1 The Pole Balancing example

The multipole example was taken from the book: [http://www-anw.cs.umass.edu/rich/book/the-book.html](http://www-anw.cs.umass.edu/rich/book/the-book.html) from R. Sutton and S. Barto. 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 the episode ends with a failure and the cart is reset. If the cart moves beyond given limits, the cart is reset with a failure too. If a failure occurs the agent gets a negative reward from -1. In this tutorial we will learn the pole balancing task with a TD-Learning algorithm.

2 Creating the Environment

First of all you need the environment where the agent can act and which describes the state transitions. Therefore you need to derivatise the class CEnvironmentModel. The user has to provide for doing 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(CPrimitivAction *) function has to calculate the internal state transitions. To indicate that the model has to be reset after this step you have to set the reset flag, to indicate that the episode failed, you set the failed flag.

- The getState(CState *state) function allows the agent to fetch the current state. The internal state variables has to be written in the "state" object.

- doResetModel(): Here you have to reset the internal model variables.

In the pole balancing task the environment has 4 state variables. The position and velocity of the cart, the angle and the angular velocity of the pole.

The header of the environment model looks like

```cpp
class CMultiPoleModel : public CEnvironmentModel, public CRewardFunction {
  protected:
    // internal state variables
    double x, x_dot, theta, theta_dot;
```
virtual void doNextState(CPrimitivAction* action);

/// needed to retrieve the calculated discrete state
CStateModifier* discState;

public:
CMultiPoleModel(CStateModifier* discState);
~CMultiPoleModel();

virtual double getReward(CStateCollection* oldState, CAction* action, CStateCollection* newState);

virtual void getState(CState* state);

virtual void doResetModel();

In the constructor of our model we have to set the properties of the model state. The properties of the model state consists of how many continuous and discrete state variables the model has. Additionally we define the minimum and maximum values for our continuous state, if we use discrete states too we also have to set the discrete state sizes.

CMultiPoleModel::CMultiPoleModel(CMultiPoleDiscreteState* discState)
    : CEnvironmentModel(4 /* num cont. states */, 0 /* num disc. states */) {
    this->discState = discState;
    x = x_dot = theta = theta_dot = 0;
properties->setMinValue(0, -2.4);
properties->setMaxValue(0, 2.4);

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

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

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

2.1 Resetting the model

Resetting the model is the easiest part of the environment creation. The method doResetModel is called each time the agent has failed, so we need to override this function.

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

2.2 Fetching the state

The agent and the learning algorithms need to know the current state, so you have to provide the getState(CState *) method. The CState object is passed from the agent and has exactly the properties of the defined model state. All you need to do is to set the continuous and discrete state variables of the state object.

```cpp
// Store the model state to the given state object
void CMultiPoleModel::getState(CState *state)
{
    /// resets 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 by the method doNextState(CPrimitivAction *action). But before we can execute an action we have to design or own action class.

```cpp
```
2.3.1 Derivating the Action class

In our pole balancing example we have 2 different actions, one for accelerating in the positiv x-direction and one for accelerating in the negativ direction. So we move the cart one time with a positiv force and the other time with a negativ force. We have to store the "Force" information in our action objects. Therefore we have to derivate the CPrimitivAction class and add the force field. Since

```cpp
class CMultiPoleAction : public CPrimitivAction
{
protected:
    double force;

public:
    CMultiPoleAction(double force);
    double getForce() { return force; };
};
```

2.3.2 Executing the action: The doNextState Methode

Whenever the agent wants to execute an action, the doNextState method is called, with the action as parameter. Since we get a CPrimitivAction object we first have to cast to CMultiPoleAction. If you 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 determine the "Force" parameter of the action and accelerate the cart according this force.

If the model has to be reseted in the next turn, you can set the reset flag, if you want additionally indicate that the episode has failed, you can set the failed flag.

```cpp
void CMultiPoleModel::doNextState(CPrimitivAction *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, using Euler's method. ***/
x += TAU * x_dot;
x_dot += TAU * xacc;
theta += TAU * theta_dot;
theta_dot += TAU * thetaacc;

// determine wether 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 only need to discretize our state and calculate the reward, nothing more to do.

3 Discretizing the State

In our example we will work with "normal" state discretization, but you can easily change the code and work with features. A State discretization with 163 states was choosen. The position, the velocity of the cart and the angle velocity of the pole gets partitioned into 3 states, the angle of the pole into 6. This

For this illustration example only few states were chosen to allow fast learning, but the drawbacks of this coarse partitioning is that the problem is strongly non-markovian, so not all learning algorithm can deal with it (e.g. model based algorithm won’t learn a optimal policy).

In the RIL toolbox we have 2 possibilities to create our state discretization. We can derivate our own state discretization class or we can use the build in classes. We will make both possibilities for illustration.

3.1 Creating your own Discretization class

To create our own discretization class we have to derivate the CAbstractStateDiscretizer class an implement the function getDiscreteStateNumber(CStateCollection *state), which returns our calculated state index. To retrieve the model state from the state collection, you just have to call the getState() method. If you want a different state from the state collection, you have to pass the state properties object to the getState method. We will need this for our reward calculation.

We will name our class CMultiPoleDiscreteState.

```cpp
class CMultiPoleDiscreteState : public CAbstractStateDiscretizer
{
  public:
    CMultiPoleDiscreteState();
};
```
virtual unsigned int getDiscreteStateNumber(CStateCollection *state) {

unsigned int CMultiPoleDiscreteState::getDiscreteStateNumber(CStateCollection *stateCol) {

    // get the model state
    CState *state = stateCol->getState();
    int box;

    // get the 4 continuous state variables
    double x = state->getContinuousState(0);
    double x_dot = state->getContinuousState(1);
    double theta = state->getContinuousState(2);
    double theta_dot = state->getContinuousState(3);

    if ((x < -2.4 || x > 2.4) || theta < -twelve_degrees || theta > twelve_degrees) {
        box = -1; /* to signal failure */
    } else {

        // partition x
        if (x < -0.8) box = 0;
        else if (x < 0.8) box = 1;
        else box = 2;

        // partition x_dot
        if (x_dot < -0.5);
        else if (x_dot < 0.5) box += 3;
        else box += 6;

        // partition theta
        if (theta < -six_degrees);
        else if (theta < -one_degree) box += 9;
        else if (theta < 0) box += 18;
        else if (theta < one_degree) box += 27;
        else if (theta < six_degrees) box += 36;
        else box += 45;

        // partition theta_dot
        if (theta_dot < -fifty_degrees);
        else if (theta_dot < fifty_degrees) box += 54;
        else box += 108;
    }

    // increase box because only positive values are allowed.
    box ++;

    return box;
}
3.2 Using the build in classes

The more comfortable way is to create the discrete state with the build in RIL class. Here you have the possibility to discretize single continous states of the model state (with CSingleStateDiscretizer). You just have to provide the partition for the continous states as double array. This single-discrete states can then be combined by the "and" operator (CDiscreteStateOperatorAnd). The only problem we have is the failed state, because it depends not on a single continous state. Therefore we have to create an own state discretizer class just for determining if the state has is a failed state or not (so it has only 2 discrete state). The non-failed state is then substituted by the calculated discrete state from the and operator.

```cpp
class CMultiPoleFailedState : public CAbstractStateDiscretizer
{
public:
    CMultiPoleFailedState();

    virtual unsigned int getDiscreteStateNumber(CStateCollection *state);
};
```

```cpp
unsigned int CMultiPoleFailedState::getDiscreteStateNumber(CStateCollection *stateCol)
{
    // get the model state
    CState *state = stateCol->getState();
    int box;
    double x = state->getContinuousState(0);
    double theta = state->getContinuousState(2);

    /// calculate wether the state is a failed state
    if (x < -2.4 || x > 2.4 || theta < -twelve.degrees || theta > twelve.degrees)
    {
        box = 0; /* to signal failure */
    }
    else
    {
        box = 1;
    }
    return box;
}
```

In our main function we have to generate the discretizer classes. The single state discretizer take the index of the continous state variable, the size of the partition array and the partition array itself as argument. Since in the partition array there are only stored the limits of the partitions, the discrete state size is the size of the array + 1.

```cpp
double partitions1[] = {-0.8, 0.8};
double partitions2[] = {-0.5, 0.5};
double partitions3[] = {-six.degrees, -one_degree, 0, one_degree, six_degrees};
double partitions4[] = {-fifty.degrees, fifty.degrees};
```
This creates the same discrete state is the CMultiPoleDiscreteState class, but its much more flexible and easier to change. For example you can use feature states instead of discrete states. You just have to use a CFeatureOperatorAnd as operator and you can use feature states like RBF networks for the single state discretization.

To use the state discretizer has to be added to the state modifier list of the agent. The agent then adds a state with the properties of the state modifier to the state collection and recalculates this modified state each time the model state changes, so the modified state is available to all listeners of the agent by the state collection.

4 Calculating the reward

We are almost finished. Only the reward calculation is missing. In our example we merged the reward function with the environment model, but this can also be separate classes if you want to have more than one reward function. In your reward function class you have to override the getReward Function.

Our reward function only returns a reward of -1.0 if the agent is in a failed state. The failed state calculation is already done by the discretizer (state 0 is the failed state), so we use fetch the discrete from the state collection to calculate the reward.

```cpp
double CMultiPoleModel::getReward(CStateCollection *oldStatecol, CAction *action)
{
    double rew;
    CState *newState = newStateCol->getState(discState);
    // calculate the reward:
    // -1: for failed
    // 0 : else
    if (newState->getDiscreteState(0) == 0)
    {
        rew = -2.0;
    }
    else rew = 0.0;
    return rew;
}
```
return rew;
}

5 Using the TD-Learning algorithm

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

For learning we need to create an agent who is navigating in our environment. We have to create the 2 actions for the agent and add them to the agents action set. Then we need to create the TD-Learner. The TD Learner first needs a Q-Function object with which he can learn. This Q-Function object is initialised 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 a state collection. After the Q-Function is created we can create the TD-Learner. We use a Q-Learning algorithm, so the estimation policy of the TD Learner is automatically a greedy policy. Having the learner we have to add the learner to the agents listener list. The agent always sends the steps (i.e. \( s_t, a_t, s_{t+1} \)) to his listener, with this information the learning algorithm can update their q-Function.

The last thing we need is an agent controller. The agent controller returns an action to execute for the current state. Since we want to evaluate the learning process we take a controller based on the Q-Function. We choose a softmax policy to provide a good mixture of exploration and exploitation.

Our main method looks like the following way:

```cpp
#include "cmultipolemodel.h"
#include "ctdlearner.h"
#include "cpolicies.h"
#include "cagent.h"

int main(void)
{
    // initialize the random
    srand((unsigned int)time((time_t*)NULL));

    // Model dependent Variables
    CEnvironmentModel *model = NULL;
    CAgentStatisticController *detController = NULL;
    CAbstractStateDiscretizer *discState = NULL;
    CRewardFunction *rewardFunction = NULL;
    CAgent *agent;

    // create the discretizer with the build in classes
    double partitions1[] = {-0.8, 0.8};
    double partitions2[] = {-0.5, 0.5};
    double partitions3[] = {-six_degrees, -one_degree, 0, one_degree, six_degrees};
    double partitions4[] = {-fifty_degrees, fifty_degrees};

    CAbstractStateDiscretizer *disc1 = new CSingleStateDiscretizer(0, 2, partitions1);
    CAbstractStateDiscretizer *disc2 = new CSingleStateDiscretizer(1, 2, partitions2);
    CAbstractStateDiscretizer *disc3 = new CSingleStateDiscretizer(2, 2, partitions3);
    CAbstractStateDiscretizer *disc4 = new CSingleStateDiscretizer(3, 2, partitions4);
```
CAbstractStateDiscretizer *disc3 = new CSingleStateDiscretizer(2, 7, partitions3);
CAbstractStateDiscretizer *disc4 = new CSingleStateDiscretizer(3, 2, partitions4);

CDiscreteStateOperatorAnd *andCalculator = new CDiscreteStateOperatorAnd();

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

discState = new CMultiPoleFailedState();
discState->addStateSubstitution(1, andCalculator);

// create the discrete state with the self-build class
/* not used
discState = new CMultiPoleDiscreteState();
*/

// create the model
model = new CMultiPoleModel(discState);
// initialise the reward function
/* reward function and model are merged*/
rewardFunction = (CMultiPoleModel *)model;

// create the agent
agent = new CAgent(model);
// add the discrete state to the agent's state modifier
agent->addStateModifier(discState);

// create the 2 actions for accelerating the cart and add them to the agent
CPrimitivAction *primAction1 = new CMultiPoleAction(10.0, failedState);
CPrimitivAction *primAction2 = new CMultiPoleAction(-10.0, failedState);
agent->addAction(primAction1);
agent->addAction(primAction2);

// Create the learner and the Q-Function
CFeatureQFunction *qTable = new CFeatureQFunction(agent->getActions(),
CAbstractTD Learner *learner = new CQLearner(rewardFunction, qTable);
// initialise the learning algorithm parameters
learner->setAlpha(0.1);
learner->getQFunction()->setGamma(0.95);
learner->getETraces()->setReplacingETraces(true);
learner->getETraces()->setLambda(0.9);

// add the Q-Learner to the listener list
agent->addSemiMDPLis tener(learner);
// Create the learners controller
CAgentStatisticController *policy = NULL;
policy = new CQGreedyPolicy(agent->getActions(), qTable);
// set the policy as controller of the agent
agent->setController(policy);

// disable automatic logging of the episode from the agent
agent->setLogEpisode(false);

int steps = 0;

// Learn for 500 Episodes
for (int i = 0; i < 500 && steps < 100000; i++)
{
    // Do one training trial, with max 100000 steps
    steps = agent->doControllerEpisode(1, 100000);
    printf("Episode%d%s with%d steps\n", i, model->isFailed() ? "failed" : "succeeded", steps);
}

delete policy;
delete learner;
delete agent;
delete qTable;
delete model;

printf("\n\n<Press Enter>\n");
getchar();