

Reinforcement Learning Toolbox Tutorial

Using the hierarchical architecture

Institute for Theoretical Computer Science

TU-GRAZ

Gerhard Neumann

November 3, 2006

1 The Taxi Problem

The Taxi Problem is a well studied benchmark problem for hierarchical RL and has been defined in ?. The agent has to move in a grid-world, his task is to pick a passenger up at a given location and drop him down again at a given destination. In our example we will use a 10x10 grid-world with 4 different locations/destinations. The source code for this example can be found in the examples directory of the toolbox.

In this tutorial we will start with learning the taxi task with a flat learning architecture, and then we will build a hierarchic architecture for learning the navigation behaviors. In the end we will compare the results.

2 Learning the Taxi-Task

The taxi model has been derivated from the *CGridworldModel* class. The state space contains 2 additional discrete state variables, the passenger's location and the destination. Because we have 4 different locations, the state size of the first variable is 5 (the passenger can also be in the taxi) and the second variable has 4 different states. In addition to the state variables of the grid world (x, y, numBounces) we have 5 discrete state variables. For a 10x10 grid-world we already have $10 \cdot 10 \cdot 5 \cdot 4 = 2000$ states, which is quite a lot for such a small problem.

We also define two additional actions, one for *picking up* and one for *dropping down* the passenger. Both actions can be identified by the type field of the action class. The agent gets a reward of +50, if he managed to drop the passenger at the correct location, and a reward of -5 if he bounces against a wall or picks up/drops down the passenger when it is not allowed.

In order to define the grid-world, we can again use the grid-world file format.

```
Gridworld Size: 10x10
StartValues: 0
TargetValues: 1,2,3,4
ProhibitedValues: 9,8
```

```

9999999999
9000090009
9039000009
9999009009
9209099009
9000009049
9909999999
9000000009
9000000019
9999999999

```

For a description of the grid-world files see the shortest path tutorial.

The target values 1,2,3,4 specify the coordinates of the possible passenger locations/destinations.

```

CTaxiDomain::CTaxiDomain(char* filename) : CGridWorldModel(filename
, 100)
{
    targetXYValues = NULL;
    delete properties;
    properties = new CStateProperties(0, 5);
    initTargetVector();
}

```

```

CTaxiDomain::~CTaxiDomain()
{
    if (targetXYValues != NULL)
    {
        std::vector<std::pair<int, int> *>::iterator it =
            targetXYValues->begin();
        for (; it != targetXYValues->end(); it++)
        {
            delete *it;
        }
        delete targetXYValues;
    }
}

```

```

void CTaxiDomain::initTargetVector()
{
    if (targetXYValues == NULL)
    {
        targetXYValues = new std::vector<std::pair<int, int>
> *>();
    }
    else
    {
        std::vector<std::pair<int, int> *>::iterator it =

```

```

        targetXYValues->begin();
        for (; it != targetXYValues->end(); it++)
        {
            delete *it;
        }
        targetXYValues->clear();
    }

    for (unsigned int x = 0; x < this->getSizeX(); x++)
    {
        for (unsigned int y = 0; y < this->getSizeY(); y++)
        {
            if (target_values->find(getGridValue(x,y))
                != target_values->end())
            {
                targetXYValues->push_back(new std::
                    pair<int, int>(x,y));
            }
        }
    }
    properties->setDiscreteStateSize(0, getSizeX());
    properties->setDiscreteStateSize(1, getSizeY());
    properties->setDiscreteStateSize(2, 0);
    properties->setDiscreteStateSize(3, targetXYValues->size()
        + 1);
    properties->setDiscreteStateSize(4, targetXYValues->size());
    ;
}

void CTaxiDomain::load(FILE *stream)
{
    CGridWorldModel::load(stream);
    initTargetVector();
}

int CTaxiDomain::getTargetPositionX(int numTarget)
{
    return (*targetXYValues)[numTarget]->first;
}

int CTaxiDomain::getTargetPositionY(int numTarget)
{
    return (*targetXYValues)[numTarget]->second;
}

void CTaxiDomain::transitionFunction(CState *oldState, CAction *
    action, CState *newState, CActionData *data)

```

```

{
    if (action->isType(GRIDWORLDACTION))
    {
        CGridWorldModel::transitionFunction(oldState,
            action, newState, data);
    }
    int pos_x = oldState->getDiscreteState(0);
    int pos_y = oldState->getDiscreteState(1);

    int pasLocation = oldState->getDiscreteState(3);
    int pasDestination = oldState->getDiscreteState(4);

    if (action->isType(PICKUPACTION))
    {
        if (pasLocation < getNumTargets())
        {
            if (getTargetPositionX(pasLocation) ==
                pos_x && getTargetPositionY(pasLocation)
                == pos_y)
            {
                pasLocation = getNumTargets();
            }
        }
    }
    if (action->isType(PUIDOWNACTION))
    {
        if (pasLocation == getNumTargets())
        {
            if (getTargetPositionX(pasDestination) ==
                pos_x && getTargetPositionY(
                pasDestination) == pos_y)
            {
                pasLocation = pasDestination;
            }
        }
    }

    newState->setDiscreteState(3, pasLocation);
    newState->setDiscreteState(4, pasDestination);
}

bool CTaxiDomain::isResetState(CState *state)
{
    return (CGridWorldModel::isFailedState(state) || state->
        getDiscreteState(3) == state->getDiscreteState(4));
}

```

```

void CTaxiDomain::getResetState(CState *resetState)
{
    CGridWorldModel::getResetState(resetState);
    int location = rand() % targetXYValues->size();
    resetState->setDiscreteState(3, location);

    int target = rand() % targetXYValues->size();

    if (target == location)
    {
        target = (target + 1) % this->getNumTargets();
    }
    resetState->setDiscreteState(4, target);
}

double CTaxiDomain::getReward(CStateCollection *oldStateCol,
    CAction *action, CStateCollection *newStateCol)
{
    CState *oldState = oldStateCol->getState();
    CState *newState = newStateCol->getState();

    double reward = this->getRewardStandard();

    if (action->isType(GRIDWORLDACTION))
    {
        if (oldState->getDiscreteState(0) == newState->
            getDiscreteState(0) && oldState->
            getDiscreteState(1) == newState->
            getDiscreteState(1))
        {
            reward += this->getRewardBounce();
        }
    }

    if (action->isType(PICKUPACTION))
    {
        // Wrong Pickup action
        if (!(oldState->getDiscreteState(3) < getNumTargets
            () && newState->getDiscreteState(3) ==
            getNumTargets()))
        {
            reward += this->getRewardBounce();
        }
    }

    if (action->isType(PUTDOWNACTION))
    {

```

```

        if (oldState->getDiscreteState(3) == getNumTargets
            () && oldState->getDiscreteState(0) ==
            getTargetPositionX(oldState->getDiscreteState(4)
            ) && oldState->getDiscreteState(1) ==
            getTargetPositionY(oldState->getDiscreteState(4)
            ))
        {
            reward += getRewardSuccess();
        }
        else
        {
            reward += this->getRewardBounce();
        }
    }
    return reward;
}

```

2.1 Q-Learning

Learning the taxi task with the flat approach is quite easy, the standard components as we already know them from the shortest-path example can be used. The only thing that we have to change is that we now want to use the 1st, 2nd, 4th and 5th state variables for the state discretization (the 3rd variable is the number of bounces, which is not needed for learning). For the evaluation of the learning trial, we use a fixed set of start states in order to get rid of the noise in the evaluation values.

```

taxi = new CTaxiDomain(grid_file);
taxi->setRewardStandard(-1);
taxi->setRewardBounce(-5.0);
taxi->setRewardSuccess(100.0);

model = new CTransitionFunctionEnvironment(taxi);

CStateList *stateList = new CStateList(model->getStateProperties())
;
CState *state = new CState(model->getStateProperties());

for (int i = 0; i < 50; i++)
{
    taxi->getResetState(state);
    stateList->addState(state);
}
model->setStartStates(stateList);

rewardFunction = taxi;
agent = new CAgent(model);
CPrimitiveAction *left = new CGridWorldAction(-1,0);

```

```

CPrimitiveAction *right = new CGridWorldAction(1,0);
CPrimitiveAction *up = new CGridWorldAction(0,-1);
CPrimitiveAction *down = new CGridWorldAction(0,1);
CPrimitiveAction *pickup = new CPickupAction();
CPrimitiveAction *drop = new CPutdownAction();

agent->addAction(left);
agent->addAction(right);
agent->addAction(up);
agent->addAction(down);
agent->addAction(pickup);
agent->addAction(drop);

CAbstractStateDiscretizer *discretizer = NULL;

CFeatureQFunction *qTable1 = NULL;
CTDLearner *learner1 = NULL;

CAgentController *policy = NULL;

int dim[] = {0, 1, 3, 4};
discretizer = new CModelStateDiscretizer(model->getStateProperties
    (), dim, 4);

agent->addStateModifier(discretizer);

// Init Learner
qTable1 = new CFeatureQFunction(agent->getActions(), discretizer);
learner1 = new CQLearner(rewardFunction, qTable1);

learner1->setParameter("QLearningRate", 0.1);

learner1->setParameter("ReplacingETraces", 1.0);
learner1->setParameter("Lambda", 0.9);

agent->addSemiMDPListener(learner1);

policy = new CQStochasticPolicy(agent->getActions(), new
    CSoftMaxDistribution(20), qTable1);

agent->setController(policy);

// Start Learning, Learn 1000 Episodes
...

```

2.1.1 Results

The results are averaged over all 50 start states.

```
--< Reinforcement Learning Benchmark - TaxiDomain >=
```

```
Episode 0 - 50 finished on average with 511 steps, (28 failed)
Episode 50 - 100 finished on average with 475 steps, (19 failed)
Episode 100 - 150 finished on average with 388 steps, (13 failed)
Episode 150 - 200 finished on average with 233 steps, (7 failed)
Episode 200 - 250 finished on average with 127 steps, (0 failed)
Episode 250 - 300 finished on average with 102 steps, (0 failed)
Episode 300 - 350 finished on average with 81 steps, (1 failed)
Episode 350 - 400 finished on average with 52 steps, (0 failed)
Episode 400 - 450 finished on average with 30 steps, (0 failed)
Episode 450 - 500 finished on average with 28 steps, (0 failed)
Episode 500 - 550 finished on average with 27 steps, (0 failed)
Episode 550 - 600 finished on average with 27 steps, (0 failed)
Episode 600 - 650 finished on average with 27 steps, (0 failed)
Episode 650 - 700 finished on average with 27 steps, (0 failed)
Episode 700 - 750 finished on average with 27 steps, (0 failed)
Episode 750 - 800 finished on average with 27 steps, (0 failed)
Episode 800 - 850 finished on average with 27 steps, (0 failed)
Episode 850 - 900 finished on average with 27 steps, (0 failed)
Episode 900 - 950 finished on average with 27 steps, (0 failed)
```

3 Using the hierarchical Framework of the Toolbox

The taxi task is well suited for using a hierarchic framework. We can define 4 navigation behaviors, one for each possible location. This behaviors always navigate to only one location, consequently they need the (x,y) coordinates as state space. We also need to create a top-hierarchy level which can choose from the navigation behaviors and also from the pick up / drop down actions.

In the RL Toolbox the hierarchical framework is very flexible, but therefore unfortunately also quite complex. There is a detailed description of the framework in the manual. In this tutorial we will only shortly look at the parts we need. First of all we need to define our behaviors, each behavior defines an individual MDP which can be learned independently. In order to define this MDP in the toolbox, we have to derivate a new class from the class *CHierarchicalSemiMarkovDecisionProcess*. This class offers the same functionality as the agent, thus we can add arbitrary many learners as agent listeners, but we also have to set the actions which will be used from the MDP and also the state modifiers.

We will call our new behavior class *CTaxiHierarchicalBehaviour*, which takes the number of the location (1,2,3 or 4) as argument. The only method which has to be implemented for our behavior class is the *isFinished* method, to indicate wether the behavior has succeeded. The *isFinished* only need to check wether the agent's coordinates matches the location of the specified destination. Additionally we define an individual reward function, which returns a

reward of 25 if the specified location has been reached. The state space of the navigation-MDPs now consists only of the $\langle x, y \rangle$ coordinates of the agent, because the agent has to navigate always to the same location. The 4 grid-world actions have to be added to each behavior class, also the xy-state modifier, which calculates the state index for the $\langle x, y \rangle$ coordinates of the agent.

```

CTaxiHierarchicalBehaviour::CTaxiHierarchicalBehaviour(CEpisode *
    currentEpisode, int target, CTaxiDomain *taximodel) :
    CHierarchicalSemiMarkovDecisionProcess(currentEpisode)
{
    this->model = taximodel;
    this->target = target;
}

bool CTaxiHierarchicalBehaviour::isFinished(CStateCollection *,
    CStateCollection *newStateCol)
{
    CState *state = newStateCol->getState();
    if (state->getDiscreteState(0) == model->getTargetPositionX
        (target) && state->getDiscreteState(1) == model->
        getTargetPositionY(target))
    {
        return true;
    }
    else
    {
        return false;
    }
}

double CTaxiHierarchicalBehaviour::getReward(CStateCollection *
    oldStateCol, CAction *action, CStateCollection *newStateCol)
{
    CState *oldState = oldStateCol->getState();
    CState *newState = newStateCol->getState();

    double reward = model->getRewardStandard();

    if (action->isType(GRIDWORLDACTION))
    {
        if (oldState->getDiscreteState(0) == newState->
            getDiscreteState(0) && oldState->
            getDiscreteState(1) == newState->
            getDiscreteState(1))
        {
            reward += model->getRewardBounce();
        }
    }
}

```

```

        if (newState->getDiscreteState(0) == model->
            getTargetPositionX(target) && newState->
            getDiscreteState(1) == model->getTargetPositionY
            (target))
        {
            reward += model->getRewardSuccess() / 2;
        }
    }
    return reward;
}

```

After defining the class for the navigation behavior we can define the hierarchical root. The hierarchical root is again a *CHierarchicalSemiMarkovDecisionProcess* object, in this case we do not need to implement a new *isFinished* method, because the top-hierarchy level is never finished (except the episode has ended). We add the pick-up and the drop down action to the action set of the hierarchical root. Then we can instantiate the 4 navigation behaviors, and also add them to the action-set of the hierarchical root. We also instantiate an individual Q-Learning algorithm (consisting of Q-Function, Q-Learner and the policy) for each behavior. The Q-Learner is added to the agent-listener list of the corresponding behavior, and the policy is set to be the controller of the behavior. For the hierarchical root, we also instantiate an individual Q-Learning algorithm, for this hierarchy level the full state space is used. The policy of the hierarchical root can now learn to choose from the 4 navigation behaviors and the 2 primitive pick-up/drop down actions.

We still need an hierarchical controller. The hierarchical controller *CHierarchicalController* maintains the hierarchy, thus it executes a navigation behaviors until it has finished, then the policy of the hierarchical root chooses a new action. The hierarchical controller is also used as controller for the agent. Sending the correct hierarchical step information to the *CHierarchicalSemiMarkovDecisionProcess* classes is also done by the hierarchical controller. Therefore we have to add all 4 behaviors and the hierarchical root to the hierarchical stack listener interface of the hierarchical controller.

```

CSemiMDPLastNRewardFunction *semiMDPRewardFunction = new
    CSemiMDPLastNRewardFunction(rewardFunction , 0.95);
agent->addSemiMDPListener(semiMDPRewardFunction);

CHierarchicalSemiMarkovDecisionProcess *hierarchicalRoot = new
    CHierarchicalSemiMarkovDecisionProcess(agent->getCurrentEpisode
    ());

/// Create 1st hierarchical Level (Pickup, Drop)

hierarchicalRoot->addAction(pickup);
hierarchicalRoot->addAction(drop);

int dim1[] = {0,1};
CAbstractStateDiscretizer *xyState = new CModelStateDiscretizer(

```

```

    model->getStateProperties(), dim1, 2);
agent->addStateModifier(xyState);

CTDLearner **learners = new CTDLearner*[taxi->getNumTargets()];
CTaxiHierarchicalBehaviour **behaviours = new
    CTaxiHierarchicalBehaviour*[taxi->getNumTargets()];
CQFunction **qfunctions = new CQFunction*[taxi->getNumTargets()];
CAgentController **policies = new CAgentController*[taxi->
    getNumTargets()];

for (int i = 0; i < taxi->getNumTargets(); i++)
{
    CTaxiHierarchicalBehaviour *behaviour = new
        CTaxiHierarchicalBehaviour(agent->getCurrentEpisode(), i
            , taxi);
    behaviour->addAction(left);
    behaviour->addAction(right);
    behaviour->addAction(up);
    behaviour->addAction(down);
    behaviour->addStateModifier(xyState);
    hierarchicalRoot->addAction(behaviour);
    behaviour->sendIntermediateSteps = false;

    CQFunction *qFunc = new CFeatureQFunction(behaviour->
        getActions(), xyState);
    CTDLearner *tdLearner = new CQLearner(behaviour, qFunc);
    tdLearner->setParameter("QLearningRate", 0.1);
    behaviour->addSemiMDPListener(tdLearner);
    CAgentController *policy = new CQStochasticPolicy(behaviour
        ->getActions(), new CGreedyDistribution(), qFunc);
    behaviour->setController(policy);

    behaviours[i] = behaviour;
    policies[i] = policy;
    qfunctions[i] = qFunc;
    learners[i] = tdLearner;
}
CActionSet *allActions = new CActionSet();

for (int i = 0; i < agent->getActions()->size(); i++)
{
    allActions->add(agent->getActions()->get(i));
}

for (int i = 0; i < taxi->getNumTargets(); i++)
{

```

```

        allActions->add(behaviours[i]);
    }

    CHierarchicalController *hierarchicalController = new
        CHierarchicalController(agent->getActions(), allActions,
            hierarchicalRoot);
    /// The controller must be added to the listener list of the agent
    (as the only hierarchical object)!
    agent->addSemiMDPListener(hierarchicalController);

    hierarchicalController->addHierarchicalStackListener(
        hierarchicalRoot);

    for (int i = 0; i < taxi->getNumTargets(); i++)
    {
        hierarchicalController->addHierarchicalStackListener(
            behaviours[i]);
    }

```

3.0.1 Results

```
--< Reinforcement Learning Benchmark - TaxiDomain >=
```

```

Episode 0 - 50 finished on average with 247 steps, (34 failed)
Episode 50 - 100 finished on average with 74 steps, (0 failed)
Episode 100 - 150 finished on average with 51 steps, (0 failed)
Episode 150 - 200 finished on average with 38 steps, (0 failed)
Episode 200 - 250 finished on average with 37 steps, (0 failed)
Episode 250 - 300 finished on average with 34 steps, (0 failed)
Episode 300 - 350 finished on average with 32 steps, (0 failed)
Episode 350 - 400 finished on average with 33 steps, (0 failed)
Episode 400 - 450 finished on average with 32 steps, (0 failed)
Episode 450 - 500 finished on average with 32 steps, (0 failed)
Episode 500 - 550 finished on average with 33 steps, (0 failed)
Episode 550 - 600 finished on average with 32 steps, (0 failed)
Episode 600 - 650 finished on average with 32 steps, (0 failed)
Episode 650 - 700 finished on average with 33 steps, (0 failed)
Episode 700 - 750 finished on average with 31 steps, (0 failed)
Episode 750 - 800 finished on average with 30 steps, (0 failed)
Episode 800 - 850 finished on average with 34 steps, (0 failed)
Episode 850 - 900 finished on average with 29 steps, (0 failed)
Episode 900 - 950 finished on average with 29 steps, (0 failed)

```

We can see that the hierarchic approach learns much faster, but the learned solution in this case is not as good as in the learned solution in the flat-learning approach.