Tutorial for the Reinforcement Learning Toolbox

DI Michael Pfeiffer
Institute for Theoretical Computer Science, TU Graz
pfeiffer@TUGraz.at

November 2, 2006

1 Introduction

The Reinforcement Learning Toolbox (RLT) is a C++ library for reinforcement learning, developed by Gerhard Neumann at the Institute for Theoretical Computer Science at TU Graz. The main idea behind it is to provide beginners and experts with all tools to concentrate on policy learning problems, without having to care about implementation details of the numerous reinforcement learning (RL) algorithms that have been developed to this day. The RLT contains almost all of the most commonly used RL algorithms, as well as a lot of standard benchmark environments, and combines all this in a fast C++ environment. The separation between learning algorithm and environment allows you to re-use your code for different tasks. You could e.g. write the code for your environment once, and then compare Q-learning vs. SARSA-learning, or value-learning vs. Q-value-learning. Maybe you would like to compare different function approximation schemes for one task, or try a set of parameters to find the best learning rate? You can also compare policy search algorithms with value-function learning on the same task. All this is possible with the RLT.

1.1 Basic Principles

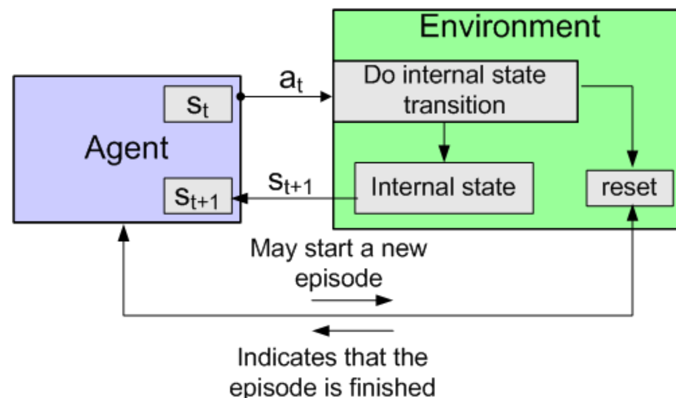

Figure 1: Structure of the learning system (from [1])

The basic structure of a learning task in the RLT is depicted in Figure 1. There are two basic blocks: the `Agent` and the `Environment`. The agent has some representation s_t of the current state of the environment. Based upon this information, the agent performs action a_t, which executes the transition function of the environment. The transition function changes the internal state of the environment, which is the exact information about the current state. The agent is informed

of the new state by receiving s_{t+1}. This representation should contain all internal state variables, but the agent can pre-process this data to create a state signal that is suitable for its learning algorithm. Since most learning algorithms however rely on the Markov property, you should make sure that the state information for the agent is rich enough, to enable the learning of a sensible policy.

Of course we also need a reward function in order to do reinforcement learning. Again, the RL toolbox is very flexible concerning this issue. Instead of having only one global reward function as part of the environment, the RLT introduces the concept of so called *listeners*, which are maintained by the agent. The agent can have several listeners, and every listener receives the triple $\langle s_t, a_t, s_{t+1} \rangle$ as input. Listeners can then either implement a learning algorithm, or they can be used for logging or parameter adaptation. The most important class of listeners are the so called *reward listeners*, which are given a handle to a reward function upon construction. The reward is then evaluated at every step, and the listener may use this additional information to update its internal data (e.g. a value function). The listeners may of course influence the action selection by the agent, as can be seen in Figure 2. Nearly all learning algorithms implement the interface of reward listeners.

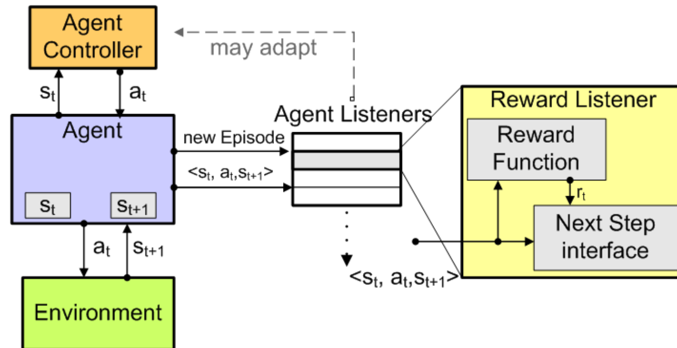

Figure 2: Agent listeners as interface for learning algorithms(from [1])

1.2 About this Tutorial

This tutorial gives you a brief introduction to the most common tasks to be solved with the RLT:

- Creating a learning environment.
- Applying learning algorithms for a given task.
- Implementing new learning algorithms.

2 Creating Environments

Environments are C++ classes that implement state transitions and communicate with the agent. The base class from which environments are derived is `CEnvironmentModel`. It provides the function `doNextState`, which immediately executes a given action and therefore changes the internal state of the model. This is a suitable interface for non-simulated tasks, e.g. robotics tasks, where the result of an action cannot be accurately predicted. On the other hand, we sometimes want to question the environment about the outcome of a particular action, without actually executing it. The interface `CTransitionFunction` implements the function `transitionFunction`, which takes a state and an action as input, and returns the corresponding next state without changing any internal variables. Transition functions can be turned into environments by creating an object

of the type `CTransitionFunctionEnvironment` with the transition function as argument. Additionally there is a number of functions to check the current state or reset the model, which are described below.

2.1 Defining the State Space

A state in the RLT is a vector of continuous and/or discrete state variables. The number of state variables must be known upon creation of the environment, and it needs to be passed to the base class constructor. Let's assume we wanted to create an environment called `CMyEnvironment` with 2 continuous and 1 discrete state variables. We also want to specify a range for all the state variables. Then the constructor would look similar to this:

```
1  CMyEnvironment::CMyEnvironment(...) : CEnvironmentModel(2,1) {
2      properties->setMinValue(0, -1);
3      properties->setMaxValue(0, +1);
4
5      properties->setMinValue(1, 0);
6      properties->setMaxValue(1, 2*PI);
7      properties->setPeriodicity(1, true);
8
9      properties->setDiscreteStateSize(0, 10);
10 }
```

In line 1, we define the number of continuous and discrete state variables by passing it to the constructor of `CEnvironmentModel`. This automatically generates the object `properties` of type `CStateProperties`, which stores all the information about state variables. In lines 2-3 we limit the first continuous variable to the the interval $[-1, +1]$. In 5-7 we limit the second continuous variable to the interval $[0, 2\pi]$, and we also make it a periodic variable. In line 9 we define that the discrete state variable can take on 10 different values.

An environment is just an ordinary C++ class, derived from `CEnvironmentModel`, so it may contain an arbitrary number of internal C++ variables that define the state. The environment however needs to pass a representation of the internal state to the agent through the method `getState(CState *state)`. `getState` writes the current state of the environment into the state object. Usually this function looks like this:

```
1  CMyEnvironment::getState(CState *state) {
2      CEnvironmentModel::getState(state);
3
4      state->setContinuousState(0, x);
5      state->setContinuousState(1, y);
6      state->setDiscreteState(0, n);
7  }
```

The call to the base method in line 2 makes sure that the properties of the state variables are correctly written into the object `state`. Lines 4-5 set the values of the two continuous state variables to x and y respectively (assuming that they have been defined somewhere). In line 6 we set the value of the discrete variable.

The `getState` method should always output all relevant state variables. Processing of this information (e.g. discretization, calculation of features, ...) is done at a later stage through so called *state modifiers*, implemented in the class `CStateModifier`.

2.2 State Modifiers

The information contained in a `CState` object is a fairly general representation of the internal state and usually cannot be used directly for learning. We first need to transform the internal state into a useful format. Typical transformation are e.g.:

- Combining multiple discrete state variables to calculate one discrete value (e.g. an index for a look-up table).
- Discretizing continuous state variables.
- Calculating activations of different RBF units for a given state.

`CStateModifier` is the interface for doing this sort of transformation. Actually, state modifiers are maintained by the agent, not by the environment, and every agent keeps a list of different modifiers. Classes derived from `CStateModifier` have to implement the function `getModifiedState(CStateCollection *originalStates, CState *modifiedState)`, which takes the original state as input and outputs the modified state. The modified state of course can have different properties than the original one, like a different number of discrete or continuous state variables. Notice that the input argument to `getModifiedState` is of the type `CStateCollection`. A state collection consists of the original state, and a number of modified states, as calculated by different state modifiers. Usually you only need the original state, which can be obtained by `CState* getState()`. Further information about state collections can be found in [1]. We will now concentrate on how to use the built in modifiers.

2.2.1 Discretization

Discretizers are derived from the class `CAbstractStateDiscretizer`. The most important function is `int getDiscreteStateNumber(CStateCollection *state)`, which returns the integer number of the discretized state. The derived class `CSingleStateDiscretizer` works on one continuous state variable, and partitions the real values into a finite number of bins. The constructor `CSingleStateDiscretizer(int dimension, int numPartitions, rlt_real *partitions)` receives as input the index of the continuous variable to discretize, the number of discretization borders, and the borders of the bins themselves. `getDiscreteStateNumber` then returns the index `i` in the partitioning, such that the value of the continuous variable falls into the i-th bin. Let `p[]` be the array of borders, then the corresponding bins are $(-\infty, b[0]], (b[0], b[1]], \ldots, (b[n-1], b[n]], (b[n], \infty)$.

Combining multiple state variables into one discrete index is possible with the use of the *and*-operator `CDiscreteStateOperatorAnd`. The *and*-operator creates a state space whose size is the product of the sizes of the individual modifiers. Different discretizers can be added to the operator with the command `addStateModifier(CAbstractStateDiscretizer *featCalc)`, and the operator transforms the individual discrete states to one unique index.

Sometimes it is convenient to have different levels of abstraction at different regions in the state space. States where the policy has failed e.g. do not need a fine resolution, whereas sometimes it makes sense to have a higher resolution close to the target. In this case you can use the command `addStateSubstitution(int discState, CStateModifier *modifier)` for all discretizers. The effect is that whenever the output of the discretizer equals `discState`, the state is further process by the given `modifier`, and the final output is the result of the new modifier applied to the state.

2.2.2 Linear Feature Calculation

Linear function approximation requires the state information to come in the form of a vector of feature values. The interface for computing features is `CFeatureCalculator`. The most common linear approximation schemes are *tile coding* and *Radial-Basis-Function (RBF) networks*. Feature states consist of a list of active features and their corresponding activations. Inactive features are

assumed to have the activation 0. In a single tiling e.g. only one feature is active with activation 1, while there may be several active RBF units with different levels of activations. The following section describes how you can use the built in classes to calculate tile or RBF activations for one or more continuous state variables.

Tile Coding

Tilings partition the state space exhaustively into cells, and the index of the cell is the active feature. Since there is only one cell into which a state can fall, there is always only one active feature per tiling. Usually however we combine multiple tilings with different offsets, sizes or shapes of the cells to get a finer resolution. In the toolbox a single tiling is represented by the class `CTilingFeatureCalculator`. When you create a tiling you have to specify on which dimensions of the continuous state space it should operate, how many partitionings per dimensions you need, and what is the offset of the first cell. The widths of the cells are then automatically determined for every dimension. For feature calculation the RL toolbox always scales continuous variables into the interval $[0, 1]$, using the minimum and maximum values that were defined in the state properties. Take this into account when you specify the offsets.

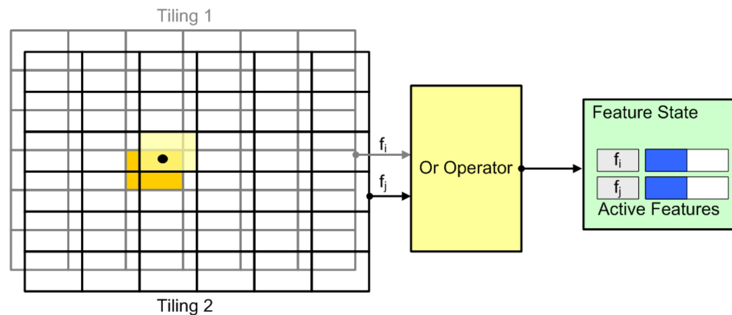

Figure 3: Combining multiple tilings with the Or-Operator (from [1])

To increase the accuracy of our approximation we usually need more than one tiling. The operator `CFeatureOperatorOr` can be used exactly for this purpose. First you can add different state modifiers with the function `addStateModifier`. The *or*-operation then combines independent feature calculators, such that all active features from the modifiers are simultaneously active. This means that the size of the resulting feature state space is the sum of the individual feature state spaces of the modifiers. In figure 3 the operation is shown for the combination of two tilings with different offsets.

Radial Basis Functions

As with tile codings, continuous variables are again scaled into the interval $[0, 1]$. When you define RBF centers you only need to specify the number of kernels you want to use, the offset of the first center, and the width σ of the Gaussian kernels. The toolbox will automatically place the centers uniformly spaced over the interval $[0, 1]$. Since the RBF kernels should overlap, you should make sure to choose the number of kernels and the width of the kernels large enough. As a rule of thumb $2 \cdot \sigma \cdot$ #kernels should be approximately 1. When calculating the activation of features the toolbox only returns the activations of those kernels, that are closer than $2 \cdot \sigma$ to the current state. The RBF networks also normalizes the activation factors such that all factors always sum to one.

There are two possibilities to define RBF features: you can either lay a uniform grid over the whole state space and place the centers of the RBF kernels there, or you can define RBF centers individually for each state variable. The class `CRBFFeatureCalculator` lays a uniform grid over the state space. In the constructor you specify on which dimensions of the state space you want to work, how many centers per dimensions you wish to use, the offset of the first RBF unit and the widths σ_i. You can find a short example with uniform RBF grids at `http://www.igi.tugraz.at/ril-toolbox/examples/multipole`.

The class `CSingleStateRBFFeatureCalculator` should be used if you want to use non-uniformly spaced RBF centers. Here you can specify the exact positions of the RBF centers (in the non-normalized state-space), and individual widths σ for every kernel. Use the `CFeatureOperatorAnd` class to combine multiple single-state RBF codings. After adding the modifiers for all dimensions and before passing the state modifier to the agent you have to call `andOperator->initFeatureOperator()` to initialize the and-operator.

2.2.3 Neural Networks

The state-modifier `CNeuralNetworkStateModifier` is used for function approximation with neural networks. It receives a handle to the properties of your environment state in the constructor. Every state variable defines one or two inputs for the neural network. Discrete variables remain unchanged. Continuous non-periodic variables correspond to one network input, which is scaled into the interval $[-1, 1]$. Periodic continuous state variables are transformed into two inputs, one defining the sinus, the other one the cosinus of the normalized state variable. Therefore the number of state variables for the neural network state is the number of original state variables plus the number of continuous periodic variables.

2.3 Actions

Actions in the RLT are also objects, derived from the base class `CAction`. In most examples, the action space is discrete, and so the index of the action within an action set is the only information. Use the base class `CPrimitiveAction` to derive your own action class for this case. There may however be also action objects that contain more information. Actions in a Semi-MDP e.g. have a certain duration, and so they need to store how long they are already executing. Also continuous actions contain some data that describes the effect of its execution.

2.4 Transition Functions

The most important function in an environment is probably the transition function that describes how the environment reacts if a certain action is applied in a certain state. Depending on whether your base class is `CEnvironmentModel` or `CTransitionFunction`, you will have to overwrite the functions `doNextState(CPrimitiveAction *action)` or `transitionFunction(CState *oldState, CAction *action, CState *newState, CActionData *actionData)` respectively. The difference is that in the first case you have to change the internal variables of your environment, while in the second case you just write the resulting state into `newState`.

Let's concentrate on `CTransitionFunction`, the transition for `CEnvironmentModel` is very similar. Usually the first thing to do within the transition function is to cast the action or state objects to the right class types (always use `dynamic_cast`!). Then we would fetch the relevant information from the `oldState` object and the `action`. Afterwards it is up to you to calculate the new state, depending on the mathematical model of your environment. Afterwards be sure to save the new state variables in `newState`. Here is an example how a transition function could look like:

```
CMyEnvironment::transitionFunction(CState *oldState, CAction *action,
                                   CState *newState,
                                   CActionData *actionData) {

  CMyAction *myAction    = dynamic_cast<CMyAction *>(action);

  ...
  // calculate state changes
  ...

  newState->setContinuousState(0, new_x);
  newState->setContinuousState(1, new_y);
  newState->setDiscreteState(0, new_n);
}
```

This function first makes the necessary casts and then computes the changes in the state variables, before it finally writes the new values to the new state object.

2.5 End of Episodes

After a transition the environment may be in a state where there is no way to continue. In this case the episode has to be reset, either because the episode has failed, or because the agent has reached its target position. In `CEnvironmentModel` this is indicated by the two binary flags `failed` and `reset`, which need to be set in the transition function `doNextState`. Environments derived from `CTransitionFunction` need to implement the two functions `isFailedState(CState *state)` and `isResetState(CState *state)`, which check whether a given state is a terminal and/or failed state.

Assuming that we want to play more than one episode, we need to specify what happens when the environment needs to be reset. In `CEnvironmentModel` this has to be done by overwriting the function `doResetModel()`, where all the internal variables are set to initial values. For environments derived from `CTransitionFunction` we need to provide a function `getResetState(CState *resetState)`, which writes new initial values for all state variables into the state object `resetState`. Here you can e.g. choose whether you want to start with random initial values or whether you prefer deterministic starting conditions.

2.6 Rewards

Reward functions are derived from the base class `CRewardFunction`, so they need to implement the function `rlt_real getReward(CStateCollection *oldState, CAction *action, CStateCollection *newState)`. This function calculates a real-valued reward, given the last state, an action and the resulting state. You will sometimes find that the reward function is not an own object, but also part of the environment. In that case of course you need to use multiple inheritance for your environment class. It is however possible to completely separate environment and reward function. In that case, the reward function is only given to a reward listener (e.g. a learning algorithm), which implements the interface `CSemiMDPRewardListener`.

2.7 Gridworlds

Gridworlds are very commonly used in the reinforcement learning literature to demonstrate new learning algorithms. These simple environments consist of a 2-dimensional grid of square cells, and usually the agent can only go left, right, up or down from one cell to the next. There are also walls through which the agent cannot pass, and special fields like goals or traps. The state- and action space are therefore always discrete and fairly small, which makes it easier to learn an

optimal policy. Since these environments are so common, the RLT provides a special tool to easily create gridworld environments.

The class `CGridWorldModel` creates gridworld environments from simple text files. Gridworld models are created by calling the constructor `CGridWorldModel(char *filename, int max_bounces)`. The first argument specifies the name of a text-file, in which the configuration data for the gridworld is stored. The second argument sets the number of bounces into walls after which an episode is reset. We first show a little example of a gridworld configuration file, before we describe the syntax of gridworld models in more detail.

```
1  # Start of gridworld configuration-file
2  Gridworld
3  Size: 7x5
4  StartValues: 1, 2
5  TargetValues: 4, 5
6  ProhibitedValues: 8, 9
7
8  9999999
9  9000059
10 90999T9
11 9100009
12 9999999
```

The start of a gridworld file is marked by the keyword **Gridworld**. In line 3 we define the size of the gridworld with `Size: CxR`, where C is the number of columns and R is the number of rows. With the line `StartValues: 1,2` we define that the symbols `1` and `2` indicate possible starting positions, and `TargetValues: 4, 5` defines `4` and `5` as symbols for goal states. `ProhibitedValues` defines the symbols for blocked cells, e.g. walls. An agent stepping into one of these fields increases its number of bounces, and when `max_bounce` is reached, the episode is reset.
The actual map of the gridworld is defined in lines 8-12. Every symbol defines the type of the cell, that is whether it is an empty field, a wall, a start or a goal state, or some special state. The `9` symbols define the walls of the environment, whereas the `0`'s indicate empty fields. `1` is the starting state and `5` is the goal state. Notice the symbol `T` in line 10. This symbol was not defined as a start, target or prohibited value. It is treated like an empty field, but we can later define that it has a different reward. If we wanted it to be also a terminal state, then we would need to include it in the list `TargetValues`. The gridworld that is produced by the above configuration file can be seen in Figure 4.

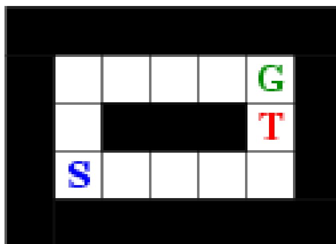

Figure 4: Example for a gridworld: (S=starting state, G=goal state, T=trap)

What is left is the definition of the actions and rewards for the gridworld. Actions objects are defined with `CGridWorldAction(int x_move, int y_move)`, which creates an action that moves

the agent `x_move` cells to the right and `y_move` cells down. The action `new CGridWorldAction(1,0)` would therefore correspond to a movement to the east.

`CGridWorldModel` also implements the interface `CRewardFunction`, so for many tasks you do not need to implement an additional class for the reward function. There are four types of rewards that are pre-defined for gridworlds: The standard reward is set with the function `setRewardStandard`. This reward is received at every step that ends on a `normal` cell, like e.g. those marked with `0` in the above gridworld. The bounce reward, set with `setRewardBounce` occurs whenever the agent hits a wall. When the agent reaches a target state, it receives the reward set by `setRewardSuccess`. Additionally you can define special rewards for other cells, depending on the symbol in the map configuration. In the above example we defined a cell with the symbol `T`, which should be a trap. Using the command `setRewardForSymbol('T', -10)` we define that the agent always receives reward -10 when it enters a cell marked with `T`.

There are also ready-made state modifiers for gridworlds. The class `CGlobalGridWorldDiscreteState` is a discretizer that transforms the 2-dimensional state in a gridworld of known size into a single discrete index. The classes `CLocal4GridWorldState`, `CLocal4XGridWorldState`, and `CLocal8GridWorldState` provide only local information about the 4 or 8 neighboring fields of the agent. See the class reference for more details.

If you need to implement special features into your gridworld, like e.g. teleports or restrictions of actions, you need to derive a new class from `CGridWorldModel` and implement the transition function for special cases yourself. More information about gridworlds can be found in the class reference and in the online tutorial at `http://www.igi.tugraz.at/ril-toolbox/examples/shortestpath`.

3 Creating Agents

The agent is the object that interacts with the environment and chooses its actions as to learn an optimal policy. The agent stores information about its internal state and has a set of actions from which it can choose. By executing one of these actions the current state of the environment changes, and the resulting state is passed to a number of listeners, which implement learning algorithms or loggers. The policy that the agent follows comes from a controller object, which is separated from the learning algorithms, but of course interaction is possible and usual.

The base class for agents is `CAgent`. Whenever you create an agent with `CAgent(CEnvironmentModel *model)` you have to supply it with a pointer to the environment model. Once the agent is created you have to define a number of components and parameters to enable the learning process. These components are described in the following chapter.

3.1 Defining the Action Set

The agent of course needs to know which actions it can execute, before it learns which of them is best in a given situation. The action set consists only of primitive actions, which of course must also be known to the environment. You first create the different action objects and then use the agent's function `addAction(CPrimitiveAction *action)` to add it to the set of available actions.

3.2 State Modifiers

State modifiers, as explained in section 2.2 transform the original state returned by the environment into a format that can be used for learning. Discretization, feature calculation etc. fall into this category. In order to tell the agent how to interpret a state object received from the environment,

we need to add one or more modifiers to the agent's internal list. We do this by calling the agent's function `addStateModifier(CStateModifier *modifier)` with the modifier as its argument.

3.3 Value Function and Q-Functions

Value- and Q-functions are used for controllers and learning algorithms. Since these functions have to store one entry for every state or state-action-pair respectively, they depend on the action and the state modifiers that are used.

3.3.1 Value Functions

You will most likely work either with `CVTable`, which stores one value entry for every discrete state in a table, or `CFeatureVFunction`, which can also handle weights for linear approximations like tile-codings or RBF networks. Both are created with the corresponding state modifier as the only input argument to the constructor.

3.3.2 Q-Functions

The most frequently used class for Q-functions is `CFeatureQFunction`, which handles Q-tables and linear approximations. Objects of this type are constructed with `CFeatureQFunction(CActionSet *actions, CStateModifier *discretizer)`. The action set can be obtained from the agent via `agent->getActions()`, and the state modifier should be the same that is used for the agent.

3.4 Using Function Approximation

If you are using linear function approximation like tile coding or RBF networks, you can proceed as usual by passing the state-modifiers to a `CFeatureVFunction` or `CFeatureQFunction`.

The RL toolbox interacts with the Torch [2] machine learning library to use neural networks for function approximation. Therefore you first have to use the Torch classes to create your neural network and then create a RL toolbox object of type `CTorchGradientFunction` from it. Typically you create neural networks of type `MLP`, which represent multi-layer perceptrons. The constructor for MLP looks like this:

```
MLP::MLP(int n_layers, int n_inputs, char *transfer, int n_outputs, ...)
```

The first argument specifies how many layers the network has (the input layer counts as the first layer). `n_inputs` defines the number of inputs to the network. What follows is a variable list of arguments, two per layer. The string indicates the transfer function of the layer, which can be either `linear, tanh, sigmoid, softmax, log-softmax, exp` or `softplus`. The second argument for a layer is the number of outputs of this layer, which is also the number of inputs to the next layer. Typically you have one hidden layer with `sigmoid` or `tanh` transfer function, and a linear summation of the hidden units in the output layer. You would create this with

```
MLP *mlp = new MLP(3, n_inputs, "linear", n_hidden, "sigmoid", n_hidden, "linear", 1)
```

In this example `n_inputs` is the number of inputs and `n_hidden` is the number of hidden units. To transform an `MLP` object to a toolbox object use the constructor `CTorchGradientFunction(mlp)` with the `MLP` as argument. You can then create a value function of type `CVFunctionFromGradientFunction`, with the network object and the network state-modifier as input. To create separate V-functions for every action, first create the different neural networks, then a `CQFunction` object and finally use the function `setVFunction(CAction *action, CAbstractVFunction *vfunction)` to use the neural network as function approximator for the specified action.

3.5 Controllers and Action Selection

One of the most important components of the agent is the agent-controller, which selects the actions during the learning process. Usually in reinforcement learning the actions during learning are selected stochastically according to the learned Q- or value function. The RL toolbox however allows you to distinguish between the learned policy and the policy that the agent follows during learning. You could e.g. follow a random or hand-coded policy during learning, and via off-policy learning still learn an optimal policy. Controllers are derived from `CAgentController`, and can be activated for an agent by calling `agent->setController(CAgentController *controller)`. Learning algorithms on the other hand are implement as listeners, and are covered in the next section.

If you want to write your own agent controller you need to implement the function `CAction* getNextAction(CStateCollection *state)`. More frequently however you will want to choose your next action according to a learned Q-function (or value function). The class `CQStochasticPolicy` implements exactly this functionality. Upon construction with `CQStochasticPolicy(CActionSet *actions, CActionDistribution *distribution, CAbstractQFunction *qfunction)` it receives the set of possible actions, a general probability distribution object (e.g. ϵ-greedy or softmax-distribution), and a pointer to a Q-function. In its `getNextAction` method the controller then automatically selects actions according to the specified probability distribution and the values of the actions in the current state.

3.5.1 Probability Distributions for Exploration / Exploitation

The RLT already implements the most commonly used probability distributions for action selection: The ϵ-greedy (`CEpsilonGreedyDistribution`) and the softmax distribution (`CSoftMaxDistribution`). The ϵ-greedy distribution is constructed with a single parameter ϵ, which indicates the probability of choosing non-optimal actions. If you want to change the parameter ϵ during learning, you need to use the function `setParameter("EpsilonGreedy", new_value)`.

The softmax-distributions receives one parameter, the so-called inverse temperature β, which indicates the *greediness* of action selection. A β close to 0 produces almost random exploration, while a large β causes almost greedy selection. To make the action selection insensitive to the scaling of rewards, the minimum Q-value is first subtracted from all values before the softmax-selection actually takes place. If you need to change the β parameter during learning, use the method `setParameter("SoftMaxBeta", new_value)`. Softmax is a differentiable distribution, so it can also be used for policy gradient learning.

3.5.2 Action Selection with Value Functions

If you learn only value functions, you need a model to predict the outcome of an action in the current state, to evaluate your choices. The class `CVMStochasticPolicy` can be used as a controller for value learning, but you need to provide a transition function and a reward function additional to the value function. Usually you would just the same functions that you use for simulation of the environment, but you could also e.g. use less perfect approximations of the real transition function to make the learned policy more robust.

3.6 Listeners

The agent keeps a list of listeners, that receive the input tuple $\langle s_t, a_t, s_{t+1}\rangle$ at every step. The listeners may now either implement a learning algorithm, they can be used for logging, or they can modify learning parameters. Listeners are derived from the base class `CSemiMDPListener`, and can be activated by calling the agent's function `agent->addSemiMDPListener(CSemiMDPListener *listener)`. Listeners have to implement the functions `nextStep`, in which they react to a

$\langle s_t, a_t, s_{t+1} \rangle$ tuple, and `newEpisode()`, which tells the listener that a new episode has begun. We will now discuss the three most important tasks for which listeners are used.

3.6.1 Reward Listeners

One of the main design principle of the RLT is that the reward function is decoupled from the environment. So we can use multiple learners with different reward functions at the same time. The reward at a given time step depends on the current state s_t, the chosen action a_t, and the resulting state s_{t+1}, so $r_t = r(s_t, a_t, s_{t+1})$. This information is received at every step by a listener object, and can be passed to a `CRewardFunction` object (see 2.6) to calculate the reward. Listeners that need information about reward are derived from `CSemiMDPRewardListener`. The constructor receives a pointer to a reward function as argument, and the `nextStep` method now also has a reward value as additional input. Almost all learning algorithms implement the `CSemiMDPRewardListener` interface, but we will describe them in more detail in section 4.

3.6.2 Loggers

Training trials can be logged to trace the learned policy or to reuse the stored trials for learning. The class `CAgentLogger` stores all the episodes of the agent in memory. It is initialized with a pointer to the model and the action set, and optionally with a maximum number of episodes that should be stored. After the learning trials the episodes can be written to a file with the command `saveBIN`, which produces a binary file, or `saveData`, which produces a text file. `CEpisodeOutput` can be used to write $\langle s_t, a_t, r_t, s_{t+1} \rangle$ tuples directly to a text file in readable format. Since it is also a reward listener, it needs to receive a pointer to a reward function.

3.7 Adaptive Parameters

All learning algorithms have some parameters like a learning rate, discount factor or exploration rate. Sometimes it makes sense to use adaptive parameters that change their values during the learning process, e.g. because the learning algorithm has converged to a better solution. In the RL toolbox parameters are identified by a string name, and are mapped to a double value. The toolbox provides an automatic way to update learning parameters during the learning process. For every object with parameters you can call the function `addAdaptiveParameter(string name, CAdaptiveParameterCalculator *paramCalc)`, which makes the parameter identified by `name` adaptive. The adaptive parameter calculator can be chosen from a number of pre-defined classes. The difference is the quantity on which the adaptation depends, e.g. the number of episodes (`CAdaptiveParameterFromNEpisodesCalculator`) or the average reward over the last steps (`CAdaptiveParameterFromAverageRewardCalculator`). You can also define a number of meta-parameters that define the type of function by which the parameter is transformed (e.g. linear, logarithmic or polynomial functions). Look at the class reference for more details.

4 Learning

After implementing the environment and the agent, the most difficult parts are done. Now we just need to put all the components together to create a learning system. The following steps need to be executed:

1. Create the learning algorithm.
2. Set the parameters of the algorithm.
3. Combine learners and controllers.
4. Run several training episodes and evaluate the results.

4.1 Setting up the Learning Algorithm

We will focus here on temporal difference (TD) learning. Aactor-critic methods are described in the *Advanced Topics* section. We will have to differentiate between algorithms that learn state value (V) functions, and state-action value (Q) functions. Both need to know the reward function and the data structure to store the V- or Q-function.

4.1.1 Learning V-functions

To create a value learner, use the constructor `CVFunctionLearner(CRewardFunction *rewardFunction, CAbstractVFunction *vFunction)`. It receives a pointer to the reward function, and a pointer to the function that is used to store state values (see 3.3.1).

4.1.2 Learning Q-functions

There are two basic algorithms in the toolbox for learning Q-functions: *Q-learning*, defined in `CQLearner`, and *SARSA* learning, defined in `CSarsaLearner`. Both receive as input to their constructors a reward function and a pointer to a Q-function object. Since SARSA works on-policy, it also needs a pointer to the policy that the agent follows. This is done by passing a pointer to the agent as the third argument to the constructor of `CSarsaLearner`.

4.1.3 Eligibility Traces

You can use eligibility traces both for value- and for Q-learning. They are automatically created when the V- or Q-learner is created. You can get a pointer to the e-traces by calling either `getVETraces` for a V-learner, or `getETraces` for a Q-learner. You can then set parameters of the eligibility traces, like e.g. setting the decay factor λ with a call to `setLambda(rlt_real lambda)`. To deactivate eligibility traces set $\lambda = 0$. You can also use the function `setReplacingETraces(bool bReplace)` to activate replacing traces. By default accumulating traces are used.

When you are using function approximation, eligibility traces are kept not for every state, but for every feature. There may be a lot of features, so for performance reasons it is better to delete traces if they are so low that they have hardly any influence on an update. There are two parameters that control how many eligibility traces are stored in memory, and when they are cut off. `ETraceMaxListSize` is the maximum number of features that can be stored in the eligibility trace list. The default value is 100, so if you have a large number of features you should increase this value. The second parameter is `ETraceTreshold`, which is the minimum value of a feature before it is deleted from the eligibility trace list. By default this value is set to 0.001.

4.1.4 Setting Parameters of Learning Algorithms

Learning algorithms have a number of parameters that you can set that influence the learning process. They can all be set by calling the function `setParameter(string name, double value)` with the string identifier of the parameter as its first argument. Table 4.1.4 gives an overview of the most frequently needed parameters for learning algorithms and their components and the class where they have to be set. For more parameters of other algorithms see the class reference.

4.2 Using Learning Controllers

In section 3.5 we learned that the controller of an agent may be separated from the learning algorithm. We also described how you can create a `CQStochasticPolicy`-controller that stochastically selects its actions according to a given value- or Q-function. If we now pass the same value function object to the controller and the TD-learner, the agent will automatically follow a policy that always depends on the currently learned value- or Q-function. It works like this:

Parameter Name	Defined in Class	Description
DiscountFactor	CVFunctionLearner, CTDLearner	Discount factor γ for the value- or Q-function
VLearningRate	CVFunctionLearner	Learning rate α for value function learners
QLearningRate	CTDLearner	Learning rate α for Q-function learners
Lambda	CAbstractVETraces, CAbstractQETraces	Decay factor for eligibility traces
ETraceMaxListSize	CAbstractVEtraces, CGradientQETraces	Maximum length of e-trace list
ETraceTreshold	CGradientVEtraces, CGradientQETraces	Minimum value at which e-traces are cut off
EpsilonGreedy	CAgentController	Exploration rate ϵ for ϵ-greedy action selection
SoftMaxBeta	CAgentController	Inverse temperature β for SoftMax exploration

Table 1: Parameters for learning algorithms and components

the agent controller uses the current value function to select the next action. The resulting tuple $\langle s_t, a_t, r_t, s_{t+1} \rangle$ (and a_{t+1} if we use SARSA) is passed to the TD learning algorithm. The learner performs an update of the value function before the next action selection by the controller occurs.

In the program this means that you have to follow the following steps before you start the learning process:

1. Create the environment model, the agent and the reward function.
2. Create the value- or Q-function object.
3. Create the learner object and pass the value function object as argument to its constructor.
4. Add the learner to the agent's list of listeners.
5. Create the controller as a stochastic-policy object with the pointer to the value function as argument to its constructor.
6. Use `setController` to set the stochastic policy as the agent's controller.
7. Set parameters, add loggers etc. before you start the learning process.

4.3 Starting the Learning Process

There are several ways to start the learning process, all of them are functions of the agent. The most common way is to run a certain number of episodes. This can be done by calling the agent's function `agent->doControllerEpisode(nEpisodes, nStepsPerEpisode)`. This methods simulates `nEpisodes` episodes, and stops every episode after at most `nStepsPerEpisode` steps. The function returns the number of steps in the last episode.

You can also simulate a task step-by-step. The method `agent->doControllerStep()` makes the agent execute exactly one action. To check whether the episode has failed or reached the target position, use the functions `isReset()` and `isFailed()` of the environment. To start a new episode use `agent->startNewEpisode()`.

4.4 Evaluating a Policy

There are several criteria to judge the quality of a learned policy. Usual measures are the average reward per episode or the expected value from a given start state. The class `CPolicyEvaluator` and

its subclasses provide listeners that can calculate this quality indices during the learning process. `CValueCalculator` calculates average discounted sum of rewards from starting states of the different episodes. `CAverageRewardCalculator` returns the average reward per episode. All evaluators receive a pointer to the agent in their constructor, and by calling the function `evaluatePolicy()` they calculate the average quality measure over a given number of episodes. In this case the learning process is automatically started, you do not have to call `doControllerEpisode` yourself.

Acknowledgements

Thanks to Gerhard and Stephan Neumann for the implementation of the toolbox.

A Advanced Topics

In this section we describe some of the more difficult tasks for which the RL toolbox can be used.

A.1 Actor Critic Learning

Actor-critic methods separate the policy from the value function. The *actor* only learns the policy, which is a function $\pi(s)$ that depends only on the current state. The *critic* on the other learns to evaluate a policy by learning the value function and giving the actor feedback about how good his last actions were. In the toolbox this functionality is provided by two different interfaces. The critic is just a normal TD learner, which implements the interface `CErrorSender`. An error-sender sends the tuple $\langle s_t, a_t, \Delta TD \rangle$, where ΔTD is the TD-error, to its list of registered error-listeners. Actors, derived from `CActor` on the other hand implement the interface `CErrorListener`, and can be added to the TD learner's list with the command `learner->addErrorListener(CErrorListener *actor)`. In the function `receiveError` the actor needs to adapt its policy according to the information received by the critic. The architecture of actor-critic learning in the RLT is depicted in Figure 5. Have a look at `http://www.igi.tugraz.at/ril-toolbox/examples/multipole` for a simple example with an actor that can only choose between two actions.

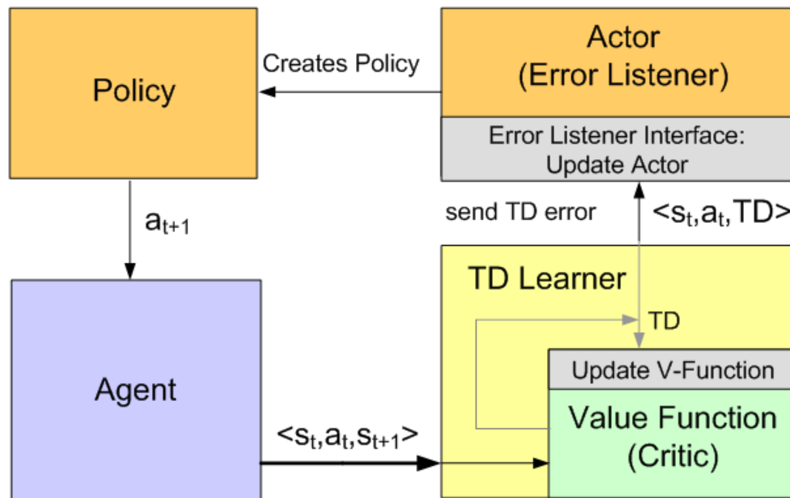

Figure 5: Actor-Critic architecture (from [1])

Contents

1 Introduction **1**
1.1 Basic Principles . . . 1
1.2 About this Tutorial . . . 2

2 Creating Environments **2**
2.1 Defining the State Space . . . 3
2.2 State Modifiers . . . 4
2.2.1 Discretization . . . 4
2.2.2 Linear Feature Calculation . . . 4
2.2.3 Neural Networks . . . 6
2.3 Actions . . . 6
2.4 Transition Functions . . . 6
2.5 End of Episodes . . . 7
2.6 Rewards . . . 7
2.7 Gridworlds . . . 7

3 Creating Agents **9**
3.1 Defining the Action Set . . . 9
3.2 State Modifiers . . . 9
3.3 Value Function and Q-Functions . . . 10
3.3.1 Value Functions . . . 10
3.3.2 Q-Functions . . . 10
3.4 Using Function Approximation . . . 10
3.5 Controllers and Action Selection . . . 11
3.5.1 Probability Distributions for Exploration / Exploitation . . . 11
3.5.2 Action Selection with Value Functions . . . 11
3.6 Listeners . . . 11
3.6.1 Reward Listeners . . . 12
3.6.2 Loggers . . . 12
3.7 Adaptive Parameters . . . 12

4 Learning **12**
4.1 Setting up the Learning Algorithm . . . 13
4.1.1 Learning V-functions . . . 13
4.1.2 Learning Q-functions . . . 13
4.1.3 Eligibility Traces . . . 13
4.1.4 Setting Parameters of Learning Algorithms . . . 13
4.2 Using Learning Controllers . . . 13
4.3 Starting the Learning Process . . . 14
4.4 Evaluating a Policy . . . 14

A Advanced Topics **15**
A.1 Actor Critic Learning . . . 15

Index

Accumulating Traces, 13
Action Selection, 11
Actions, 6
Actor-Critic Learning, 15
Adaptive Parameters, 12
addAction, 9
addAdaptiveParameter, 12
addSemiMDPListener, 11
addStateModifier, 4, 10
addStateSubstitution, 4
Agent, 9–12
And-Operator, 4

Classes
- CAbstractStateDiscretizer, 4
- CAction, 6
- CAgent, 9
- CAgentLogger, 12
- CAverageRewardCalculator, 15
- CDiscreteStateOperatorAnd, 4
- CEnvironmentModel, 2, 6
- CEpisodeOutput, 12
- CEpsilonGreedyDistribution, 11
- CErrorListener, 15
- CErrorSender, 15
- CFeatureCalculator, 4
- CFeatureOperatorAnd, 6
- CFeatureOperatorOr, 5
- CFeatureQFunction, 10
- CFeatureVFunction, 10
- CGlobalGridWorldDiscreteState, 9
- CGridWorldAction, 8
- CGridWorldModel, 8
- CNeuralNetworkStateModifier, 6
- CPolicyEvaluator, 14
- CPrimitiveAction, 6
- CQLearner, 13
- CQStochasticPolicy, 11, 13
- CRBFFeatureCalculator, 6
- CRewardFunction, 7, 9, 12
- CSarsaLearner, 13
- CSemiMDPListener, 11
- CSemiMDPRewardListener, 7, 12
- CSingleStateDiscretizer, 4
- CSingleStateRBFFeatureCalculator, 6
- CSoftMaxDistribution, 11
- CStateCollection, 4
- CStateModifier, 4
- CStateProperties, 3
- CTilingFeatureCalculator, 5
- CTorchGradientFunction, 10
- CTransitionFunction, 2, 6
- CTransitionFunctionEnvironment, 3
- CValueCalculator, 15
- CVFunctionLearner, 13
- CVMStochasticPolicy, 11
- CVTable, 10

Controller, 11, 13

Discount Factor, 12, 14
Discretization, 4
doControllerEpisode, 14, 15
doControllerStep, 14
doNextState, 2, 6
doResetModel, 7

ϵ-greedy, 11, 14
Eligibility Traces, 13
Environment, 2–9
ETraceMaxListSize, 13
ETraceTreshold, 13
Evaluation, 14
Exploration, 11, 12, 14

Failed States, 7, 14
Function Approximation, 4–6, 10

getActions, 10
getDiscreteStateNumber, 4
getETraces, 13
getModifiedState, 4
getNextAction, 11
getResetState, 7
getReward, 7
getState, 3
getVETraces, 13
Gridworld, 7–9

Internal State, 3
isFailed, 14
isFailedState, 7
isReset, 14
isResetState, 7

λ, 13–14
Learning, 12–15
Learning Parameters, 12, 13
Learning Rate, 12, 14
Linear Features, 4–6
Linear Function Approximation, 4–6, 10
Listeners, 2, 11
Logger, 12

MLP, 10

Neural Networks, 6, 10
newEpisode, 12
nextStep, 11

Or-Operator, 5

`properties`, 3

Q-Functions, 10, 13
Q-Learning, 13

Radial Basis Functions (RBFs), 4, 5, 10
receiveError, 15
Replacing Traces, 13
Reset States, 7, 14
Reward Function, 12
Reward Functions, 7
Reward Listener, 7, 12

SARSA, 13
saveBIN, 12
saveData, 12
setContinuousState, 3
setController, 11
setDiscreteState, 3
setDiscreteStateSize, 3
setLambda, 13
setMaxValue, 3
setMinValue, 3
setParameter, 11, 13
setPeriodicity, 3
setReplacingETraces, 13
setRewardBounce, 9
setRewardForSymbol, 9
setRewardStandard, 9
setRewardSuccess, 9
setVFunction, 10
SoftMax, 11, 14
startNewEpisode, 14
State, 3–6
State Collection, 4
State Modifiers, 4–6, 10
State Properties, 3
State Substitution, 4

Temporal Difference Learning, 13
Tile Coding, 4, 5
Torch, 10
Transition Function, 2, 6–7
transitionFunction, 6

Value Functions, 10, 11, 13

References

[1] G. Neumann. *The Reinforcement Learning Toolbox, Reinforcement Learning for Optimal Control Tasks.* MSc Thesis, Graz, 2005.

[2] R. Collobert, S. Bengio, J. Mariethoz. *Torch: a modular machine learning software library.* Technical Report IDIAP-RR 02-46, IDIAP Martigny (CH), 2002. Available at `http://www.torch.ch/`