Journal of Machine Learning Research 1 (2000) 1-48 Submitted 4/00; Published 10/00

CubeAI A C++ Reinforcement Learning Library

Marina Meilă MMP@STAT.WASHINGTON.EDU
Department of Statistics
University of Washington
Seattle, WA 98195-4322, USA

Michael I. Jordan JORDAN@CS.BERKELEY.EDU
Division of Computer Science and Department of Statistics
University of California
Berkeley, CA 94720-1776, USA

Editor: Leslie Pack Kaelbling

Abstract

This paper describes the mixtures-of-trees model, a probabilistic model for discrete multidimensional domains. Mixtures-of-trees generalize the probabilistic trees of Chow and Liu (1968) in a different and complementary direction to that of Bayesian networks. We present efficient algorithms for learning mixtures-of-trees models in maximum likelihood and Bayesian frameworks. We also discuss additional efficiencies that can be obtained when data are "sparse," and we present data structures and algorithms that exploit such sparseness. Experimental results demonstrate the performance of the model for both density estimation and classification. We also discuss the sense in which tree-based classifiers perform an implicit form of feature selection, and demonstrate a resulting insensitivity to irrelevant attributes.

Keywords: Bayesian Networks, Mixture Models, Chow-Liu Trees

1. Introduction

rllib was introduced in [RLLib]. The library is oriented towards value function estimation.

In this work we introduce cubeAI, a C++ library for the development of reinforcement learning algorithms. Probabilistic inference has become a core technology in AI, largely due to developments in graph-theoretic methods for the representation and manipulation of complex probability distributions (Pearl, 1988). Whether in their guise as directed graphs (Bayesian networks) or as undirected graphs (Markov random fields), *probabilistic graphical models* have a number of virtues as representations of uncertainty and as inference engines. Graphical models allow a separation between qualitative, structural aspects of uncertain knowledge and the quantitative, parametric aspects of uncertainty...

Remainder omitted in this sample. See http://www.jmlr.org/papers/ for full paper.

2. cubeAI

At the time of the writing cubeAI implements the following algorithms

- Q-learning and double Qlearning
- SARSA and n-step SARSA
- Dynamic programming based algorithms such as value iteration and policy iteration
- Monte Carl algorithms such as
- policy-based algorithms such as REINFORCE, A2C and PPO

The library is benefited by a clear separation of concerns. In particular, the library distinguishes the following concepts

- Environment
- Algorithm
- Trainer
- Agent

We exemplify what each entity above entails below

2.1 Environment

cubeAI itself does not implement any environments. Indeed this road would lead to tremendous amount of code and maintenance efforts. Instead cubeAI follows a simpler approach. Each algorithm, see next subsection, is parameterized by the type of the environment that is meant to be executed on. An environment used in cubeAI should abide to DeepMind environment API. The compiler upon code generation will fail to produce code if the environment does not expose the expected semantics. Moreover, C++ concepts are used wherever possible to both restrict the generic parameters and allow the compiler to generate clearer error messages. In fact, the library comes bundled with gymfcpp an aside project we develop in order to accommodate various OpenAI-Gym environments

2.2 Algorithm

The algorithm concept models a reinforcement learning algorithm. An Algorithm is deployed on an Environment in order to learn a policy π. An Algorithm in cubeAI should abide with the following API

```
template<typename EnvType>
class RLAlgoBase: private boost::noncopyable
{
public:

    /// Execute any actions the algorithm needs before starting the iterations
    virtual void actions_before_training_begins(env_type&) = 0;

    /// Actions to execute after the training iterations have finished
```

```
    virtual void actions_after_training_ends(env_type&) = 0;

    ///
    virtual void actions_before_episode_begins(env_type&,
                                               uint_t /*episode_idx*/){}

    ///
    virtual void actions_after_episode_ends(env_type&,
                                            uint_t /*episode_idx*/){}

    /// Do one on_episode of the algorithm
    virtual EpisodeInfo on_training_episode(env_type&,
                                            uint_t /*episode_idx*/) = 0;
};
```

Typically, derived classes will need to override the on_training_episode function. This function is meant to implement the core logic of how an algorithm learns its policy. Moreover, the RLAlgoBase allows for hierarchies that...Notice that an algorithm is not meant to store the environment it operates on. Although this is partly shown by the semantics of the exposed API, i.e all functions accept the instance of the environment as a parameter, currently the library does not enforce any other means for preventing user defined algorithms to store the environment they operate on.

2.3 Trainer

The Trainer concept models the training process of a reinforcement learning algorithm on a given environment. Listing **??** demonstrates the API

```
template<typename EnvType, typename AgentType>
class IterativeRLTrainerBase: private boost::noncopyable
{

public:

    typedef EnvType env_type;
    typedef AgentType agent_type;

    virtual ~IterativeRLTrainerBase()=default;

    /// Iterate to train the agent on the given environment
    virtual IterativeAlgorithmResult train(env_type& env);

    /// Execute any actions the trainer needs to execute before starting the tra
    virtual void actions_before_training_begins(env_type& env){agent_.actions_be

    /// Execute any actions the algorithm needs before starting the episode
    virtual void actions_before_episode_begins(env_type& env, uint_t episode_idx
```

```
    /// Execute any actions the algorithm needs after ending the episode
    virtual void actions_after_episode_ends(env_type& env, uint_t episode_idx){a

    /// Execute any actions the algorithm needs after the iterations are finishe
    virtual void actions_after_training_ends(env_type& env){agent_.actions_after

protected:

    /// \brief IterativeRLTrainerBase constructor
    IterativeRLTrainerBase(uint_t max_episodes, real_t tolerance,
                           agent_type& agent, uint_t out_msg_frequency=CubeAICo

    ...
};
```

2.4 Agent

An Agent models a trained algorithm. It is ultimately the entity that an application would like to deploy on a task.

Listing 2.4 demonstrates an application that uses cubeAI. Specifically, it demonstrates training of an agent using REINFORCE

```
int main(){

using namespace example;

try{

    Py_Initialize();
    auto gym_module = boost::python::import("__main__");
    auto gym_namespace = gym_module.attr("__dict__");

    // create environment
    auto env = CartPole("v0", gym_namespace, false);
    env.make();

    // application defined policy
    Policy policy;
    torch::optim::AdamOptions opt_ops(1e-2);
    auto opt_ptr = std::make_unique<torch::optim::Adam>(policy->parameters(),
                                                        opt_ops);

    // REINFORCE algorithm options
    ReinforceConfig opts = {1000, 100, 100,
```

```
                                100, 1.0e-2, 0.1,
                                195.0, true};

    // REINFORCE algorithm
    SimpleReinforce<CartPole, Policy> algo(opts, policy);

    // Trainer specific for PyTorch based algorithms
    PyTorchRLTrainerConfig trainer_config{1.0e-8, 1001, 50};
    PyTorchRLTrainer<CartPole,
                     SimpleReinforce<CartPole,
                                     Policy>> trainer(trainer_config,
                                                      algo,
                                                      std::move(opt_ptr));

    // train the algorithm on the
    // given environment
    trainer.train(env);

}
catch(const boost::python::error_already_set&){
    PyErr_Print();
}
catch(std::exception& e){
    std::cout<<e.what()<<std::endl;
}
catch(...){

    std::cout<<"Unknown exception occured"<<std::endl;
}
    return 0;
}
```

Acknowledgments

We would like to acknowledge support for this project from the National Science Foundation (NSF grant IIS-9988642) and the Multidisciplinary Research Program of the Department of Defense (MURI N00014-00-1-0637).

Appendix A.

In this appendix we prove the following theorem from Section 6.2:
Theorem *Let u, v, w be discrete variables such that v, w do not co-occur with u (i.e., $u \neq 0 \;\Rightarrow\; v = w = 0$ in a given dataset $\mathcal{D}$). Let N_{v0}, N_{w0} be the number of data points for which $v = 0, w = 0$ respectively, and let I_{uv}, I_{uw} be the respective empirical mutual information values based on the sample $\mathcal{D}$. Then*

$$N_{v0} \;>\; N_{w0} \;\;\Rightarrow\;\; I_{uv} \;\leq\; I_{uw}$$

with equality only if u is identically 0. ■
Proof. We use the notation:

$$P_v(i) \;=\; \frac{N_v^i}{N}, \quad i \neq 0; \quad P_{v0} \;\equiv\; P_v(0) \;=\; 1 - \sum_{i \neq 0} P_v(i).$$

These values represent the (empirical) probabilities of v taking value $i \neq 0$ and 0 respectively. Entropies will be denoted by H. We aim to show that $\frac{\partial I_{uv}}{\partial P_{v0}} < 0$....

Remainder omitted in this sample. See http://www.jmlr.org/papers/ for full paper.

References

C. K. Chow and C. N. Liu. Approximating discrete probability distributions with dependence trees. *IEEE Transactions on Information Theory*, IT-14(3):462–467, 1968.

Judea Pearl. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. Morgan Kaufman Publishers, San Mateo, CA, 1988.