

Reinforcement Learning Toolbox Tutorial

Institute for Theoretical Computer Science

TU-GRAZ

Gerhard Neumann

November 3, 2006

1 The Pole Balancing example

The multipole example was taken from ?. The task is to balance a pole hinged to a cart by accelerating and slowing down the cart. If the pole falls past a given angle or moves through the wall ($x \pm 2.4$), a negative reward is given and the cart is reset. There is also a c-source code example file from R. Sutton, where an actor critic algorithm is used. This source code can be downloaded [here](#). We will try 2 algorithms, Q-Learning and the Algorithm used by R.Sutton. The source code of this example can be found in the examples directory of the toolbox.

2 Creating the Environment

First of all we need an environment where the agent can act and which describes the state transitions. Therefore we need to derive a new environment model class from the super-class CEnvironmentModel. The user has to provide methods for the internal state transitions, resetting the model and fetching the model state into a state object. This functionality is done by 3 functions which have to be overridden.

- The doNextState(CPrimitiveAction *) function has to calculate the internal state transitions. To indicate that the model has to be reset after this step we have to set the

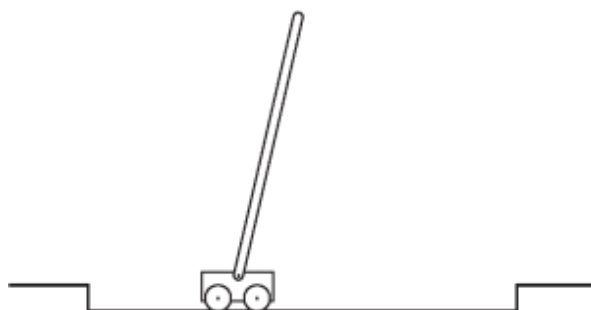


Figure 1: The pole balancing task

reset flag to true, to indicate that the episode failed, you set the failed flag. Actually both flags have the same effect, the failed flag is just used for logging some statistics. The information whether the current state is a reset state is important for the most value based learning algorithms, because they are not allowed to consider the value of the current state in their updates any more (actually, because the episode has ended in the current state, the value is zero).

- The `getState(CState *state)` function allows the agent to fetch the current state. The internal state variables have to be written in the *state object*.
- *doResetModel(): Here we have to reset the internal model variables.*

In the pole balancing task the environment has no discrete state variables and 4 continuous state variables, the position and velocity of the cart, the angle and the angular velocity of the pole. The header of our environment class can be seen in the following listing, the classes for this example are part of the toolbox and can be found in the files *cmultipole.cpp* and *cmultipole.h*.

```

class CMultiPoleModel : public CEnvironmentModel, public
    CRewardFunction
{
    protected:
        /// internal state variables
        double x, x_dot, theta, theta_dot;
        /// calculate the next state based on the action
        virtual void doNextState(CPrimitiveAction *action);

    public:
        CMultiPoleModel();
        virtual ~CMultiPoleModel();

        ///returns the reward for the transition, implements the
        CRewardFunction interface
        virtual double getReward(CStateCollection *oldState,
            CAction *action, CStateCollection *newState);
        ///fetches the internal state and stores it in the state
        object
        virtual void getState(CState *state);
        ///resets the model
        virtual void doResetModel();
};

```

In the constructor of our model class we have to set the properties of the model state. The properties of the model state specify how many continuous and discrete state variables the model has (in our case 4 continuous state variables). Additionally we define the minimum and maximum values for our continuous state variables. When using discrete states, we also have to set the discrete state sizes. Continuous state variables always have to be in there given interval (if a state variable exceeds its limits, it gets automatically set to the minimum or maximum value).

```

CMultiPoleModel::CMultiPoleModel() : CEnvironmentModel(4, 0)
{
    x= x_dot = theta = theta_dot = 0;

    properties->setMinValue(0, -2.4 * 1.1);
    properties->setMaxValue(0, 2.4 * 1.1);

    properties->setMinValue(1, -2);
    properties->setMaxValue(1, 2);

    properties->setMinValue(2, -twelve_degrees * 1.1);
    properties->setMaxValue(2, twelve_degrees * 1.1);

    properties->setMinValue(3, -fifty_degrees * 1.5);
    properties->setMaxValue(3, fifty_degrees * 1.5);
}

```

2.1 Resetting the model

Resetting the model is the easiest part of the environment. The method *doResetModel* is called each time the agent has failed or the user wants to start a new episode (this is done by the agent's *startNewEpisode* method), so we need to override this function. Our implementation sets all internal state variables to zero

```

void CMultiPoleModel::doResetModel()
{
    /// Reset internal state variables
    x = x_dot = theta = theta_dot = 0;
}

```

2.2 Fetching the state

The agent and the learning algorithm need to know the current state, for that reason we have to provide the *getState(CState *state)* method. In this method, the current state is written in the given state object. The CState class is a universal interface for states in the toolbox. A state object can contain an arbitrary number of continuous and discrete states. The properties of the state object, which is passed to the *getState* function have already been specified in the constructor of the model (so the *state* object has 4 continuous and 0 discrete state variables). All we need to do is to set the continuous state variables of the state object. This can be done with the method *setContinuousState(int dim, double value)*.

```

/// Store the model state to the given state object
void CMultiPoleModel::getState(CState *state)
{
    /// initializes the state object
    CEnvironmentModel::getState(state);

    /// Set the 4 internal state variables to the

```

```

    // continuous state variables of the model state
    state->setContinuousState(0, x);
    state->setContinuousState(1, x_dot);
    state->setContinuousState(2, theta);
    state->setContinuousState(3, theta_dot);
}

```

2.3 Executing an Action

The actions are executed in the method `doNextState(CPrimitiveAction *action)`. But before we can execute an action we implement our own action class.

2.3.1 Derivating the Action class

In our pole balancing example we have 2 different actions, one for accelerating in the positive x-direction and one for accelerating in the negative x-direction. We have to store the *force* which is applied to cart in our action objects. Therefore we have to derivate the *CPrimitiveAction* class and add a force data-element. Alternatively we could also use continuous actions (see the *CContinuousAction* class), but for simplicity we use an individual class here.

```

CMultiPoleAction::CMultiPoleAction(double force) : CPrimitiveAction
()
{
    this->force = force;
}

double CMultiPoleAction::getForce()
{
    return this->force;
}

```

2.3.2 Specifying the transition function: The *doNextState* Method

Whenever the agent wants to execute an action, the *doNextState* method of the environment class is called, with the action as argument. This function has to implement the internal state transition. Since we get a *CPrimitiveAction* object we first have to cast it to a *CMultiPoleAction* object. If we have more than one action class you can use the type field of the action to determine the class of the action. After casting we can retrieve the force from the action and accelerate the cart according to the force.

If the model should to be resettled in the next turn (the agent has failed to balance the pole), we have to set the reset flag, if we want additionally indicate that the episode has failed, we have to set the failed flag too. In our case the model is resettled each time the cart leaves the specified area, or the absolute angle of the pole is higher than 12 degree. In these cases the agent didn't manage to balance the pole correctly.

```

void CMultiPoleModel::doNextState(CPrimitiveAction *act)
{
    double xacc, thetaacc, force, costheta, sintheta, temp;

```

```

// cast the action to CMultiPoleAction
CMultiPoleAction* action = (CMultiPoleAction*)(act);
// determine the force
force = action->getForce();

// calculate the new state
costheta = cos(theta);
sintheta = sin(theta);
temp = (force + POLEMASS*LENGTH * theta_dot * theta_dot *
        sintheta) / TOTALMASS;
thetaacc = (GRAVITY * sintheta - costheta* temp) / (LENGTH
        * (FOURTHIRDS - MASSPOLE * costheta * costheta /
        TOTALMASS));
xacc = temp - POLEMASS*LENGTH * thetaacc* costheta /
        TOTALMASS;

// Update the four state variables
x += TAU * x_dot;
x_dot += TAU * xacc;
theta += TAU * theta_dot;
theta_dot += TAU * thetaacc;

// determine whether the episode has failed
if (x < -2.4 || x > 2.4 || theta < -twelve_degrees ||
    theta > twelve_degrees)
{
    reset = true;
    failed = true;
}
// indicate that a new episode has begun
if (reset)
{
    printf("Failed State: x=%f; theta=%f\n", x,
        theta);
}
}

```

Our model for navigating the agent is finished, now we need to discretize our state and calculate the reward.

3 Discretization of the State

To begin with, we will use a standard discrete state representation. Our discrete state representation has 163 states (it is the same state discretization as it is used in ?). The position, the velocity of the cart and the angle velocity of the pole gets divided into 3 partitions, the angle of the pole into 6. This gives us a discrete state size of $3 \cdot 3 \cdot 3 \cdot 6 = 162$ states. The drawback of this coarse partitioning is that the problem loses its Markov property, thus our

RL algorithm is not guaranteed to converge at all.

In the RL toolbox we have 2 possibilities to create our state discretization. We can use the build in classes for state discretization, which offers us enough functionality for the most applications, or, for some special purposes, if we need to use individual state discretization classes, we have to derivate a discretizer class from the class *CAbstractStateDiscretizer* and implement the *getDiscreteStateNumber* method, which has to return the index of the discrete state. But in this tutorial we will use the build-in classes.

3.1 Using the build in classes

With the build in classes we have the possibility to discretize single continuous state variables of the model state (with *CSingleStateDiscretizer*). Here we have to provide the partitions for each continuous state variable as a double array. These discrete states, which discretize only one state variable, can then be combined by the "and" operator (*CDiscreteStateOperatorAnd*) into a global discrete state.

In our main function we have to generate the discretizer classes. The single state discretizer class takes the index of the continuous state variable, the size of the partition array and the partition array itself as arguments. Since only the limits of the partitions are stored in the partitions array, the discrete state size of a single state discretizer is the size of the partition array plus one. After the generation of the single state discretizers, we create our *and* operator to merge them. When the construction of the discretizer is finished, it is essential for the state architecture of the toolbox that the discretizer gets added to the agent's state modifier list. No modifier can be used for learning if it hasn't been added to the agent's state modifier list.

```
// create the discretizer with the build in classes
// create the partition arrays
double partitions1 [] = { -0.8, 0.8}; // partition for x
double partitions2 [] = { -0.5, 0.5}; // partition for x_dot
double partitions3 [] = { -six_degrees, -one_degree, 0, one_degree,
    six_degrees}; // partition for theta
double partitions4 [] = { -fifty_degrees, fifty_degrees}; //
    partition for theta_dot

// Create the discretizer for the state variables
CAbstractStateDiscretizer *disc1 = new CSingleStateDiscretizer(0,
    2, partitions1);
CAbstractStateDiscretizer *disc2 = new CSingleStateDiscretizer(1,
    2, partitions2);
CAbstractStateDiscretizer *disc3 = new CSingleStateDiscretizer(2,
    5, partitions3);
CAbstractStateDiscretizer *disc4 = new CSingleStateDiscretizer(3,
    2, partitions4);

// Merge the 4 discretizer
```

```

CDiscreteStateOperatorAnd *andCalculator = new
    CDiscreteStateOperatorAnd();

andCalculator->addStateModifier(disc1);
andCalculator->addStateModifier(disc2);
andCalculator->addStateModifier(disc3);
andCalculator->addStateModifier(disc4);

discState = andCalculator;

```

4 Calculating the reward

Now only the reward calculation is missing. In our example we will implement the reward function in the environment model class, but the reward function can also be implemented in a separate class. In the derived reward function class we have to implement the function *getReward(CStateCollection *oldState, CAction *action, CStateCollection *newState)*.

The reward function gets the old state (as *CStateCollection* object), the current action and the new state as arguments. A state collection object is a collection of state objects, containing the model state and all states coming from the state modifiers which have been added to the agent's state modifier list. The state objects can be retrieved by passing a pointer to the state properties of the desired state object to the *getState* method of the state collection. In our case we want to retrieve the model state from the state collection, so we use the state properties of the environment model as argument for the *getState* method. If we specify a NULL pointer as argument for the *getState* method, always the model state is returned.

The reward function returns a reward of -1.0 if the agent has failed, otherwise the reward is 0.0 . Whether the agent has failed or not is stored in a flag in each state object, this flag can be retrieved by the method *isResetState*. Alternatively we could recalculate whether the agent has failed by retrieving the state variables from the current state object.

```

double CMultiPoleModel::getReward(CStateCollection *, CAction *,
    CStateCollection *newStateCol)
{
    double rew = 0.0;
    CState *newState = newStateCol->getState(getStateProperties
        ());

    // calculate the reward:
    // -1: for failed
    // 0 : else
    if (newState->isResetState())
    {
        rew = - 1.0;
    }
    else rew = 0.0;
    return rew;
}

```

5 Using the TD-Learning algorithm

The environment is finished, now we still have to merge all parts in our main function.

In the beginning, we create the environment model, then the agent and the action objects can be created. We create two actions, with the forces ± 10 . These action objects have to be added to the agent's action set. Every primitive action which can be executed by the agent has to be in this action set. After that we can create our state discretizer.

```
// create the model
model = new CMultiPoleModel();
// initialize the reward function
/* Our environment model also implements the reward function
   interface */
rewardFunction = model;

// create the agent
agent = new CAgent(model);

// add the discrete state to the agent's state modifier
// discState must not be modified (e.g. with a State-Substitution)
// by now
agent->addStateModifier(discState);

// create the 2 actions for accelerating the cart and add them to
// the agent's action set
CPrimitiveAction *primAction1 = new CMultiPoleAction(10.0);
CPrimitiveAction *primAction2 = new CMultiPoleAction(-10.0);

agent->addAction(primAction1);
agent->addAction(primAction2);
```

Now we create the TD-Learner. The TD Learner first needs a Q-Function object. Here we use a Feature-Q-Function which is a tabular representation of a Q-Function, Feature Q-Functions represents linear function approximators (discrete states can be seen as special case of a linear function approximator).

This Q-Function object is initialized with the state discretizer as parameter, so the Q-Function can determine the discrete state size and is also able to retrieve the discrete state from the state collection. After the Q-Function is created we can create the TD-Learner object. We use a Q-Learning algorithm, the learning algorithm needs the Q-Function and the reward function as argument. Having the learner we have to add it to the agents listener list. The agent always sends the step information (i.e. $\langle s_t, a_t, s_{t+1} \rangle$) to all his listeners, with this information the learning algorithm can update the Q-Function. We also set some parameters of the learner.

```
// Create the learner and the Q-Function
CFeatureQFunction *qTable = new CFeatureQFunction(agent->getActions
(), discState);
```

```

CTDLearner *learner = new CQLearner(rewardFunction, qTable);
// initialise the learning algorithm parameters
// initialise the learning algorithm parameters
learner->setParameter("QLearningRate", 0.1);
learner->setParameter("DiscountFactor", 0.99);
learner->setParameter("ReplacingETraces", 0.0);
learner->setParameter("Lambda", 1.0);

// Set the minimum value of a etrace, we need very small values
learner->setParameter("ETraceTreshold", 0.00001);
// Set the maximum size of the etrace list, standard is 100
learner->setParameter("ETraceMaxListSize", 163);

// add the Q-Learner to the listener list
agent->addSemiMDPListener(learner);

// Create the learners controller from the Q-Function, we use a
  SoftMaxPolicy
policy = new CQStochasticPolicy(agent->getActions(), new
  CEpsilonGreedyDistribution(0.1), qTable);

// set the policy as controller of the agent
agent->setController(policy);

```

Parameters are always set with the method *setParameter(paramName, paramValue)*, which is the general interface of the toolbox for parameter setting. For more details about the parameter handling of the toolbox please consult the class reference of *CParameters* and *CParameterObject*. Additionally all parameters of a class are specified in the class reference. There is a listening of all Parameters of a class at the end of the description.

The parameter setting in our example were roughly optimized by trial and error.

Now the agent controller is still missing. The agent controller returns an action to execute in the current state. We use a stochastic controller. A stochastic controller chooses its actions according to a given distribution. This distribution depends usually on Q-Values and can be specified by the user. We can use a Greedy Distribution, an Epsilon-Greedy Distribution or a Soft-Max Distribution. Here we choose a Soft-Max policy to provide a good mixture of exploration and exploitation.

Having created the controller we can start learning. Therefore we use the method *doControllerEpisode(nEpisodes, nStepsPerEpisode)* of the agent class. This method simulates *nEpisodes*, each episodes having *nStepsPerEpisode* steps (or less if the episode fails earlier). In our case, we learn for 500 episodes, and stop if the algorithm manages to balance the pole for more than 100000 steps.

```

int steps = 0;

int max_Steps = 100000;
// Learn for 500 Episodes
for (int i = 0; i < 500; i++)
{

```

```

// set adaptive Epsilon
policy->setParameter("EpsilonGreedy", 0.1 / (i + 1));
// Do one training trial, with max max_Steps steps
steps = agent->doControllerEpisode(1, max_Steps);

printf("Episode %d_%s with %d_steps\n", i, model->isFailed
      () ? "failed" : "succeeded", steps);

if (steps >= max_Steps)
{
    printf("Learned to balance the pole after %d_
          Episodes\n", i);
    break;
}
}

```

5.1 Results of the Q-Learning algorithm

Now we can compile and execute our program. Here is the output of 5 different trials:

```
--< Reinforcement Learning Benchmark - Pole Balancing with Q-Function Learning >=
```

```

Failed State: x = 0.171890; theta = -0.263239
Episode 0 failed with 12 steps
Failed State: x = 0.140651; theta = -0.215186
Episode 1 failed with 9 steps
Failed State: x = 0.151481; theta = -0.213644
Episode 2 failed with 19 steps
.
.
.
Episode 101 failed with 32459 steps
Failed State: x = 1.047413; theta = -0.215213
Episode 102 failed with 32576 steps
Failed State: x = 0.382649; theta = 0.250533
Episode 103 failed with 41403 steps
Failed State: x = 2.417235; theta = -0.087960
Episode 104 failed with 38826 steps
Episode 105 succeeded with 100000 steps
Learned to balance the pole after 105 Episodes

```

```
<< Press Enter >>
```

```
--< Reinforcement Learning Benchmark - Pole Balancing with Q-Function Learning >=
```

```

Failed State: x = 0.152474; theta = -0.235747
Episode 0 failed with 10 steps

```

Failed State: $x = 0.152485$; $\theta = -0.236025$
Episode 1 failed with 10 steps
Failed State: $x = 0.151481$; $\theta = -0.213644$
Episode 2 failed with 19 steps
Failed State: $x = -0.152485$; $\theta = 0.236025$
Episode 3 failed with 10 steps

.
.
.

Failed State: $x = -2.441029$; $\theta = -0.067404$
Episode 496 failed with 700 steps
Failed State: $x = -2.441029$; $\theta = -0.067404$
Episode 497 failed with 700 steps
Failed State: $x = -2.441029$; $\theta = -0.067404$
Episode 498 failed with 700 steps
Failed State: $x = -2.441029$; $\theta = -0.067404$
Episode 499 failed with 700 steps

<< Press Enter >>

--< Reinforcement Learning Benchmark - Pole Balancing with Q-Function Learning >=--

Failed State: $x = 0.140651$; $\theta = -0.215186$
Episode 0 failed with 9 steps
Failed State: $x = 0.147725$; $\theta = -0.211095$
Episode 1 failed with 20 steps
Failed State: $x = 0.151481$; $\theta = -0.213644$
Episode 2 failed with 19 steps
Failed State: $x = -0.152485$; $\theta = 0.236025$
Episode 3 failed with 10 steps
Failed State: $x = -0.384895$; $\theta = 0.228260$
Episode 4 failed with 75 steps

.
.
.

Episode 162 failed with 2435 steps
Failed State: $x = 0.918521$; $\theta = -0.214989$
Episode 163 failed with 1520 steps
Failed State: $x = 0.984002$; $\theta = -0.215042$
Episode 164 failed with 1721 steps
Failed State: $x = 0.999411$; $\theta = -0.210668$
Episode 165 failed with 64340 steps
Episode 166 succeeded with 100000 steps
Learned to balance the pole after 166 Episodes

<< Press Enter >>

--< Reinforcement Learning Benchmark - Pole Balancing with Q-Function Learning >=

Failed State: x = 0.140651; theta = -0.215186
Episode 0 failed with 9 steps
Failed State: x = 0.137344; theta = -0.223039
Episode 1 failed with 19 steps
Failed State: x = 0.249749; theta = -0.215719
Episode 2 failed with 28 steps
Failed State: x = -0.140651; theta = 0.215186
Episode 3 failed with 9 steps
Failed State: x = -0.130046; theta = 0.222500

.
. .

Episode 224 failed with 351 steps
Failed State: x = 1.248260; theta = 0.224913
Episode 225 failed with 928 steps
Failed State: x = 0.864278; theta = -0.212732
Episode 226 failed with 366 steps
Failed State: x = 2.409275; theta = -0.032653
Episode 227 failed with 2500 steps
Failed State: x = 1.432316; theta = -0.217736
Episode 228 failed with 1358 steps
Episode 229 succeeded with 100000 steps
Learned to balance the pole after 229 Episodes

--< Reinforcement Learning Benchmark - Pole Balancing with Q-Function Learning >=

Failed State: x = 0.160274; theta = -0.247547
Episode 0 failed with 10 steps
Failed State: x = 0.152485; theta = -0.236025
Episode 1 failed with 10 steps
Failed State: x = 0.191981; theta = -0.224828
Episode 2 failed with 24 steps
Failed State: x = -0.152485; theta = 0.236025
Episode 3 failed with 10 steps
Failed State: x = 0.140986; theta = -0.222948
Episode 4 failed with 12 steps

.
. .

Episode 136 failed with 71219 steps
Failed State: x = -2.403256; theta = 0.109602
Episode 137 failed with 7597 steps
Failed State: x = -2.417099; theta = 0.072864
Episode 138 failed with 5842 steps
Failed State: x = -0.549189; theta = -0.237117

```

Episode 139 failed with 5691 steps
Failed State: x = -2.400209; theta = 0.117046
Episode 140 failed with 59091 steps
Episode 141 succeeded with 100000 steps
Learned to balance the pole after 141 Episodes

```

As we can see the results are very different. The best trial finished in 105 steps, in the worst trial the algorithm didn't succeed at all (after 500 episodes). This behavior can be explained by the coarse discrete state representation which was used. The results are likely to be improved when the parameters are adjusted more accurately.

6 Using the Actor Critic Algorithm

As comparison to the Q-Learning algorithm we will also use the Actor-Critic algorithm used by R.Sutton. Actor Critic algorithms use a V-Function as "Critic", which critiques the actor's actions. In this example, a very simple actor, which uses only action values for one action is used. This actor chooses the action the following way: If the action value of action a_1 is positive, action a_1 is more likely to be taken, otherwise action a_2 is more likely.

It is very easy to use other algorithms in the Toolbox, we just have to change the TD-Learning part. For the actor critic learning algorithm we need now 2 learner. A V-Function learner, which learns the critic's V-Function separately, and an actor which learns the policy. The actor itself also need a V-Function which represents the action value for action a_1 .

The toolbox provides a general architecture for actor critic learning. Each TD-Learner (either a V-Function or a Q-Function learner) implements an extra interface, which is called *CErrorSender*. Each time it calculates a td value, it sends this information in combination with the state s_t and the action a_t to all error listeners which have been added to the error sender interface. Error listeners (class *CErrorListener*) therefore receive the tuple $\langle s_t, a_t, td \rangle$ from the TD learner at every time step. This is all information needed by the most actors, so all actor classes are implemented by this interface.

Due to this architecture, we only have to add the actor to the error sender interface of the TD-learner to establish learning. The actor, which is used in this example, simultaneously implements a policy and can therefore be used as agent controller.

```

// Create the learner and the Q-Function
CFeatureVFunction *vTableActor = new CFeatureVFunction(discState);
CFeatureVFunction *vTableCritic = new CFeatureVFunction(discState);

CVFunctionLearner *vLearner = new CVFunctionLearner(rewardFunction,
    vTableCritic);
CActorFromActionValue *actor = new CActorFromActionValue(
    vTableActor, primAction1, primAction2);

vLearner->addErrorListener(actor);

vLearner->setParameter("VLearningRate", 0.1);
vLearner->setParameter("Lambda", 0.85);

```

```

vLearner->setParameter("ReplacingETraces", 0.0);

actor->setParameter("ActorLearningRate", 100.0);
actor->setParameter("ReplacingETraces", 0.0);
actor->setParameter("Lambda", 0.95);

agent->addSemiMDPListener(vLearner);

// set the policy as controller of the agent
agent->setController(actor);

```

6.1 Results of the Actor Critic Algorithm

Again we show here the results of 5 different trials:

```
--< Reinforcement Learning Benchmark - Pole Balancing with Actor-Critic Learning >--
```

```

Failed State: x = -0.153297; theta = 0.254336
Episode 0 failed with 18 steps
Failed State: x = 0.133458; theta = -0.217305
Episode 1 failed with 13 steps
Failed State: x = 0.166574; theta = -0.225796
Episode 2 failed with 22 steps
Failed State: x = -0.133486; theta = 0.218084
.
.
.
Failed State: x = -1.125134; theta = -0.224579
Episode 71 failed with 8393 steps
Failed State: x = -2.266797; theta = -0.217293
Episode 72 failed with 4501 steps
Failed State: x = -0.052670; theta = -0.211017
Episode 73 failed with 3416 steps
Episode 74 succeeded with 100000 steps
Learned to balance the pole after 74 Episodes

```

```
--< Reinforcement Learning Benchmark - Pole Balancing with Actor-Critic Learning >--
```

```

Failed State: x = 0.136852; theta = -0.211665
Episode 0 failed with 10 steps
Failed State: x = -0.141241; theta = 0.228731
Episode 1 failed with 13 steps
Failed State: x = -0.030081; theta = -0.220379
Episode 2 failed with 39 steps
Failed State: x = -0.208044; theta = 0.258420
.

```

.
.
Episode 44 failed with 345 steps
Failed State: x = -1.129284; theta = -0.219223
Episode 45 failed with 15076 steps
Failed State: x = -0.769336; theta = -0.212685
Episode 46 failed with 9012 steps
Episode 47 succeeded with 100000 steps
Learned to balance the pole after 47 Episodes

--< Reinforcement Learning Benchmark - Pole Balancing with Actor-Critic Learning >--

Failed State: x = 0.056877; theta = -0.213038
Episode 0 failed with 30 steps
Failed State: x = 0.077752; theta = 0.211264
Episode 1 failed with 41 steps
Failed State: x = 0.144699; theta = -0.224493
Episode 2 failed with 11 steps

.
.
.
Failed State: x = 0.912410; theta = 0.212099
Episode 89 failed with 9358 steps
Failed State: x = -2.408235; theta = -0.001478
Episode 90 failed with 26745 steps
Failed State: x = -2.414515; theta = 0.125916
Episode 91 failed with 46321 steps
Episode 92 succeeded with 100000 steps
Learned to balance the pole after 92 Episodes

--< Reinforcement Learning Benchmark - Pole Balancing with Actor-Critic Learning >--

Failed State: x = -0.152709; theta = 0.241117
Episode 0 failed with 19 steps
Failed State: x = 0.141109; theta = -0.225776
Episode 1 failed with 12 steps
Failed State: x = -0.155462; theta = 0.221767
Episode 2 failed with 48 steps
Failed State: x = 0.129589; theta = -0.212119

.
.
.
Failed State: x = 0.905785; theta = 0.229068
Episode 130 failed with 5690 steps
Failed State: x = 2.403843; theta = -0.141673

```
Episode 131 failed with 5212 steps
Episode 132 succeeded with 100000 steps
Learned to balance the pole after 132 Episodes
```

```
--< Reinforcement Learning Benchmark - Pole Balancing with Actor-Critic Learning >=--
```

```
Failed State: x = -0.071805; theta = 0.221187
Episode 0 failed with 27 steps
Failed State: x = 0.137015; theta = -0.215471
Episode 1 failed with 11 steps
Failed State: x = -0.140651; theta = 0.215186
Episode 2 failed with 9 steps
.
.
.
Failed State: x = 2.404692; theta = 0.019832
Episode 61 failed with 21950 steps
Failed State: x = -2.416398; theta = -0.015000
Episode 62 failed with 13304 steps
Episode 63 succeeded with 100000 steps
Learned to balance the pole after 63 Episodes
```

The results are again quite different, but we can see that this algorithm, even if the actor is very simple (or even because of its simplicity) can cope better with the state discretization than the Q-Learning algorithm. But we also have to consider that the parameters of this algorithm are likely to be better optimized since they were taken from Sutton's example. It was observed that this algorithm is very sensitive to its parameter setting, so for example with the Lambda parameter of the critic set to 0.95 instead of 0.85 the algorithm's performance is much worse, which is actually quite surprising.

The main reason for the rather bad performance is the state discretization. We could use another state discretization with more states, but searching good state discretizations is very time consuming and we don't want to waste our time. We will take a short look on linear approximators and RL with continuous states.

7 Using Linear Approximators

We will use the Q-Learning algorithm again, but now we will use a RBF-network (with constant centers and shapes, so the RBF net is a linear approximator) as Q-Function. When we are dealing with linear approximators we need to use feature states. Each feature has an activation factor, which can be between 0 and 1. There can be several features active at the same time. In the toolbox, a feature state is represented, like all states with the *CState* object. The state object consists in this case of *numActive* discrete states (represent discrete state number of the feature) and the same number of continuous states (which represents the activation factors). *numActive* is the maximum number of active features. The discrete and continuous state variables with the same index belong to one feature. All feature indices

which are not listed in the current state object are assumed to have an activation factor of zero. It is usually the case that many features of a feature state are inactive (like in RBF networks), thus using this sparse representation of the state is more efficient.

We will use a straight forward RBF-network approach with an uniform distributed grid ($5 \cdot 5 \cdot 10 \cdot 10$) of RBF-Centers. Consequently our feature state consists of 2500 features. In the toolbox we have 2 possibilities to build RBF-networks. We can use a uniform grid as we will do in this example, or we can define the RBF-centers individually for each state variable (similar to *CSingleStateDiscretizer*), which is certainly more powerful, but not needed in this example (see class reference of *CSingleStateFeatureCalculator*) for further details.

For a simple uniform grid we will use the class *CRBFFeatureCalculator*. Here we have to define which of the continuous state variables we want to use for our RBF-Grid (we want to use all of them), how much RBF-Functions are used for each dimension, an offset for each dimension, which is added to the location of the RBF-Centers, and the σ values for each dimension (specifying the shape of the RBF-Function). The class refers the σ values always to the normalized state variables (normalized state variables are scaled in to the interval $[0,1]$). We won't use any offsets, for the σ values a good rule of thumb is that the RBF-centers should be in a distance of $2 \cdot \sigma$ in each dimension (so $2 \cdot \sigma_i \cdot numRBFFunc_i$ should be approximately 1). The number of active features is calculated for each dimension separately, each feature is considered as active if it is within a distance of $2\sigma_i$ for each dimension from the current state. With the specified σ values this gives us 3 active features per dimension. We have to be careful, wrongly chosen σ values can drastically increase the number of active features. It is recommendable to check the number of active feature before starting learning the first time. In our case, we have $3 \cdot 3 \cdot 3 \cdot 3 = 81$ active features at the same time.

The RBF-network class also normalizes all features after calculating the active factors, so all factors always sum to 1.0.

Having created the RBF feature calculator we can use it to create our feature Q-Function. We will set 2 additional parameters of our learner, "ETraceMaxListSize" is the maximum number of features that can be stored in the etrace list. The standard value is 100, we have to increase this value because we have a lot more features than in the previous example. The second parameter is "ETraceTreshold", which is the minimum value of a feature in he etrace list before it gets deleted from the list. Since we have 81 features at the same time, the activation factors will be much smaller than 1, so we also have to decrement this value (standard is 0.001). The remaining code is the same as used for our previous Q-Learning example, we just replace *discState* with *rbfCalc*.

```

unsigned int dimensions [] = {0, 1, 2, 3};
unsigned int partitions [] = {5, 5, 10, 10};
double offsets [] = {0.0, 0.0, 0.0, 0.0};
double sigma [] = {0.1, 0.5, 0.05, 0.05};

```

```

CFeatureCalculator *rbfCalc = new CRBFFeatureCalculator(4,
    dimensions, partitions, offsets, sigma);

```

7.1 Results with the RBF-network

Here is the listening of 5 different trials:

```

--< Reinforcement Learning Benchmark - Pole Balancing with Q-Function Learning >--

```

Failed State: x = -0.144668; theta = 0.223803
Episode 0 failed with 10 steps
Failed State: x = 0.085883; theta = -0.210676
Episode 1 failed with 23 steps
Failed State: x = -1.285247; theta = -0.211621
Episode 2 failed with 153 steps
Failed State: x = -0.755425; theta = 0.218169
Episode 3 failed with 147 steps
Episode 4 succeeded with 100000 steps
Learned to balance the pole after 4 Episodes

--< Reinforcement Learning Benchmark - Pole Balancing with Q-Function Learning >=

Failed State: x = 0.144668; theta = -0.223803
Episode 0 failed with 10 steps
Failed State: x = 0.062283; theta = 0.232154
Episode 1 failed with 40 steps
Failed State: x = -0.140651; theta = 0.215186
Episode 2 failed with 9 steps
Failed State: x = 0.038721; theta = -0.213671
Episode 3 failed with 36 steps
Failed State: x = -2.406821; theta = 0.140414
Episode 4 failed with 188 steps
Failed State: x = -1.069869; theta = -0.229641
Episode 5 failed with 159 steps
Failed State: x = -2.407397; theta = -0.093332
Episode 6 failed with 24915 steps
Episode 7 succeeded with 100000 steps
Learned to balance the pole after 7 Episodes

--< Reinforcement Learning Benchmark - Pole Balancing with Q-Function Learning >=

Failed State: x = -0.137053; theta = 0.216335
Episode 0 failed with 11 steps
Failed State: x = 0.115486; theta = -0.222224
Episode 1 failed with 19 steps
Failed State: x = -2.406728; theta = -0.183478
Episode 2 failed with 181 steps
Failed State: x = -1.288102; theta = -0.228951
Episode 3 failed with 147 steps
Failed State: x = -1.198822; theta = 0.210250
.
.
.

Episode 164 failed with 739 steps
Failed State: x = 1.627081; theta = 0.214098
Episode 165 failed with 748 steps
Failed State: x = 1.940820; theta = -0.210778
Episode 166 failed with 1635 steps
Episode 167 succeeded with 100000 steps
Learned to balance the pole after 167 Episodes

--< Reinforcement Learning Benchmark - Pole Balancing with Q-Function Learning >=--

Failed State: x = -0.133752; theta = 0.223947
Episode 0 failed with 16 steps
Failed State: x = 0.133490; theta = -0.218071
Episode 1 failed with 13 steps
Failed State: x = -2.405397; theta = 0.189988
Episode 2 failed with 198 steps
Failed State: x = -2.015033; theta = 0.244611
Episode 3 failed with 151 steps
Failed State: x = -0.612253; theta = -0.215054
Episode 4 failed with 94 steps
Failed State: x = -1.667129; theta = -0.211936
Episode 5 failed with 302 steps
Episode 6 succeeded with 100000 steps
Learned to balance the pole after 6 Episodes

<< Press Enter >>

--< Reinforcement Learning Benchmark - Pole Balancing with Q-Function Learning >=--

Failed State: x = -0.126246; theta = 0.218807
Episode 0 failed with 20 steps
Failed State: x = 0.148828; theta = -0.235675
Episode 1 failed with 12 steps
Failed State: x = -2.291497; theta = -0.239118
Episode 2 failed with 1310 steps
Failed State: x = -1.843375; theta = 0.222521
.
.
.
Episode 35 failed with 418 steps
Failed State: x = 2.402953; theta = -0.122837
Episode 36 failed with 23731 steps
Failed State: x = 1.430086; theta = 0.226359
Episode 37 failed with 2369 steps
Failed State: x = 1.652282; theta = 0.225271
Episode 38 failed with 12398 steps

Episode 39 succeeded with 100000 steps
Learned to balance the pole after 39 Episodes

As we can see, the results are, except for one trial very good. We would have to adjust the policy parameter "SoftMaxBeta" (which was chosen very high, so almost a greedy policy was used) or perhaps use more features to get more uniform results.