this information.[7] Alfred has a choice between several changes which will allow him to play tic-tac-toe with his Q-learner. Alfred's first thought is to add `getMaxQTableSize` to the tic-tac-toe code. (Alfred is able to modify Imi's tic-tac-toe environment because she sent him the source code. In other circumstances, however, he might have only had a run-time library and not been able alter it.) However, recall that Alfred's Q-learner cannot communicate directly with Imi's tic-tac-toe because they use different function names; instead he will route communication through an adapter. Bypassing the adapter to call `getMaxQTableSize` does not seem like a clean design. Furthermore, adding the method to tic-tac-toe only solves the problem for this one environment; it is not a general solution. Effectively, it extends tic-tac-toe's protocol to be partly compatible with that of Alfred's Q-learner, but it does not change the protocol of any other environment Imi might write. Alfred's next thought is to add `getMaxQTableSize` to the adapter. This is better than Alfred's first idea because now his Q-learner does not need to communicate with tic-tac-toe directly, and because `getMaxQTableSize` is now available regardless of what environment is used with the adapter; it is a general solution. However, this overlooks the fact that `getMaxQTableSize` must return a value. Alfred could hard-code the correct bound on the Q-table size for tic-tac-toe into the adapter, but then the adapter would only work for tic-tac-toe, and no longer be a general solution. In software engineering terms, the adapter would be heavily *coupled* to tic-tac-toe.[8] Note that the adapter becomes coupled not just to the protocol tic-tac-toe uses but to the tic-tac-toe environment itself. Suppose Imi sends Alfred a mountain-car environment which follows the same protocol as her tic-tac-toe environment. Alfred's adapter class would not work correctly for the mountain car because it reports the state space for tic-tac-toe. Alternatively, suppose the tic-tac-toe environment supports

[7] It is easy to give an upper bound on the number of state/action pairs in tic-tac-toe based on the size of the board and number of symbols, but somewhat difficult to compute the exact number of possible state/action pairs since not all assignments of symbols to cells are possible. For example, a board with nine Xs cannot be produced by any tic-tac-toe game.

[8] Coupling refers to the degree to which two classes depend on each other's details. Good designs minimise coupling so that changing one class requires minimal changes to others.

different board sizes. How can the adapter handle that? The adapter class is not the right place for this information. Alfred could add `getMaxQTableSize` to both the adapter and tic-tac-toe, and have the adapter simply pass the call on to tic-tac-toe. Again, however, this is not a general solution inasmuch as it would need to be repeated to interface with any other environment following Imi's protocol. Finally, Alfred decides to extend his Q-learner to accept information on the size of the state space when it is initialised. This can be specified in the code which sets up the experiment, which means the Q-learner does not need to request it from the environment via `getMaxQTableSize`. Consequently, he does not need to add this method to the adapter or tic-tac-toe, and can even remove it from his blackjack environment. This simplifies his protocol and reduces the coupling between the Q-learner and environment. Note that all these solutions involved modifying either the learner's or the environment's protocol. If the only differences between protocols are the function names used, an adapter can translate between them. However, for more complex differences the addition of an adapter alone is not enough to bridge between protocols.

Alfred has stumbled across the fact that an RL system has not only a protocol for interaction between agent and environment, but also a protocol for interaction between the agent and its Q-function, and for that matter other components of the agent such as action selection methods (e.g. ε-greedy, softmax, ...).

Discussion. Alfred has noted the need for agents and environments to implement an agreed protocol and the diversity of protocols with which a given experiment can be implemented. He has gone from a system dedicated to one environment to one handling multiple environments, which revealed shortcomings in his original design, and has noted the need for general state and action representations. Perhaps the most striking problem was the difficulty of integrating systems which were not designed to work together. Alfred concludes that researchers who want to share code need to consider how to do so before writing it! The best and easiest way to share code is to implement the same protocol from the start. Since every RL system needs a protocol, choosing one is not extra work but a necessary step, and, indeed, adopting an existing protocol solves design problems. If Alfred and Imi had used, for example, RL-Glue, they would have avoided all the difficulties in this section, while actually making their design problems easier.

3.6 Same problem, different ways of formulating it

Alfred notices that, for any given blackjack state and action, it is quite easy to compute a distribution over successor states. For example, if the player has two cards summing to 16 and a new card is drawn, it is easy to determine the probability of each of the possible sums of three cards. Alfred decides to modify his Q-learner to exploit this information in the hope that it will learn more quickly. He modifies the blackjack environment to add a `getDistributions` method which returns three lists of equal length: the first lists all successors states which have non-zero probability, the second specifies the probability of each of these successors, and the third has the reward for transitioning to each successor.

On each time step, Q-learning updates a Q-value toward that of the single successor state. Now, however, Alfred's agent can update a Q-value toward all possible successors,

```
Initialise Q arbitrarily
Repeat (for each episode):
    Initialise s
    Repeat (for each step of episode):
        Repeat (for each action):
            getDistributions()
            Update Q(s, a) weighted by distribution over successors and rewards
        Choose a using policy derived from Q
        Take action a, observe r, s'
        s = s'
```

Fig. 8 Q-learning modified to perform a full backup on each action. Additions to standard Q-learning (figure 1) are in bold. Note that the reward r is not used and the chosen action a is not used in the update but only to advance the environment to the successor state

in proportion to their probability. In other words, it can make full backups rather than the usual sample backups (Sutton and Barto, 1998, section 9.5). Furthermore, he repeats the process for both possible actions in a state, although he still chooses only one action to send back to the environment so that it can advance itself to a successor state. His new agent is no longer a Q-learner; we shall call it a "full-backup agent". At some point it occurs to Alfred that he has partly reversed his decision in section 3.2 to use a blackjack simulation rather than to work with explicit probability distributions! He is running a simulation, but he now uses it to calculate some explicit probabilities on the fly. This seems like an improvement as it requires little change to the simulator, and yet provides more information to the learner. There is also a trade-off involved: he does waste time recomputing the same probabilities each time he revisits a state/action pair, but he avoids the memory overhead of storing the entire state transition matrix (which, as noted in section 3.2, is not an issue for blackjack but for many environments would be too large to store).

Complete pseudocode of the full backup agent in the format used in figure 1 is given in figure 8; note the extensive code duplication between these two figures. Thanks to Alfred's modular approach, however, to implement the changes in his learner he must only modify the `oneStep` method in figure 6 to produce the version in figure 9.

Software engineering note. Figures 8 and 9 encapsulate the pros and cons of code reuse. Figure 6 conveniently has all the code for an experiment in one place but much of it is duplicated elsewhere for other experiments (and this is pseudocode; real code would likely be longer). In contrast, in figure 9 the changes are localised to small modules (`oneStep` itself and `updateQValue` have been changed and `getDistributions` added). These modules can be reused by any experiment which uses this version of `oneStep`. However, the definition of the experiment is distributed among different modules, which makes it harder to understand the whole.

Alfred realises he has changed the protocol used by his code, that Imi's Q-learner cannot take advantage of this extra information, and that her tic-tac-toe environment does not provide it. However, he eventually also realises he has also changed the nature of the learning problem his agent faces. He is certain that making the distribution over successor states available affects the difficulty of the learning problem. An implication

```
Repeat (for each action a):
    successors, probabilities, rewards = getDistributions(s, a)
    updateQValue(s, a, successors, probabilities, rewards)
a = chooseAction(s)
s', r = observeTransitionAndReward(s, a)
s = s'
```

Fig. 9 A version of the `oneStep` method called by `episodicRL` in figure 5. This version makes full backups over all actions, unlike the Q-learning (figure 6) and Sarsa (figure 7) versions

is that it would not be comparing like with like to compare, for example, his full-backup agent to Sarsa. He suspects this sort of change may even alter the complexity class of some problems. He concludes that by adding the distributional information he has changed from one *problem formulation* to another. He thinks a general-purpose RL system should be able to handle different formulations of a problem. Alfred resolves to modularise the system by adding a formulation class.

Discussion. Alfred's change of problem formulation meant he had to change his code, just as in earlier examples of changing the statistics generated, the learning algorithm, and the protocol. This again complicates code reuse, but the same solutions apply.

Software engineering note. Figures 8 and 9 illustrate two approaches to implementing a new problem formulation. The approach in figure 9 is to extend the design of the software (in this case by adding a new version of `oneStep`) to accommodate the new formulation, which makes the system more complex. Although in this case the extension was simple, the existing design will often not support the desired extension and must first be refactored. (Refactoring is the process of implementing improvements to a system's design without altering its functionality.) The other approach is to fall back on the unmodular approach (at least to some extent), as in figure 8. This code is equivalent to the combined code from the various modules (figures 5 and 9). The advantage of writing it in an unmodular way is that it avoids the need to extend, and quite possibly refactor, the existing design. (To some extent the unmodular approach uses the rest of the system more as a library and less as a framework, since it takes the flow of control away from it; see section 5.2.) The unmodular approach allows the new functionality to be implemented in a quick-and-dirty way; it allows rapid prototyping by circumventing the constraints of the existing design. (This is particularly useful with strongly typed languages since the type system imposes strong constraints on how code can be used. It may also circumvent restrictions on accessing the private details of one class from another class.) Once the new functionality has been explored, however, if we want to reuse it, it will need to be modularised and incorporated into the design. We suspect a typical progression is therefore to implement the extension in an unmodular way, evaluate it, refactor the system, and finally to reimplement the extension in a modular way.

3.7 Conclusion of the story

Alfred found that implementing a single agent and environment was straightforward but as he varied the statistics he generated, changed learning agents, and eventually tried to get his system to interact with another system, the complexity of the software spiraled. We leave him at the point where he begins to consider what a general-purpose RL system would be like. No doubt many researchers have also reached that point and a few have progressed to implement such systems. In the next section we discuss some of the issues which this raises.

4 Requirements Analysis

In this section we cover the preliminary considerations needed to produce a design for an RL system. We are most concerned with general-purpose or broad RL systems (ones which handle a wide range of agents and environments) as they introduce the fullest set of requirements; less broad systems have a subset of these requirements. When we refer to "the system", we mean a hypothetical RL system of at least some breadth. Since any system should be designed with its users in mind, we begin in section 4.1 by considering potential users of the system. As an extension, in section 4.2, we cover the ways in which they may use the system. In section 4.3 we list requirements which aim to fulfill the users' core needs, and in 4.4 we discuss additional optional features. Finally, in section 4.5, we argue that analysis and design of RL systems can yield insights into the structure of the RL domain.

4.1 System users

We can identify three groups of users:

- Students of RL
- Those applying RL to real-world problem-solving
- Researchers

Each group is likely to have somewhat different requirements and designers must decide which to prioritise. We expect students and researchers to have a greater emphasis on flexibility and those solving large, real-world problems to have a greater emphasis on speed. It is unclear to us to what extent the two are mutually exclusive. However, it is possible for a slow-but-flexible system and a fast-but-inflexible system to be complementary. If the slow-but-flexible system is too slow for large, real-world problems, if may still be useful for rapidly prototyping solutions to them. For example, different algorithms could be quickly implemented, debugged, and evaluated on scaled-down versions of the problem, and once a solution has been found it could be reimplemented in the faster system. We can also distinguish between novice and advanced users and a system might cater to one more than the other.

4.2 Uses

In section 2.1 we listed things RL software can provide: ready-made agents and environments, an experimental platform, a protocol and reusable software components.

In this section we look at a related subject: general ways in which we may use such software. An RL system may be used:

- as a learning tool. This requires a shallow learning curve, good examples and easy ways to configure the system to sensible initial settings.
- as a solution to a problem. For example, the user may have a control problem and want to find a good control policy for it. The user may not really be interested in RL itself but just want to use it as a tool. If they know little about RL there is a greater need for ease of use. The system must also be easy to interface with the problem, and run fast enough to be useful.
- to compare results with a system written by the user. For example, the user may question whether their own RL system is implemented correctly. The system must be easy to use, and easily adapted, in case changes are needed (e.g. to generate different statistics, or run modified experiments).
- as a source of design ideas for a user's own system. A user may be implementing an RL system and be curious as to how other designers have solved similar problems.
- as a specification for an algorithm, such as Q-learning.
- as a supplement to the user's own RL system. For example, a researcher may want to compare their own agent to a set of others on a benchmark task, but have implemented neither the benchmark nor the other agents. Such users must be able to interface with the system easily.
- as the main platform for experimentation for an RL researcher. An ability to rapidly prototype new agents and experiments is likely to be an important requirement.

Each of the above would impose a different (though related) set of requirements. A system that aims to be used in all these ways would be more complex than one which attempts only a subset. It is quite possible that a set of systems, each aimed at a different use, would be more successful than one system which attempts everything. However, since they would be related they might be able to share code or design ideas.

4.3 Requirements

In this section we consider requirements for a broad RL system: things it should be able to do and qualities it should have. We give only one list of requirements for RL systems, regardless of their intended use (whether as a teaching tool, research platform, etc.), because it is possible for a given system to have multiple intended uses, and to give different priorities to these uses. Our list is consequently the union of the requirements for the uses we considered in the previous section. Where we write "the system should do X", it can be read as "other things being equal, it would be desirable for the system to do X". Designers are left to impose their own set of priorities on the requirements in accordance with the set of uses they have in mind.

- **The system should support good scientific research.** This has various implications. For example, the system should work as expected. If it implements Q-learning, the implementation should be bug-free, it should implement what is accepted as Q-learning, and it should document it adequately. The requirement for good science is an overarching one, and it implies the following requirements.
- **It should be possible to extend the system's breadth to match that of the underlying domain.** (The underlying domain being reinforcement learning,

or some subset or superset of it chosen by the designer; see section 5.6.) This extensibility is not just a bonus or a convenience for the end-user: as discussed in section 2.3, it is good science to have fine-grained control over experiments, to evaluate alternative approaches and to evaluate combinations of techniques; the RL system should facilitate all of this. Some particular examples of breadth follow.

  - **The system should allow a problem to be formulated in the ways allowed by the underlying domain.** For example, Alfred used both sample backups and full backups on blackjack. When different formulations are possible, the user should not be artificially restricted to a subset by the design of the system. When the system is not restricted in this way, we say it is *neutral regarding problem formulation*, or simply *formulation-neutral*.
  - **The system should not restrict the range of agents used.** For example, it should allow model-free, model-based, and policy search methods, but also, in the interests of good science, evolutionary algorithms and planning methods.
  - **The system should not restrict the range of function approximators which can be used.**

  Note that we do not require that the system implement all the variations, but only that they can be added. Furthermore:

  - **The system should be easy to extend.** In principle any system can be extended, but if the cost is too great it is not much use.

  In order to be extensible across the breadth of the domain:

  - **The system should have a good model of the underlying domain.** We discuss this in section 5.5.

- **The system should maximise interoperability of its components.** It is not enough for the system to be broad: the system's design should not unnecessarily restrict the ways in which components such as agents, environments and function approximators can be combined. For example, it should not be the case that a neural network-based Q-function can be used with a Q-learning agent but not a Sarsa agent. Similarly, it should not be the case that Q-learning can be applied to a particular environment, but that Sarsa cannot. In other words, we should be able to *mix-and-match* components. An extension is that:

  - **It should not only be possible to combine components, but easy**. The overhead of changing e.g. to a different (but already implemented) agent or environment should be minimal (it should not involve any programming). We could call this a *plug-and-play* requirement and it is an important part of *rapid prototyping* (discussed below).

  However, while we require that the design of the software does not unnecessarily restrict combination of component, the domain itself may restrict them. For example, policy iteration can only be used when the transition and reward functions are fully accessible, so it cannot be applied (directly) to learning from sample experience. Furthermore, since policy iteration solves the sequencing problem (determining the jointly optimal sequence of actions in an episode) it does not make sense to apply it to single step problems (those in which all episodes have length one, such as Bandit problems). A requirement is that:

  - **The system should clearly indicate which components can be combined, and how.** For example, although we cannot apply policy iteration directly to sample experience, we can learn approximate transition and reward functions from samples and then apply policy iteration to the approximations.

This requirement may be met entirely by good documentation, or the code itself may support it. For example, in a strongly-typed language the type system can allow the compiler to detect mismatches between agents and environments.

- **The system should be able to interact with agents, environments or function approximators which use different protocols.** For example, the system should be able to use RL-Glue agents and environments even if its own agents and environments do not communicate with each other using the RL-Glue protocol. If a system's design does not unnecessarily restrict the protocols with which it can easily be extended to interoperate, we say it is *protocol-neutral*, in analogy with formulation-neutrality introduced earlier in this section.
- **The system should allow rapid prototyping.** This is particularly important for more exploratory work, whether it is developing new algorithms or finding the most suitable algorithms and configurations. Rapid prototyping depends on the extensibility and interoperability discussed above.
- **The system should be easy to use.** This requirement touches on many of the others, and we have already mentioned aspects when discussing extensibility and interoperability, but there are other aspects, including:
  - **The system should have a reasonable learning curve for new users**, to avoid discouraging them.
  - **The system should be well-documented.** This includes clearly stating the system's capabilities and how to use them.

  Perhaps more than the other requirements, ease of use depends on good design, good coding, and good documentation. All are important when writing software for private use, and much more so when it is to be made public (Sonnenburg et al, 2007, section B.1). One way to measure ease of use is to ask how quickly a new user can get the system working. This depends on many things including the installation process, documentation, and design of the system.
- **The system should maximise run-time speed.** RL agents typically need a lot of experience to learn; for example, Sutton and Barto report an experiment with blackjack involving 500,000 games (Sutton and Barto, 1998). RL therefore tends to be compute-intensive, and, typically, the faster a system runs, the better. There are exceptions, however. For example, if the system is controlling a real robot in real time, it may not benefit from running faster than is needed to respond to inputs and output actions in real time. In most RL experiments the only limit on experience is the rate at which it can be generated and processed, i.e. the speed of the hardware and software, and the time available to the researcher. (This is in contrast to supervised learning, where we typically have a fixed amount of data at the outset.) At the same time, RL typically requires a great deal of computation because each experience typically gives an agent little new information and yet changes in the estimated value of one state may affect the value of many others. An RL system which is not fast enough for production use on a particular environment may nonetheless be useful for rapid prototyping. Flexible, extensible systems tend to run a more slowly than dedicated systems because of their extensive use of interfaces and decoupling of components (see section 5.1), but it is not clear how significant this effect is.
- **The system should be open-source.** To be widely adopted and extended it should be open-source. The copyrights on open-source systems typically stipulate that only other open-source systems can incorporate them, so only an open-source

system has full access to other open-source libraries. Many other advantages of open-source systems were noted in section 2.3.

Unfortunately some requirements conflict with others: e.g., as a system gains breadth it is likely to lose ease of use. There is therefore a Pareto front of solutions which make good trade-offs between breadth and ease of use. Since it is not possible for one system to fully meet all requirements, a given system must address only a subset, or give them varying degrees of emphasis. It is also possible that a set of systems complements each other: e.g., one is well-suited as a learning tool and another as a research platform.

Briefly turning to consider the current state of RL software, we would like to put some emphasis on problem-formulation-neutrality and protocol-neutrality, as they have not been discussed explicitly elsewhere to our knowledge, and they are an important aspect of breadth. We briefly note how to achieve them in section 5.1.

4.4 Optional features

In addition to the requirements above we may want some of the following features. It would be difficult, to say the least, to provide all in one system and this list is not meant to imply that that should be attempted. Its purpose is, rather, to illustrate some possibilities.

- **Stand-alone execution.** It is preferable for the system to run without the installation of any other software, for ease of use. If other libraries are needed for some functionality it would still be preferable for the system to be able to run without them, if it can provide any functionality on its own.
- **Use of external function approximator libraries.** Using external libraries reduces the burden on the RL system's developers and leverages the (possibly ongoing) efforts of another community of developers. For instances, the system can be designed to use the function approximators which come with WEKA, RapidMiner or another supervised learning system. There are drawbacks, however: dependency on another system is introduced, installation is more complex, and there may be a run-time penalty. Furthermore, very few of WEKA's function approximators are fully incremental, which limits their utility for RL, and other supervised libraries may be similar.
- **Data management.** Running sets of experiments, storing set-up files, and pre- and post-processing data are all important aspects of the experimental process. However, the RL system itself may not need to provide this functionality: a system such as WEKA or RapidMiner may be able to provide much or all of it.
- **Interfacing with an experiment database.** The results of an RL experiment are normally used only for the purpose for which they are generated. It is possible, however, to routinely collect the results of many experiments, perhaps from across a community, into a database which can then itself be the subject of data mining; this meta-learning can reveal new insights on e.g. how algorithms compare, or can be used to predict which algorithm to use for a given environment (Vanschoren, 2010). The RL-Logbook (section 2.2) is an example intended for RL.
- A **graphical user interface** for ease of use. In addition to standard features such as loading and saving files, a particularly useful feature is what me might call an **experiment inspector**. This would be a debugger-like feature which allows the user to step through the execution of an experiment interactively. If an RL

experiment consists of a sequence {state, action, reward...}, the user could advance to the next point in the sequence by clicking on a "forward" button, or return to the last point by clicking "backward", and at any point could inspect the value of variables such as state, action, and Q-function.

- An **embedded (or extension) language.** While interacting with an RL experiment through a GUI can be convenient, it does have limitations. Suppose a user has applied an agent to an environment and wants to know the policy in a certain subset of states. The user could step through many episodes, identify the relevant states by inspection, and note the policy, but this is obviously very tedious and it would be much better if the user could write code which identifies the relevant states and logs the policy in a file. One approach is for the user to modify the source code of the RL system, but this exposes the user to its full complexity, it may use an unfamiliar language, the user may need to recompile the whole system, and it may not be possible to make changes while the system is running. An alternative which avoids these problems is to embed an interpreter within the RL system so the user can write and interact with scripts while the RL system is running. This also reduces the significance of the choice of language for the underlying system. Embedded languages are increasingly common. For example, JavaScript code is widely embedded in HTML pages, macro scripts are embedded in spreadsheets, and the emacs text editor is written in lisp and can be extended with it. Scripting languages are typically better suited to rapid prototyping than compiled languages. Embedding them makes them even more so as it gives access to the system at run-time and allows the user to inspect the state of the system using the scripting language (e.g. printing out the value of variables). Interacting with the system in this way is particularly helpful when debugging. However, an embedded interpreted language will typically be slower than a compiled language. It may be useful to rapidly prototype and debug an extension in the embedded language and then reimplement it in a compiled language. In summary, given the great variety of possible RL experiments, end-users will need great flexibility in defining and manipulating experiments, flexibility which can only be achieved by writing code. An embedded language is a particularly powerful option.
- **A domain-specific language.** An excellent maxim is to always use the right tool for the job, and it may be that a language designed specifically for RL is better than any other. Unix shell scripts are well-known examples of domain-specific languages which are specialised, among other things, to manipulating files. A simple example for RL is to use a format like XML to specify experiment parameters (as MMLF does; see section 6.5). However, much more sophisticated languages are possible. In effect, the smaller modules Alfred created in section 3.4 define building blocks which he then combined into overall algorithms (such as figure 5). He worked in Java, a general-purpose language, but a domain-specific language could do the same and could be tailored to this particular task. A domain-specific language could even be used to evolve RL algorithms (with, for example, genetic programming).
- **A dataflow GUI.** A dataflow, or scientific workflow, is a form of domain-specific language (Johnston et al, 2004; McPhillips et al, 2009). Many such systems are visual programming languages and they are particularly convenient; they allow specification of programs by connecting icons in a GUI. WEKA's Knowledge Flow interface is one example. RL experiments are well-modeled as dataflow processes and this would allow non-programmers to use the system, and remove dependence on having a programming language installed. However, while dataflow GUIs are

very suitable for some tasks they are less so for others, and power users may not be fully satisfied with a dataflow GUI.

- **Access through a browser.** The easiest approach to installation is Java Web Start, or similar technology which allows a system to be installed directly from a browser. Alternatively, if the system ran inside a browser no installation would be needed. However, this would incur a run-time penalty and security restrictions limit the functionality of such systems. Another alternative is to run only the user interface in a browser, which allows the back-end to be located anywhere. For example, a group of users could share access to an RL engine running on a supercomputer and access it through their browsers. This also means researchers can be located anywhere, as long as they have access to a browser: they can check on the progress of an experiment by phone while in a pub, if they are so inclined. One RL system could have multiple interfaces (e.g. a browser, a non-browser GUI, a text interface) which might have more or less functionality or be more or less easy to use.[9]

4.5 Analysis and design can generate insight into RL

We believe that the process of designing a system able to handle diverse use cases will shed light on how different classes of environments are related. Implementing a system capable of running a *range* of problem formulations and learning agents is the most rigorous way to examine their relationships. The broader the system the more likely its implementation is to lead to new insights. A broad and flexible system would also make it easy to explore the mapping of problems and algorithms, in other words to learn which algorithms to apply to which problems. We discuss software as a model of the RL domain in section 5.5 and the appropriate scope of this model in section 5.6.

At a more detailed design level, designers will also encounter issues of how to implement RL algorithms efficiently; the intersection of algorithms research and RL may prove a fruitful area. For example, designers can consider what data structures are most suitable for implementing Q-tables.

5 Design Considerations

A complete design is outside our scope, as are detailed designs for parts of an RL system. Instead, we discuss some issues and ideas for design at a more abstract level. In other words, we discuss *how* RL software can meet the requirements in section 4.3. We are most concerned with broad systems, as they pose more design challenges. We will make some suggestions based on our own experience but different approaches are possible and programming styles are to some extent a matter of personal preference.

[9] In addition to a command line interface, WEKA has three graphical interfaces. The main interface, called the "explorer", gives interactive access to all of WEKA's features. The "knowledge flow" interface is specialised to allow users to specify dataflow processes by drawing diagrams. Finally, the "experimenter" automates complex experiments involving multiple datasets and algorithms. Although the same results can be obtained with the explorer, the experimenter can run experiments in the background or on remote machines, which makes it suitable when experiments are too slow to be interactive.

Section 5.1 very briefly introduces the concept of design patterns and mentions a few relevant patterns. Next, we consider types of systems: section 5.2 discusses the difference between frameworks and other libraries, while section 5.3 discusses compositional and integrated systems. In section 5.4 we discuss a technique for rapid prototyping in complex frameworks. We then switch focus in section 5.5 to discuss RL software as a model of the RL domain and explain why broad RL software must be complex. Given this, section 5.6 discusses where to limit the scope of RL software. Finally, section 5.7 argues that we have only begun to exploit the potential of adopting standards across RL systems.

Before proceeding, the benefits of *good software engineering* are worth emphasising, particularly as it is often neglected by scientists who are ultimately interested in results rather than the software which produces them.

5.1 Design patterns

A design pattern is a solution to a common design problem (Gamma et al, 1995; Shalloway and Trott, 2005). Patterns are tremendously useful in software engineering, but they are largely outside the scope of this work and we will mention only a few to illustrate their relevance to RL systems.

- The *adapter pattern* has an obvious application in adapting components which use different protocols (analogously to the adapters used to connect electrical items to sockets when traveling abroad). In section 3.5 Alfred wrote an adapter to connect his code to Imi's.
- There are a range of *creational patterns* which store information about relationships between components and help hide this complexity from programmers. For example, some objects may be typically created and initialised together and a class can be used to capture this information. As another example, the choice of one type of object may influence the choice of another: e.g., given an agent and environment with different protocols, a creational class could encode the knowledge of which adapter is needed to interface them.
- The *strategy* and *template method patterns* are useful for building a range of related algorithms out of simpler ones. This is typical of RL algorithms; see section 3.4.
- General state and action representations are possible, as noted in section 3.5. We have not seen this described as a design pattern though it could be taken as one. The RapidMiner documentation (Mierswa et al, 2006) uses the term *data transparency* to refer to the property whereby all components of a system know what to do with the data they receive (such as states, actions and rewards).
- The *interpreter pattern* can be used to implement domain-specific languages (section 4.4).
- The *interface pattern* (or *programming to interfaces*) decouples modules by routing communication through an interface.

Although the interface pattern is well-known, its significance, and the subtly of fully exploiting it, is such that it is difficult to overemphasise it. (See (Grand, 2002) for a brief treatment or (Barnes and Kölling, 2008) for a more tutorial one.) It is also particularly relevant: a protocol, both in the broad data communication sense and specifically as used within RL software, is an example of the interface pattern. RL-Glue is no more

or less than an implementation of a protocol. If Alfred and Imi had coded to the same interface their systems would have been interoperable from the start.

When programming a module to use an interface we are neutral to the choice of module on the other side of it. We can use interfaces to achieve neutrality not just at the code level, but for the functionality of the entire system. Hence, interfaces can help meet the requirements for protocol-neutrality and formulation-neutrality from section 4.3, or what we might call function-approximator-neutrality: the ability to substitute different function approximators within an agent.

5.2 Libraries vs. frameworks

A program is software that a user runs. A software *library* (or toolkit, in object-oriented terminology) is a collection of related reusable code that is used to write programs. A software *framework* is a kind of library where the emphasis is on reusing design, in addition to, or instead of, reusing code (Gamma et al, 1995).

A typical library might contain mathematical functions to calculate exponents, logs, trigonometric functions, and so on. The library does not impose a design on the programs which call it. Instead, a user designs a program and simply calls the library functions as needed. The role of the library is to provide reusable code which would otherwise need to be reimplemented by many programs involving mathematics.

In contrast, a framework *does* provide a design for the user's program. A framework can be seen as a meta-design, since it can be specialised by many users to produce many programs. All these programs will share the framework's architecture, by which we mean the overall design, including what modules exist, how they interact, and how the flow of control progresses through the system.

The flow of control is a key difference between frameworks and other libraries. With an ordinary library, the user's program defines the overall flow of control as it executes, and the library simply responds to calls by the program. But with a framework, control is inverted: the framework defines the flow and calls the user's code when it sees fit. This feature is referred to as *inversion of control*. Hence, the framework is, in a sense, the program, and the user simply extends it. To us, the distinction between libraries and frameworks is actually fuzzy, since it is possible to specify more or less of the flow.

In some cases, a framework focuses entirely on design and provides little or no code for reuse (other than the code which specifies the design). This is the case with RL-Glue, which we consider a framework, since it specifies an important part, though not all, of the flow of control of the systems which use it.[10] Since it does not provide reusable code for writing agents and environments, RL-Glue is not, in that sense, a library. If the terminology is confusing, think in terms of what a system provides: code reuse, design reuse, or both.

Interestingly, all the libraries we survey in section 6 are clearly frameworks, apart from the MATLAB MDP library. This is because they provide not only reusable code and a protocol, but everything else needed for complete runnable systems. Hence, by definition, they must specify the entire flow of control and be a framework. The exception to this rule, the MATLAB MDP library, is much simpler than the other

[10] RL-Glue is a protocol. In addition to specifying a set of methods, a protocol also specifies the sequence in which events occur (see section 2.1), and this sequencing defines part of the flow of control of the overall system.

systems and consists of a set of functions which can be called by the user's program. It does specify part of the flow of control, but not nearly as much as the others.

Frameworks solve major design decisions for their users and typically allow them to implement programs much more quickly than simply using libraries. An RL framework lets its users concentrate more on RL and less on software engineering.

Difficulty of design. Designing libraries is harder than designing programs in that a program solves one problem, but a library contributes to the solution of many problems. A library designer can make fewer simplifying assumptions about how it will be used than a program designer, and library code must therefore be more flexible. Since frameworks provide not only reusable code but reusable architectures, they are even harder to design than libraries. As Gamma et. al put it: "A framework designer gambles that one architecture will work for all applications in the domain" (1995, p. 27).

Ease of use. A library is also typically easier to use than a framework because it provides less; since a library is simpler there is less of a learning curve involved. The expertise and needs of the user are also factors. An expert is more likely to have their own working design and only want to reuse code, not a design, in which case a library may be more suitable. This is particularly true if the user wants to reuse the code in complex ways unintended by a framework's creators. In contrast, a novice may appreciate the greater structure provided by a framework and have less need to use it in obscure, unintended ways.

Backwards compatibility. Because a framework defines the architecture of a program, substantial changes to the framework are likely to have major effects on the program. For example, suppose a new version of a framework is released and that it has major differences from the last one. Any programs written using the last one will likely need substantial rewriting in order to become compatible with the new framework. In contrast, major changes to a library may require updates to programs which use them, but their overall architecture is unlikely to be affected. Therefore, libraries are likely to be more backwards compatible than frameworks.

5.3 Compositional vs. integrated systems

As noted in the last section, the full process of running an RL experiment may involve pre- and post-processing, visualisation, and so on, and these may be handled by different systems. Indeed, SL also needs such systems and many already exist, but even if they are written for RL, they may be independent of, yet complementary to, other RL projects. That is, it is possible for an RL system to be composed of many independent systems which communicate as needed through some interface. A simple example is that an RL experiment might write its data to a text file which is then read in by MATLAB, which plots it. Such *compositional systems* are widespread. (Indeed the Unix operating system is built around this idea, and contains many programs which communicate through text files, including sed, awk, and grep to name but three.) A more complex example follows. Researcher A interfaces his RL agent with an environment written by researcher B, using RL-Glue to connect them. A's agent uses neural networks provided by the Torch library (Collobert et al, 2002), its output is postprocessed by A's Perl scripts, and the result plotted with Gnuplot. Just as a library of

reusable software components makes it easier to build an RL agent, a complementary set of libraries covering different areas makes it easier to build a (compositional) RL system. This is a tremendous benefit, and most of the systems we survey have considerable compositionality. For example, MMLF and PyBrain use SciPy and NumPy for mathematics, MMLF uses QT for its graphics, and RLT, libpgrl, MMLF, and PyBrain all use external function approximation libraries.

Nonetheless, *integrated systems* have some advantages over compositional ones: installing multiple libraries is time-consuming, incompatibilities may arise when a library is updated, run-time efficiency may suffer as communication between the components is typically slower than in an integrated system, and generic libraries may not cater well to the particular needs of RL software. The success of integrated SL systems such as WEKA and RapidMiner suggests there may be a niche for integrated RL systems as well, although we believe integrated RL systems face more difficult design and implementation hurdles than integrated SL systems because RL itself is more complex than SL.[11] In conclusion, integrated systems are preferable if the cost of implementation time is no object. For those on a budget, however, compositional systems may be the only option.

5.4 Specialised and generalised components

When extending a system there may be a choice between rapidly implementing a specialised extension and a more time-consuming but more general implementation. For example, when adding an agent to a framework, we may be able to get a prototype working more quickly by making assumptions about how it will be used: perhaps we assume a particular environment and function approximator. To get the agent to work with all suitable environments or function approximators may take additional work. (This is similar to the initial unmodular and later modular extension discussed in section 3.6.) It is useful to be able to rapidly prototype components, whether they are later generalised or not. In the interest of ease of use, and to promote generalisation, the system should document the process, for example with a tutorial. This might include a guide to the interfaces which must be implemented, how to generalise handling of state, and reducing reliance on certain function approximators.

5.5 Model of the RL domain

In this section we explain why broad RL software is necessarily complex. In section 4.3 we noted that in order to be extensible across the breadth of the RL domain, the system needs a good model of it. Many programs model things in the real world; a traffic simulation might for example model cars, roads, junctions, bridges and so on. (It may also contain code which does not correspond to objects in the real world, such as for writing statistics to a log file.) Similarly, an RL system models agents, environments, and so on which fall within the scope of RL. To construct a sufficiently broad model requires an analysis of the scope of RL, and successfully implementing a broad RL system, and applying it to a wide range of RL problems, constitutes a

[11] In support of this claim we point out that SL does not involve the explore/exploit dilemma, RL environments typically require code whereas SL environments do not, and, finally, RL makes heavy use of SL function approximators.

validation of this analysis. We feel such an analysis is valuable in its own right (as discussed in section 4.5), as well as being a prerequisite for a broad RL system.

The imperative style of programming, used with languages such as Fortran, C and Pascal, specifies a solution to a problem as a series of instructions: do this, do that, then do this. In imperative programs, the domain model is represented largely by the choice of modules (functions) in the system.

In contrast, the object-oriented programming style, used with languages including Java, C++, and Smalltalk, models the domain more directly. It tends to represent classes of objects in the real world (such as cars and bridges) as classes within the program. The problem is then solved by passing messages between objects; for example, a car may tell a bridge that it is driving over it and the bridge may then update its internal state to determine whether to wobble or not. The difference between this approach and the imperative one is essentially that whereas an imperative program is dedicated to solving one problem, step by step, an object-oriented program builds a model of reusable components and then solves the problem by putting them together in a suitable way. Of course, an imperative language can be used to write reusable components (indeed, this may be the "problem" that we are trying to solve, if for example we are writing a library), but the difference is the emphasis on reuse, and the presence of techniques which support reuse, such as inheritance, in the object-oriented approach. Reuse is useful, because if we want to later solve a related problem, or, as often happens, if we change the requirements of our system (to upgrade it, or because they were incorrect to begin with), then we need only recombine the existing reusable components in a new way.

Using either programming style, an RL system encodes its model of the RL domain in part through its modularisation. Since RL is a complex domain, we expect an RL system which models it well to be complex, and to contain many modules. For example, Alfred had two versions of the `oneStep` method, one for Q-learning and one for Sarsa. Many other versions of this module would be needed for a comprehensive system. This complexity gives rise to what we called the *design hurdle* in section 2.4; the difficulty of designing a broad RL system. The hurdle is greater for libraries than protocols, and, as discussed in section 5.2, it is even greater for frameworks (as opposed to other libraries). Given the potential size of the hurdle, we must consider what the scope of an RL system should be.

5.6 Scope of an RL system

We argued in section 4.3 that good science required a broad system. In fact, we do not see a natural limit on the scope of an RL system. If it supports policy search, why not planning methods? If planning, why not evolutionary algorithms?[12] Ultimately, RL is situated within the broader field of Artificial Intelligence, and RL methods *should* be compared to those methods with which they can be compared.

Not only is RL situated within AI, AI is found within RL agents, in the sense that an RL agent is a complete problem-solving agent (Sutton and Barto, 1998). Learning from reinforcement is one technique an agent can use, but it need not be used exclusively: humans, for example, learn both from reinforcement and in other ways. Thus, RL and

[12] In fact, we have argued elsewhere that there is a close relationship between RL and evolutionary algorithms (Kovacs, 2004).

other methods may co-exist within an agent. Indeed, RL makes heavy use of Supervised Learning for function approximation. As another example, RL can sit atop a hierarchy of other decision-making systems and learn which to listen to.

It is possible for one software system to support RL and more, and, in fact, MMLF and PyBrain are mainly SL system with small RL extensions (section 6). Function approximators may be particularly suitable for sharing between RL and SL systems, although RL and SL have different requirements of function approximators (Kovacs and Kerber, 2006). Furthermore, both RL and SL involve, at an abstract level, computational experiments, which can involve a pipeline of events (pre- and postprocessing, feature selection, etc.) Both output data which must be postprocessed, organised, and exported to visualisation software. There is much scope for sharing of tools outside the core learning process. (Such tools fall under the heading of experimental platforms in section 2.1.)

Unsurprisingly, such broad systems would have great design and implementation hurdles. Worse, their complexity would give them steep learning curves and otherwise make them hard to use. If a well-designed, truly broad system was available, would anyone be able to use it? Or would it be too complex? Good documentation is necessary, but not sufficient, to maximise ease of use. (In our opinion this is demonstrated by RLT – see section 6.2.) In addition, design techniques for hiding complexity are needed (see section 5.1 for a few pointers).

Given the enormity of modelling all of AI with one software system, we must consider how to restrict a system to a manageable scope. Indeed, RL alone is complex enough that it is tempting to restrict the scope to a subset, for example including only agents which use value functions and omitting direct policy search. One point at which it seems natural to limit scope is the border between a protocol and a library of reusable software components (as distinguished in section 2.1). A protocol allows agents and environments to interact and is without doubt useful. A protocol by itself is not overly complex, but extending the scope of the system to add reusable components requires a leap in complexity.

The other points at which it seems natural to limit scope are the borders of RL proper and related areas. For example, the system might provide reusable components for other agent subsystems but not function approximators. This is quite reasonable given the range of function approximation libraries available.

Apart from the cases just mentioned, it is difficult to identify natural points at which to limit the scope of an RL system: the increase in complexity of accommodating one more problem formulation or one more kind of agent is not particularly great. Similarly, the savings in complexity from omitting one is not enormous either. Where should the line be drawn? All the systems surveyed in section 6 address a subset of RL. Designers of such systems need to consider carefully at the design stage how they will limit their scope and a detailed list of use cases may help them decide. Whatever the scope selected, a framework should be broad and flexible enough to model it well, particularly since backwards compatibility is an issue for frameworks (as noted in section 5.2).

It may be that a number of systems of relatively modest scope are needed to cover a domain as broad as RL. On the other hand, it may be that in years to come we will find systems with comprehensive support for both RL and SL (which MMLF and PyBrain already have, to a limited extent).

5.7 Adopting standards

Even though an integrated system may appear as such to the user, it is nonetheless good design practise to separate it into modules at the code level (such as user interface, function approximator code, and so on) with well-defined interfaces, and to program to the interfaces (section 5.1). Thus integrated systems are, internally, compositional. Consequently, both integrated and compositional systems need well-defined interfaces for their modules.

It is possible for all systems to define their own mutually-incompatible interfaces, but if they adopt the same interface they take a step toward interoperability. RL-Glue provides one kind of interface, and it allows diverse systems to interoperate at a certain level: their agents and environments can interoperate. However, they are not fully interoperable: it is not possible to plug the function approximator from one agent into another agent, because RL-Glue does not specify the interface between function approximator and the rest of the agent. However, we anticipate no reason why the agent/function approximator interface could not become standardised, which would allow us to mix-and-match not only agents and environments but also function approximators.

We feel the agent/environment and agent/function approximator interfaces are the most important, but various other interfaces could become standardised between RL systems, such as the interface to the action selection methods, or the interface to an experiment database. In addition, a given domain-specific language might become a standard. It is difficult for the designer of an interface to anticipate the needs of all its eventual users (that is, to make it broad enough), and interfaces will likely need to evolve as requirements are discovered. As noted in (Sonnenburg et al, 2007), a de facto standard may emerge over time but it is also possible that a committee may be formed to propose a standard. There are many examples of both in computer science.

6 Survey of Existing RL Systems

In section 2.2 we briefly discussed currently available RL software and we now review individual systems in detail. We consider only general RL systems (those which are designed to be extensible), and exclude programs such as Smarts (Gasser, 2008) which, while an interesting application of RL for Artificial Life, is an executable.[13] Other systems were not included either because they are no longer available, such as the Reinforcement Learning Toolkit,[14] or they have been out of development for more than five years, such as CLSquare (Riedmiller et al, 2006)).

While there are websites that link to numerous RL libraries, there are no comparative reviews like this one to provide recommendations. However, as our review is a snap-shot of what is currently available, it will go out of date. It would be helpful for online resources such as the Machine Learning Open-Source Software site `mloss.org` to encourage online reviews of RL systems.

Table 1 lists the RL systems we review and compares their functionality. The leftmost column lists features while each of the remaining columns list the values of

[13] Smarts does, however, deserve an honourable mention as it allows comparison of RL and policy search on the same environment via a GUI, which is useful for educational purposes.

[14] See `http://rlai.cs.ualberta.ca/RLAI/RLtoolkit/RLtoolkit1.0.html`.

those features for a particular system. Unlike the others, RL-Glue provides only a protocol and not implementations of agents. Therefore, RL-GLue's column indicates whether or not it allows (rather than implements) various kinds of agents; for example, it does not provide any POMDP environments or agents but its protocol allows agents to interact with POMDPs.

Before considering each system independently, let us examine their striking similarities. We can see that the systems have one thing in common: they are all open-source, which is good for the RL community as noted in section 2.3. Other than the RL-Glue protocol, all the systems are libraries or frameworks that implement model-free RL algorithms as a minimum and all but one provide some form of neural network for function approximation.[15] Apart from the MATLAB MDP Toolbox they define agents and environments separately, providing explicit protocols for communication between them.

In the following we will review each of the systems in table 1, giving each a score out of ten on: project activity, ease of installation and use, platform independence, inclusion of examples, documentation, provision of algorithms, and provision of function approximators. Each system was installed on MS Windows and Mac OS X. We have attempted to estimate the activity of each project based on recent updates. It is worth noting that, according to `mloss.org`, even libpgrl, which has had no updates since 2007, is still downloaded about once a day.

To evaluate ease of use we looked at documentation, tutorials and examples. Extensibility was evaluated by looking at the design, but not by implementing code. The systems are presented in no particular order; we considered ranking them according to an aggregate score but there was no obvious way to weight the scores. At the end of this section we make some recommendations based on our discussion of requirements.

6.1 RL-Glue

The first system is the odd one out: RL-Glue (Tanner and White, 2009) is the only system here that implements not a single learning algorithm or environment. It therefore occupies a different niche from the other systems we survey, which all provide reusable code for writing agents and environments. What RL-Glue does provide is a protocol that allows agents and environments to communicate and, consequently, a framework which defines much of the flow of control of an RL system which uses it.

> "We can trace RL-Glue back as far as 1996 through a project by Rich Sutton and Juan Carlos Santamaria called RL-Interface. Since then, the project has gone through several designs and languages. Over time the objectives of the project became more ambitious - it grew from being a convenient calling convention within a single language to a complete protocol allowing all sorts of various languages to communicate with each other." (Tanner, 2009a)

RL-Glue is useful in two ways. First, it solves the design problem of allowing agents and environments to communicate. Second, and more importantly, any agent or environment using the RL-Glue protocol can communicate with any other. This second form of utility depends on the number agents and environments implemented in RL-Glue: as more systems use it, it becomes more useful to adhere to it. Hence, the value

[15] The MATLAB MDP Toolbox has no neural function approximators, although other MATLAB libraries provide them.

Feature \ System:	RL-Glue	RLT	libpgrl	PIQLE	MMLF	PyBrain	MDP Toolbox	Connectionist Q
Version	3.04	2.0	126	2	0.9.8	0.3	3.0	1.0.1
Language	Multiple[1]	C++	C++	Java	Python	Python	Matlab/ Scilab	Java
Is open source?	y	y	y	y	y	y	y[5]	y
Supports RL-Glue Interface?	y	n	n	y	n	y	n	n
Has GUI?	n[2]	n	n	y	y	y	n	y
Has model-free agents?	allows	y	y	y	y	y	y	y
Has model-based agents?	allows	y	n	y	y	n	y	n
Has policy search agents?	allows	y	y	n	y	y	n	n
Has POMDP agents/envs?	allows	y	y	n	y[4]	y	n	n
Has hierarchical RL agents?	allows	y	n	n	n	n	n	n
Allows multiple agents in an env?	n	y	y	y[3]	n	n	n	n
Most recent activity	2010	2006	2007	2009	2010	2010	2009	2008
Project activity score / 10	8	1	4	4	10	10	6	3
Installation score / 10	10	2	4	9	6	8	7	7
Platform-independence score / 10	10	4	6	10	10	10	9	10
Examples score / 10	4	7	4	8	8	9	3	6
Documentation score / 10	10	8	6	8	7	10	5	5
Algorithms score / 10	0	10	8	6	8	8	4	2
Function approximation score / 10	0	10	6	6	9	6	0 (8)[6]	4

Table Notes:
1. RL-Glue has socket interface codecs in a number of languages and native libraries for C/C++ and Java
2. RL-viz is a work in progress
3. Two-player games only
4. Only a single environment has hidden state, and no agent specialises in solving POMDPs
5. Some functionality requires Matlab, a commercial software package
6. The toolbox has no function approximators of its own, but matlab has other libraries, which we have given a score of 8

Table 1 Comparison of RL Systems

of RL protocols, like social network sites or operating systems, depends largely on their market share (and less on any intrinsic qualities).

RL-Glue is perhaps the most well-known RL framework, for a number of reasons. Researchers can work in the language of their choice (see below), and are free to design and implement their own agents and environments. Compared to the other systems reviewed here, it is light-weight and simple. Finally, it has been the protocol specified by the (mostly) annual RL competition workshops held at ICML or NIPS.

RL-Glue has been ported to a number of languages including native C/C++/Java and Matlab/Python/lisp via sockets. The fact that RL-Glue supports communication through sockets is a useful feature: although sockets are slower than within-process communication, it means modules can be written in any language, as long as an RL-Glue codec has been written for that language. In this sense RL-Glue is *language-neutral*. This language-neutrality is easier to achieve with a protocol-only system such as RL-Glue than it would be with a library of reusable components, which are more tightly integrated. Nonetheless, other systems could provide for communication through sockets and achieve the same language-neutrality.

We see two significant limits to the scope of RL-Glue. First, it only applies to single-agent, sample-based environments (it does not, for example, support distributions over successor states or handle multiple agents within the same environment). Second, it only specifies communication between agents and environments, and not, e.g., between agents and value functions. However, we must emphasise that these are not problems with RL-Glue itself, but simply things which are beyond its scope, or more precisely its current scope. As noted in the quote above, RL-Glue has already grown in scope and it could be extended to handle cases other than single-agent, sample-based environments, or to specify communication within an agent. Equivalently, a new protocol which complements RL-Glue could be created to specify only within-agent communication.

Also notable is the way in which RL-Glue has been used to date. While the RL-Glue project does not aim to provide reusable components of agents and environments, there is nothing preventing the creation of a library which uses RL-Glue's protocol. Alternatively, libraries could provide adapters for RL-Glue, as PIQLE does. We believe both would be valuable.

In summary, RL-Glue plays a useful role and we feel there is scope to expand this role. We will be watching RL-Glue with interest.

Project Activity 8/10 With a library we would hope for continuous improvements, such as the addition of new agents. As RL-Glue is primarily a protocol, however, we hope rather for stability, or a limited number of backwards-compatible extensions. Although there has not been any update to code or documentation since 2009 we have given 8/10 for activity, partly in the expectation of future RL competitions, which have thus far driven activity on this project.

Installation 10/10 Installation is quick and easy as it uses up-to-date install/uninstall wizards on all major platforms.

Platform Independence 10/10 RL-Glue is tested under Windows, Linux and Mac, as well as providing source code for any other platform.

Examples 4/10 The two simple examples included are easy to compile and are reasonably well-documented. While examples might be considered less important given that this is primarily an interface, it is not possible to give it a high score when they are small and so few.

Documentation 10/10 This is a very well-documented project, which was no doubt made easier by its relative simplicity. As an interface, documentation is a point at which it needs to excel, as, hopefully, a wide variety of people will use it.

6.2 Reinforcement Learning Toolbox (RLT)

RLT (Neumann, 2005) is the most complete system, covering a range of different agents and variations such as partial observability, hierarchical tasks and multiple agents.[16] Part of this flexibility is made possible by feature subsumption: MDPs are a specialisation of SMDPs, single agents are a specialisation of multiple agents, concrete actions are a specialisation of a system that includes hierarchical options, and observable states are a specialisation of partially observable ones.

The design of the system provides a good model of the RL domain. To give an example, a class can listen in on the trajectories of other learners or policies as they act, allowing for imitation learning. There are classes for recording these trajectories for batch learning or just to store. By providing these and other options all in one system, the system reveals possible approaches that one may not have thought of for solving a particular problem, and allows comparisons with a broad range of approaches.

This flexibility is not without its price: all these variations make its structure very complex. Thus the learning curve is very high and the documentation, though extensive, is not enough to allow complete understanding without looking at the API,[17] which is itself hard to understand due to the system's complexity. Another issue this raises is one of scripting. Several systems we review are either in a scripting language (Python or MATLAB), or have a scripting interface. However, we found that the extensive use of multiple inheritance in RLT makes adding a scripting interface difficult for a language like Python and impossible for a single-inheritance language like Java.

Project Activity 1/10 There has been no activity on this system since 2006, which is a shame as it is quite comprehensive and allows direct comparisons of many different algorithms and components thereof.

Installation 2/10 A 200MB library has to be installed just to get the build tool (the qmake build maker from the Qt framework). In order to get this working without qmake we had to rework the make process to use a more up-to-date make system. On the plus side, the external libraries are bundled with the system, though one of these had been adapted to use the qmake build system too.

Platform Independence 4/10 While only available as source code, there isn't any system-specific code to cause problems. However, RLT's author only tested it under Linux.

[16] RLT should not be confused with the now unavailable Reinforcement Learning Toolkit developed in Alberta `http://rlai.cs.ualberta.ca/RLAI/RLtoolkit/RLtoolkit1.0.html`.

[17] Application Programmer's Interface: a low-level documentation of the source code.

Examples 7/10 A good range of examples: discrete, continuous and hierarchical problems with several solution methods. As above, compiling these examples without qmake is a trial.

Documentation 8/10 There is extensive documentation in the form of a user guide (an MSc thesis) and API. Example tutorials on the website help ease the otherwise steep learning curve.

Algorithms 10/10 RLT has the most extensive set of algorithms of any system we review, including: model-free, model-based, policy-search, actor-critic, residual, look-ahead, hierarchical RL and dynamic programming algorithms. It is able to handle continuous and heterogeneous time, partial observability, imitation learning, and multiple agents.

Function Approximation 10/10 RLT uses the Torch library to provide function approximators including neural networks, tile coding, RBF networks, k-nearest neighbours, linear approximators, and regression trees.

6.3 Libpgrl

Libpgrl (Aberdeen et al, 2006) initially provided policy gradient algorithms for a planner, but now implements both model-free RL and policy search algorithms, though not any model-based learning. Libpgrl aims to be efficient in a distributed RL environment, something the other systems do not concentrate on. While there are similarities with RLT, in that libpgrl is a fast C++ implementation that has abstract classes to model a subset of RL, Libpgrl's subset is much smaller. Consequently libpgrl has a proportionally less complex structure.

Project Activity 4/10 While there have not been updates to the library itself in recent years, the stochastic planner that uses Libpgrl has been updated more recently.

Installation 4/10 Libpgrl requires the external uBlas library which is only available as part of the large Boost package of C++ libraries (which is not included with libpgrl). Due to its age, some code editing was needed to make libpgrl work with the current Boost library and compiler, and its Makefiles had to be edited manually.

Platform Independence 6/10 Libpgrl's authors tested it on most popular operating systems, though as is common, windows requires cygwin/gnu.

Examples 4/10 Only one minimal toy problem on which Q-learning and Sarsa are run with different exploration approaches, though there are a seven POMDP examples. None of them are well documented; their use must be gleaned by diving into the code comments.

Documentation 6/10 A long readme file gives a good amount of information, though much of it would be better if merged into a more comprehensive API. The tutorial and template example help improve the learning curve greatly.

Algorithms 8/10 As befits a library originally designed to implement policy gradient algorithms, the policy search algorithms are the most developed, with the more recent addition of the model-free Q-learning, Sarsa and Natural Actor-Critic algorithms. There are predictive state representation and hidden Markov model approaches to solving POMDPs.

Function Approximation 6/10 Approaches include predictive state representation and neural networks.

6.4 Platform Implementing Q-LEarning (PIQLE)

PIQLE (Comité and Delepoulle, 2005) is the most flexible Java library in our list, although as it is common to use Java for teaching purposes (Kerr et al, 2003; Roberts, 2004), it is likely that other unpublished Java RL libraries exist. The object-oriented design shares many similarities with RLT, but does not implement as wide a range of algorithms, environments or function approximators. It does, however, include an RL-Glue interface as it was used in the ICML 2008 RL competition. For this competition it implemented a multi-agent system (swarm) of agents, which is functionality that none of the other systems can boast at present. PIQLE's authors are to be commended for publishing the source code used in a competition.

Project Activity 4/10 There have been no major updates since 2007 and no activity at all since the update for the 2008 RL competition added an RL-Glue interface.

Installation 9/10 As only pure Java and no libraries are used, the installation is quite easy on any system.

Platform Independence 10/10 Java is platform-independent.

Examples 8/10 There are many examples ranging from a simple maze to the more complex multi-agent missionaries and continuous Acrobot. GUIs exist for some examples, making this suitable as a learning tool.

Documentation 8/10 PIQLE has a Javadoc API and a more descriptive guide. The latter links RL theory to the API using many diagrams and two detailed case-studies.

Algorithms 6/10 The algorithms are mostly variations on Q-learning, including a batch neural network estimator.

Function Approximation 6/10 Neural networks and tile coding.

6.5 Maja Machine Learning Framework (MMLF)

MMLF (Edgington, 2009) comes close to RLT in its scope and implementation. Table 1 shows that both include model-free, model-based and direct policy search methods, thus fitting the requirements in section 4.3 for including multiple problem formulations, as these three approaches each require different forms of interaction between agent and environment. MMLF is not limited to RL, and the system is written flexibly enough that many of the non-RL problem formulations and algorithms can be applied to RL.

The downside to this flexibility is verbosity, which MMLF partially overcomes by using an XML format to define agents and environments, with a YAML[18] format for experiments. These files make it simple to run different experiments from the command-line without changing code. This simplifies automated testing, including on clusters, where each experiment can be run as an independent process. An example "world" (their term for the combination of agent and environment) looks as follows:

```
<world worldname="mountain_car">
 <agents>
  <agent agentmodulename="td_lambda_agent">
   <configDict gamma = "1.0"
               epsilon = "0.01"
               lambda = "0.95"
               minTraceValue = "0.5"
               defaultStateDimDiscretizations = "9"
               defaultActionDimDiscretizations = "7"
               update_rule = "'SARSA'"
               function_approximator =  "{'name': 'CMAC',
                   'learning_rate': 0.5,
                   'update_rule': 'exaggerator',
                   'number_of_tilings': 10}"
               policyLogFrequency = "5000"
               plotting = "{'frequency': 1000,
                   'plotStateDims': ['position','velocity'],
                   'plotActions': None,
                   'interactive' : False}"
   />
  </agent>
 </agents>
 <environment environmentxmlfile="env.xml"/>
</world>
```

Project Activity 10/10 At the time of writing this is one of only two projects with very recent activity, including code updates.

Installation 6/10 On the plus side, installation can be as easy as untarring the archive. However, it also requires installation of many libraries, which is left to the user to discover after installation.

Platform Independence 10/10 Python is platform-independent.

[18] A "human friendly data serialization standard" (Ben-Kiki et al, 2009).

Examples 8/10 Discrete and continuous examples, both fully and partially observable. Most show at least two agents for the task, highlighting the ease of applying agents to tasks.

Documentation 7/10 Fairly good documentation with a very good API. Not enough installation information, and somewhat lacking on introduction and tutorials, though the GUI should make initial experiments and exploration easier.

Algorithms 8/10 Well-organised sets of encodings as well as model-based, model-free and direct policy-search learners. There is some support for hidden state, but no specific algorithms for solving POMDPs.

Function Approximation 9/10 These include neural networks, tile coding, RBF networks, k-nearest neighbours, linear approximators, and CMAC.

6.6 PyBrain

PyBrain (Schaul et al, 2010), like MMLF, is a more general Machine Learning library that also includes RL. As the name suggests, it specialises in neural networks, both feed-forward and recurrent networks, which can be used to solve RL directly via optimisation, or as function approximators. PyBrain implements its own neural networks as part of an integrated system, which means they are better integrated, requiring fewer lines of code (ignoring differences in languages) to create and initialise, than in a compositional system like RLT (which uses the Torch library). The current range of algorithms and environments is limited, but unlike some of the older libraries (RLT, PIQLE, CLS^2) PyBrain appears to be under active development and so may be expanded in the future.

Project Activity 10/10 This is one of only two projects with very recent activity, including code updates, at the time of writing.

Installation 8/10 Requires installation of a number of libraries, though most are quite standard. The installation could be simplified by the inclusion of the libraries when downloading the system.

Platform Independence 10/10 Python is platform-independent.

Examples 9/10 There is an excellent set of examples. In addition to some standard tabular approaches and direct policy search, there is a range of interesting continuous problems with up to 11 degrees of freedom solved with neural networks. However, hierarchical and multi-agent problems are not included.

Documentation 10/10 Very clear quick-start and tutorials plus the examples provided make it very easy to jump into this library. The API is extensive and clear enough to allow the rest of the system's complexity to be exposed once needed.

Algorithms 8/10 PyBrain does not have a large number of RL algorithms; just the usual variations of Q-learning. PyBrain's strength is that it is part of a wider supervised and semi-supervised learning library, and so supports direct policy search via optimisation or evolution, Support Vector Machines and neural networks.

Function Approximation 6/10 Currently neural networks only, but functionality is being added to PyBrain.

6.7 MATLAB MDP Toolbox

While MATLAB is a scripting language, the existence of an (albeit limited) RL library allows it to be considered a complete RL system for the purposes of this review. The MDP Toolbox (Chadès et al, 2009) implements only a few basic algorithms such as tabular Q-learning, Sarsa and dynamic programming. However, the extensive MATLAB libraries, which include planning, neural networks, linear algebra and regression trees, can all be used in RL agents. These libraries are related by their systemic use of matrices to represent data and the shared MATLAB scripting language, which allows rapid prototyping. The issues involved in using MATLAB for empirical RL are the downsides of its flexibility and wealth of libraries: it is only weakly typed, so that relationships between libraries have to be known a priori as the possible uses are not specified by types, and MATLAB is a commercial system, so it cannot be used as a ubiquitous platform by the community. A version has also been released for Scilab, a free alternative to MATLAB, but it does not have the same breadth of libraries.

Project Activity 6/10 Major update in 2009, but such a small project could easily stall with the loss of a single person. At the point of writing, bugs flagged up 5 months ago have not had anyone assigned to them.

Installation 7/10 There are no instructions at all, but as there are only source files and knowledge of MATLAB or Scilab is a requirement, this is not a major problem.

Platform Independence 9/10 Platform independence is as good as the system this library is used with: MATLAB is available and stable on all platforms, as Scilab also claims to be (though we had difficulty installing Scilab under OS X).

Examples 3/10 There are two minimal test examples and one larger race-track example (which only works in MATLAB).

Documentation 5/10 There is an adequate readme file and API for such a simple system. For French speakers, there is a lengthy report on the race-track example.

Algorithms 4/10 Minimal (Q learning, value iteration and policy iteration) but other MATLAB libraries (e.g. for non-linear optimisation) could relatively easily be adapted for comparison (in this example for policy search).

Function Approximation 0/10 or 8/10 None included, but MATLAB/Scilab have numerous function approximation libraries available, though the code needed to use them would have to be worked out in each case.

6.8 Connectionist Q-Learning (QCON)

QCON (Kapusta, 2005), which has a very basic structure of three classes, only implements a version of Q-learning with a neural network as function approximation. It is, however, extensible in the sense that its environments are modular. Thus, new environments can be written, and the current environments could be adapted to work with other systems. It is the extensibility combined with informative and attractive GUIs that warrant this system's inclusion in this review.

Project Activity 3/10 This is clearly the work of an single individual. The latest activity was in 2008, though as that was after a three year break there is always the possibility of more work in the future.

Installation 7/10 The system is pure Java and has no external libraries, though there are virtually no instructions as to how to install.

Platform Independence 10/10 Java is platform-independent.

Examples 6/10 Although there are only two RL examples, they are well documented with animated GUIs that could be useful in demonstrating RL visually.

Documentation 5/10 Very little documentation, though the tutorial gives a good concrete example. The API is reasonable.

Algorithms 2/10 A single connectionist Q-learning algorithm makes this virtually not a library at all, more of an application.

Function Approximation 4/10 A neural network is the only form available.

6.9 Recommendations on the systems surveyed

For those who want to implement their own agents and environments, RL-Glue is the obvious choice. It is the only system designed for this purpose, and it is simpler than the others because it attempts less.

For those who want to modify or build on top of reusable agents and environments the choice is less clear. All the systems have their strengths and weaknesses, but overall we recommend MMLF or PyBrain. Both are under active development and broad in scope, being general Python ML libraries. MMLF implements the largest variety of algorithms after RLT, but is easier to install and extend, making it ideal for comparisons. PyBrain's algorithms and function approximators are a little limited, but it is the most usable system: it is one of the easiest to install and it has excellent documentation, examples and tutorials, which make the learning curve shallower than with the other systems.

If those systems are not broad or fast enough, and if the user succeeds in installing it, RLT is fast and the most broad system, making it a good candidate for a suite of comparisons of existing algorithms. It is not the easiest to extend and has no scripting interface, but the wealth of existing algorithms and richness of the structure mean that

it may be worth the overhead of extending it. Libpgrl is designed for speed rather than usability, but is more recent and simpler and therefore easier to understand and to install than RLT.

PIQLE is a good choice for Java users, and is the easiest to get working. Along with good examples, GUIs and an RL-Glue interface, we recommend it as a good choice for an RL course. The other systems each have particular strengths that might overcome their weaknesses if a feature was particularly needed:

- QCon has the most appealing example GUIs.
- MDP Toolbox is the only MATLAB/Scilab library, and likely very convenient for problems which are well-represented as a set of matrices.

6.10 A note on installation

There are a number of factors that determine how easy a system is to install: libraries and build tools that might have to be found and in the right versions, amount of manual configuration required, and clarity of instructions. Many of the systems rely on external libraries. While it does make sense to reuse existing specialised systems, notably for function approximation, reliance on external systems makes installation harder.

Both the C++ systems (RLT and libpgrl) needed manual editing to get them to work. The newer the system, the less work was needed. This is one of the issues with compiled systems that are no longer maintained: while a scripting language will probably stay reasonably usable, build tools and libraries move on and eventually are no longer backward compatible. Compare this to, for example, QCon, which despite having not been updated in several years still installs easily because it is written in Java, which was designed to be much more platform-independent than traditionally compiled languages. The same would hold true for the Python systems, but they are more recent in any case. None of the reviewed systems used the function approximators in RapidMiner, WEKA or the Modular Toolkit for Data Processing (Zito et al, 2008).

7 Conclusions

Science is founded on replicability. Having access to the source code used by others is invaluable and makes for better science. Having a broad library of reusable code is even better as it also allows fine control over experiments, encourages comparisons, and, more generally, makes it easier to explore the space of agents, environments and experiments (section 2.3). This gives rise to our requirement (section 4.3) that it should be possible to extend the breadth of a system to match that of the underlying domain. In particular, systems should be protocol-neutral and problem-formulation-neutral: that is, their design should not unnecessarily restrict the ways in which a problem can be formulated, or the protocols which can be supported.

RL is a very complex domain, with complex requirements, and very complex software would be needed to model the whole of it. Indeed, we argued in section 5.6 there are few natural points at which to limit the scope of an RL system, and that software may usefully cross the border between RL and other parts of artificial intelligence.

It is difficult to predict what sort of RL systems will emerge in the future, or indeed what kinds are most appropriate. Is a single, comprehensive, integrated system like WEKA realistic? Or will RL's complexity mean that multiple, complementary systems

are needed? Are integrated or compositional systems more suitable? We suspect the only way to find out is for researchers to implement the alternatives. Regardless of the approach, we feel that moving toward standard interfaces (section 5.7) will benefit the field. We conclude with three recommendations drawn from our discussions:

- Broad RL systems should be formulation-neutral and and protocol-neutral.
- RL software should use or provide libraries of reusable code.
- The field should converge on standards.

Acknowledgements We thank David Kirchheimer, Narayanan Edakunni and Michael Meadows for discussions, our three anonymous reviewers for their careful reviews, and the editors Shimon Whiteson and Michael Littman for their patience and encouragement.

References

Aberdeen D, Buffet O, Selmi-Dei FP, Zhang X, Lopes T (2006) libpgrl, URL `http://code.google.com/p/libpgrl/` 38

Asuncion A, Newman D (2010) UCI machine learning repository. URL `http://www.ics.uci.edu/~mlearn/MLRepository.html` 9

Barnes DJ, Kölling M (2008) Objects First with Java - A Practical Introduction Using BlueJ, 4th edn. Prentice Hall / Pearson Education 27

Ben-Kiki O, Evans C, döt Net I (2009) YAML Ain't Markup Language (YAML^{TM}) Version 1.2. URL `http://www.yaml.org/spec/1.2/spec.html` 40

Bouckaert RR, Frank E, Hall MA, Holmes G, Pfahringer B, Reutemann P, Witten IH (2010) WEKA-Experiences with a Java Open-Source Project. JMLR 11:2533–2541 4

Chadès I, Cros MJ, Garcia F, Sabbadin R (2009) Markov Decision Processes (MDP) Toolbox, URL `http://www.inra.fr/mia/T/MDPtoolbox/` 42

Collobert R, Bengio S, Mariéthoz J (2002) Torch: a modular machine learning software library. Tech. Rep. IDIAP-RR 02-46, IDIAP 29

Comité FD, Delepoulle S (2005) PIQLE: A Platform for Implementation of Q-Learning Experiments. In: NIPS workshop : Reinforcement Learning Benchmarks and Bake-offs II 39

Drummond C (2009) Replicability is not reproducibility: Nor is it good science, in Proceedings of the Twenty-Sixth International Conference on Machine Learning: Workshop on Evaluation Methods for Machine Learning IV 5

Edgington M (2009) Maja machine learning framework, URL `http://mmlf.sourceforge.net/` 40

Gamma R, Helm R, Johnson R, Vlissides J (1995) Design Patterns. Elements of Reusable Object-Oriented Software. Addison-Wesley 27, 28, 29

Gasser M (2008) Smarts 2.0, URL `http://www.cs.indiana.edu/~gasser/Smarts/` 33

Grand M (2002) Patterns in Java, vol 1, 2nd edn. Wiley 27

Hall M, Frank E, Holmes G, Pfahringer B, Reutemann P, Witten IH (2009) The WEKA Data Mining Software: An Update. SIGKDD Explorations 11(1):233–234 3

Johnston WM, Hanna JRP, Millar RJ (2004) Advances in dataflow programming languages. ACM Comput Surv 36(1):1–34, DOI http://doi.acm.org/10.1145/1013208.1013209 25

Juergens E, Deissenboeck F, Hummel B, Wagner S (2009) Do code clones matter? In: ICSE '09: Proceedings of the 31st International Conference on Software Engineering, IEEE Computer Society, pp 485–495 12

Kapusta D (2005) Connectionist Q-learning Java Framework, URL http://elsy.gdan.pl/ 43

Kerr AJ, Neller TW, Pilla CJL, Schompert MD (2003) Java resources for teaching reinforcement learning. In: In Proceedings of the International Conference on Parallel and Distributed Processing Techiniques and Applications, pp 1497–1501 39

Koschke R (2007) Survey of research on software clones, in: Duplication, Redundancy, and Similarity in Software. Dagstuhl seminar proceedings 12

Kovacs T (2004) Strength or Accuracy: Credit Assignment in Learning Classifier Systems. Springer, URL http://www.cs.bris.ac.uk/~kovacs/author.directory/thesis/thesis.html 31

Kovacs T, Kerber M (2006) A study of structural and parametric learning in XCS. Evolutionary Computation 14(1):1–19 32

McPhillips T, Bowers S, Zinn D, Ludäscher B (2009) Scientific workflow design for mere mortals. Future Gener Comput Syst 25(5):541–551, DOI http://dx.doi.org/10.1016/j.future.2008.06.013 25

Mierswa I, Wurst M, Klinkenberg R, Scholz M, Euler T (2006) YALE: Rapid prototyping for complex data mining tasks. In: Ungar L, Craven M, Gunopulos D, Eliassi-Rad T (eds) KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining, ACM, pp 935–940, DOI http://doi.acm.org/10.1145/1150402.1150531, URL http://rapid-i.com/component/option,com_docman/task,doc_download/gid,25/Itemid,62/ 3, 27

Mitchell T (1997) Machine Learning. McGraw Hill 6

Moler C, Little J, Bangert S (1987) Pro-Matlab User's Guide. The Mathworks, Cochituate Place, 24 Prime Park Way, Natick, MA, USA 3

Neumann G (2005) Reinforcement Learning for optimal control tasks. Master's thesis, Technischen Universitat, Graz 37

Riedmiller M, Lange S, Timmer S, Hafner R (2006) CLSquare, URL http://www.ni.uos.de/index.php?id=70 33

Roberts E (2004) Resources to support the use of Java in introductory computer science. ACM SIGCSE Bulletin 36(1):233–234 39

Roy CK, Cordy JR (2007) A survey on software clone detection research. Tech. Rep. 2007-541, Queen's University 12

Schaffer C (1994) A conservation law for generalization performance. In: Hirsh H, Cohen WW (eds) Machine Learning: Proceedings of the Eleventh International Conference, Morgan Kaufmann, San Francisco, CA, pp 259–265 7

Schaul T, Bayer J, Wierstra D, Sun Y, Felder M, Sehnke F, RückstießT, Schmidhuber J (2010) PyBrain. Journal of Machine Learning Research 11:743–746 41

Shalloway A, Trott JR (2005) Design Patterns Explained. A New Perspective on Object-Oriented Design, 2nd edn. Pearson Education 27

Sloman A (1994) Explorations in design space. In: Cohn AG (ed) Proceedings European Conference on Artificial Intelligence 1994, John Wiley, pp 578–582, URL http://www.cs.bham.ac.uk/~axs 7

Sonnenburg S, Braun ML, Ong CS, Bengio S, Bottou L, Holmes G, LeCun Y, Müller KR, Pereira F, Rasmussen CE, Rätsch G, Schölkopf B, Smola A, Vincent P, Weston

J, Williamson RC (2007) The need for open source software in machine learning. Journal of Machine Learning Research 8:2443–2466 5, 7, 23, 33

Stodden V (2010) The scientific method in practice: Reproducibility in the computational sciences. Tech. Rep. 4773-10, MIT Sloan School of Management 5, 8

Sutton RS, Barto AG (1998) Reinforcement Learning: an Introduction. MIT Press 1, 9, 14, 18, 23, 31

Tanner B (2009a) Project details for RL-Glue and Codecs on mloss.org. http://mloss.org/software/view/151/ 34

Tanner B (2009b) RL-Logbook, URL `http://logbook.rl-community.org/` 4

Tanner B (2010) RL-Library, URL `http://library.rl-community.org/` 4

Tanner B, White A (2009) RL-Glue : Language-independent software for reinforcement-learning experiments. Journal of Machine Learning Research 10:2133–2136 34

Vanschoren J (2010) Understanding machine learning performance with experiment databases. PhD thesis, Katholieke Universiteit Leuven 24

Whiteson S, Tanner B, White A (2010) The reinforcement learning competitions. AI Magazine 31(2):81–94 4

Witten IH, Frank E (2005) Data Mining: Practical Machine Learning Tools and Techniques, 2nd edn. Elsevier 3, 9

Wolpert DH (1996) The lack of a priori distinctions between learning algorithms. Neural Computation 8(7):1341–1390 7

Zito T, Wilbert N, Wiskott L, Berkes P (2008) Modular toolkit for Data Processing (MDP): a Python data processing framework. In: Frontiers in Neuroinformatics, vol 2