Journal of Machine Learning Research 1 (2010) 999-1000 Submitted 11/09; Published ?/10

PyBrain

Tom Schaul[1] TOM@IDSIA.CH
Justin Bayer[1] JUSTIN@IDSIA.CH
Daan Wierstra[1] DAAN@IDSIA.CH
Yi Sun[1] YI@IDSIA.CH
Martin Felder[2] FELDER@IN.TUM.DE
Frank Sehnke[2] SEHNKE@IN.TUM.DE
Thomas Rückstieß[2] RUECKSTI@IN.TUM.DE
Jürgen Schmidhuber[1,2] JUERGEN@IDSIA.CH
[1] *IDSIA, University of Lugano, Galleria 2, Manno-Lugano, 6900, Switzerland*
[2] *Technische Universität München, Garching D-86748, Germany*

Editor: unknown editor

Abstract

PyBrain is a versatile machine learning library for Python. Its goal is to provide flexible, easy-to-use yet still powerful algorithms for machine learning tasks, including a variety of predefined environments and benchmarks to test and compare algorithms. Implemented algorithms include Long Short-Term Memory (LSTM), policy gradient methods, (multidimensional) recurrent neural networks and deep belief networks.

Keywords: Python, machine learning, neural networks, reinforcement learning, optimization

1. Introduction

PyBrain is a machine learning library written in Python designed to facilitate both the application of and research on premier learning algorithms such as LSTM (Hochreiter and Schmidhuber, 1997), deep belief networks,and policy gradient algorithms. Emphasizing both sequential and non-sequential data and tasks, PyBrain implements many recent learning algorithms and architectures ranging from areas such as supervised learning and reinforcement learning to direct search / optimization and evolutionary methods.

PyBrain is implemented in Python, with the scientific library SciPy being its only strict dependency. As is typical for programming in Python/SciPy, development time is greatly reduced as compared to languages such as Java/C++, at the cost of lower speed. PyBrain embodies a compositional setup, which means that it is designed to be able to connect various types of architectures and algorithms.

PyBrain goes beyond existing Python libraries in breadth in that it provides a toolbox for supervised, unsupervised and reinforcement learning as well as black-box and multi-objective optimization. In addition to standard algorithms (some of which, to the best of our knowledge, are not available as Python implementations elsewhere) for application-oriented users, it contains reference implementations of a number of algorithms at the bleeding edge of research. Furthermore, it sets itself apart by its flexibility for composing

©2010 T. Schaul, J. Bayer, D. Wierstra, S. Yi, M. Felder, F. Sehnke, T. Rückstieß and J. Schmidhuber.

custom neural networks architectures, ranging from (multi-dimensional) recurrent networks to restricted Boltzmann machines or convolutional networks.

2. Library Overview

The library includes different types of training algorithms, trainable architectural components, specialized datasets and standardized benchmark tasks/environments. The available algorithms generally function both in sequential and non-sequential settings, and appropriate data handling tools have been developed for special applications, ranging from reinforcement learning to handwriting recognition applications. Implemented algorithms and methods come with unit tests in order to assure correctness and soundness.

In the following, we will provide a short overview of the different features of the library.

Supervised Learning Training algorithms include classical gradient-based methods and extensions both for non-sequential and sequential data. PyBrain also features Gaussian processes, the evolino algorithm and an SVM wrapper.

Black-Box Optimization / Evolutionary Methods Various black-box optimization algorithms have been implemented. In addition to traditional evolution strategies, covariance matrix adaptation, co-evolitionary and genetic algorithms (including NSGA-II for multi-objective optimization), we have included recent new algorithms (not available in other libraries) such as fitness expectation maximization, natural evolution strategies and policy gradients with parameter-based exploration.

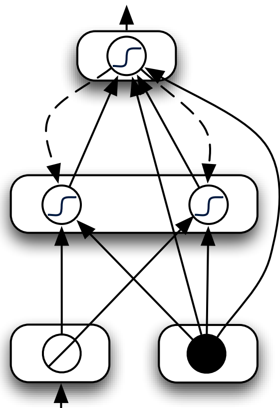
Reinforcement Learning The reinforcement learning algorithms of PyBrain encompass basic methods such as Q-learning, SARSA and REINFORCE, but also natural actor-critic, neural-fitted Q-iteration, recurrent policy gradients, state-dependent exploration and reward-weighted regression.

Architectures Available architectures include standard feedforward neural networks, recurrent neural networks and LSTM, bi- and multidimensional recurrent neural networks and deep belief networks. The library puts emphasis on, but is not limited to, (recurrent) neural network structures arbitrarily structured as a directed acyclic graph of modules and connections.

Compositionality The basic structure of the library enables a compositional approach to machine learning. Different algorithms and architectures can be connected and composed, and then used and trained as desired. For example, arbitrarily structured recurrent neural network graphs can be trained using various different algorithms such as black-box search, policy gradients and supervised learning. For example, both an LSTM architecture and a deep belief network, though constituting wildly different architectures, can be trained using the same gradient implementation (e.g. RPROP).

Tasks and Benchmarks For black-box and multi-objective optimization, standard benchmarks are included in the library. For reinforcement learning settings, there are classical (PO)MDP mazes, (non-Markov) (double) pole balancing and various ODE environments based on physics simulations. In order to make data processing as easy as possible, PyBrain features functionality for constructing, serializing and deserializing datasets to files or over network connections.

Speed Optimization The development cycle of Python/SciPy is short, especially compared to languages such as Java/C/C++ and especially in scientific / machine learning



```
# Load Dataset.
ds = SequentialDataSet.loadFromFile('parity.mat')

# Build a recurrent Network.
net = buildNetwork(1, 2, 1, bias=True,
                   hiddenclass=TanhLayer,
                   outclass=TanhLayer,
                   recurrent=True)
recCon = FullConnection(net['out'], net['hidden0'])
net.addRecurrentConnection(recCon)
net.sortModules()

# Create a trainer for backprop and train the net.
trainer = BackpropTrainer(net, ds, learningrate=0.05)
trainer.trainEpochs(1000)
```

Figure 1: Code example for solving the parity problem with a recurrent neural network.

settings. However, this comes at a cost – reduced speed. In order to partially mitigate this potential problem, we have implemented many parts of the library in C/C++ using SWIG as a bridge. Parallel implementations in both Python and C++ exist for crucial bottleneck elements of the library. The resulting speedup approaches an equivalent C++ implementation within an order of magnitude, and comes even closer for large architectures.

3. An illustrative example

In order to provide a useful example to a novice user, we construct a recurrent network that is able to solve the parity problem. The network is given a sequence of binary inputs and the corresponding objective is to determine whether the network has been provided an even or an odd number of `1`s so far. The implemented topology and the corresponding code are shown in Figure 1.

We first construct a feed forward network consisting of a single linear input cell, a layer of two hidden cells and one output cell. The network also needs a bias, which is connected to the hidden and output layer. The layers are fully connected with one another.

The network is first created using a 'convenience' function. A recurrent connection from the output unit to the hidden layer is crucial to learn the problem and thus added afterwards.

It should be stressed that the clarity and shortness of this code example is representative for many algorithms and architectures used in actual problems. Constructing networks for classification, control or function approximation often requires even fewer lines of code.

4. Concluding Remarks

In PyBrain we emphasize simplicity, compositionality and the ability to combine various architectures and algorithm types. We believe this to be advantageous in the light of recent developments in sequence processing, deep belief networks, policy gradients and visual processing. Pybrain is easy to use, and is well-documented, both in code and in documents and tutorials explaining the use of the basic capabilities of the library. Among machine learning

libraries written in Python, PyBrain stands out for its combination of breadth, versatility and maturity of development. In order to further demonstrate its practical applicability to scientific research, we could emphasize that PyBrain, so far, has already been used in fourteen scientific peer-reviewed publications, e.g. (Sehnke et al., 2008; Rückstießet al., 2008; Schaul and Schmidhuber, 2009; Sun et al., 2009).

5. Availability and Requirements

As of the time of writing, PyBrain has reached version 0.3 and both the entire source code and the documentation are available from the website, `www.pybrain.org`, under the BSD license. The available platforms are Mac OS X, other Unixes such as Linux or Solaris, and Microsoft Windows. The only strict dependency is SciPy (`www.scipy.org`), highly recommended are matplotlib (required for most example scripts) and setuptools. The documentation includes a quickstart tutorial, installation instructions, tutorials on advanced topics, and an extensive API reference. Since PyBrain is under active development, we encourage researchers to contribute their work and provide guidelines for how they can do so.

Acknowledgments

This research was funded in part by SNF grants 200021-111968/1, 200021-113364/1 and 200020-116674/1, the Excellence Cluster Cognition For Technical Systems (CoTeSys) of the German Research Foundation (DFG) and the EU FP6 project #033287.

References

Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9:1735–1780, 1997. ISSN 0899-7667.

Thomas Rückstieß, Martin Felder, and Jürgen Schmidhuber. State-Dependent Exploration for Policy Gradient Methods. In *Proceedings of the European Conference on Machine Learning (ECML)*, 2008.

Tom Schaul and Jürgen Schmidhuber. Scalable Neural Networks for Board Games. In *International Conference on Artificial Neural Networks (ICANN)*, 2009.

Frank Sehnke, Christian Osendorfer, Thomas Rückstieß, Alex Graves, Jan Peters, and Jürgen Schmidhuber. Policy Gradients with Parameter-based Exploration for Control. In *Proceedings of the International Conference on Artificial Neural Networks (ICANN)*, 2008.

Yi Sun, Daan Wierstra, Tom Schaul, and Jürgen Schmidhuber. Efficient Natural Evolution Strategies. In *Genetic and Evolutionary Computation Conference (GECCO)*, 2009.