TensorMol Documentation

Release 0.1

K. Yao, J. E. Herr, J. Parkhill

Mar 30, 2018

Contents:

Is a package of Neural Network Model Chemistries authored by the Parkhill Group at the University of Notre Dame.

CHAPTER 1

About TensorMol

The purpose of TensorMol is to enable simulations using Neural Network models of chemistry. A secondary purpose is to provide an experimental Pythonic playground where these models can be mixed and matched to facilitate development. It relies heavily on the numerical facilities of Google's tensorflow framework, and runs efficently on either CPU's or GPU's. Many sort of chemical models are supported in the code, and interfaced with simple environments to execute simulations. There is also a socket interface to the efficient I-Pi Force engine.

Although TensorMol is Object-Oriented, inheritance is minimized throughout the code, and most functions produce physical observables such as energies and forces given simple arrays of coordinates and atomic numbers.

The code is divided into two Modules. A standard C-Python extension called MolEmb, for small routines which must be run rapidly, and the TensorMol module itself. Installation can be easily achieved with pip (see README.md).

CHAPTER 2

Tutorials

These tutorials assume basic familiarity with terminals and interactive python sessions. TensorMol can be used interactively (in a terminal or Jupyter Notbook). It is easily imported as a python module if the */TensorMol* directory is located in your *$PYTHONPATH* (which it will be if you use pip to install it) for example:

```
> python
>>> import TensorMol as tm
```

Alternatively you can author your own python scripts which import TensorMol when they are first executed. This is the convention taken by several of the test-XXX.py scripts provided with the package. If you already have jupyter installed on your machine, the best way to learn how to use TensorMol would probably be a jupyter notebook. If not I suggest using an interactive python terminal and pasting the lines pulled from test_tensormol01.py in, line-by-line, I will annotate each block of code below explaining the purpose.

2.1 Input/Output and Logging.

This short tutorial walks through some of the tests, performing various optimizations and dynamics. In whatever directory is the working directory of the python executing, TensorMol assumes the existence of the following folders:

```
/datasets/ (Training data)
/networks/ (Pre-Trained and new Neural Networks)
/results/ (Results of calculations)
/logs/ (Execution logs)
```

Output appears in results (often .xyz files), in logs, and is also printed to the python session which is executing in TensorMol. Most all the printout of a TensorMol session is also logged in the time-stammped .log file, which you can use to look back at successful experiments. Many of the .xyz files that TensorMol generates contain multiple geometries of the same structure. These can be rendered as animations and movies with several common molecular viewers (Avogadro, VMD, JMol)

2.2 Units

The length unit of TensorMol is the Angstrom. Most routines which return energies return them in Hartrees. When these quantities are fed into code which performs dynamics, they are converted into SI units, and the time unit of Tensormol is the femtosecond. The temperature unit of TensorMol is the Kelvin. These are all pretty much the most *natural* units for their domains, but of course they are incompatible with each other, so you'll see lots of unit conversion throughout the code. TensorMol/PhysicalData.py contains several conversion factors which are globally available constants.

2.3 Importing TensorMol

```
from __future__ import absolute_import
from __future__ import print_function
from TensorMol import *
from TensorMol.ElectrostaticsTF import *
os.environ["CUDA_VISIBLE_DEVICES"]="" # set to use CPU
```

TensorMol is python2/python3 compatible, the first two lines ensure this. The third line makes all the routines of TensorMol available to this python, and the fifth line instructs TensorFlow to use the CPU. If you have one or more GPU's on your system you can use this environmental variable to control the GPU used by TensorMol. Next paste in the following code block:

```
# Functions that load pretrained network
def GetWaterNetwork(a):
        TreatedAtoms = a.AtomTypes() # an integer list. Fewer elements = faster.
        PARAMS["tf_prec"] = "tf.float64" # double precision is often needed for␣
↪accurate forces.
        PARAMS["NeuronType"] = "sigmoid_with_param"  # This is a second-
↪differentiable activation function we like.
        PARAMS["sigmoid_alpha"] = 100.0   # a parameter of that activation function.
        PARAMS["HiddenLayers"] = [500, 500, 500]  # the topology of this network has␣
↪3 hidden layers with 300 neurons in each.
        PARAMS["EECutoff"] = 15.0   # Electrostatics are smoothly cutoff at 15␣
↪angstrom.
        PARAMS["EECutoffOn"] = 0
        PARAMS["Elu_Width"] = 4.6  # when elu is used EECutoffOn should always equal␣
↪to 0
        PARAMS["EECutoffOff"] = 15.0
        PARAMS["DSFAlpha"] = 0.18 # Parameter of Gezelter's damped shifted force.
        PARAMS["AddEcc"] = True # Add electrostatic correction.
        PARAMS["KeepProb"] = [1.0, 1.0, 1.0, 1.0] # this is a parameter related to␣
↪dropout.
        d = MolDigester(TreatedAtoms, name_="ANI1_Sym_Direct", OType_="EnergyAndDipole
↪")
        tset = TensorMolData_BP_Direct_EE_WithEle(a, d, order_=1, num_indis_=1, type_=
↪"mol",  WithGrad_ = True)
        manager=TFMolManage("water_network",tset,False,"fc_sqdiff_BP_Direct_EE_
↪ChargeEncode_Update_vdw_DSF_elu_Normalize_Dropout",False,False)
        return manager
```

This is a little function we build to set up all the parameters of a network trained to simulate water off disk. PARAMS is a global python dictionary, made up of (string,value) pairs which hold lots of little parameters that need to be passed around the code. PARAMS defaults are set in TMParams.py, and the PARAMS dictionary is initalized and logged by Util.py, which is the TensorMol startup script. There are very few allowed Global Variables in TensorMol the

most important being the PARAMS dictionary, and a set of physical constants. Default parameters can be modified at run-time by altering PARAMS before instantiating objects.

```
a=MSet("morphine", center_=False) # makes an empty set of molecules named "Morphine".
a.ReadXYZ("morphine") # finds morphine.xyz in /datasets and reads it.
manager = GetChemSpiderNetwork(a, False) # load chemspider network, using the␣
↪elements in morphine.
```

TensorMol provides a molecule class (Mol), and a class for a set of Molecules (MSet) which takes as it's argument arrays of atomic numbers and coordinates. For example a set containing one water molecule can be made like this:

```
a = MSet()
a.mols.append(Mol(np.array([1,1,8]),np.array([[0.9,0.1,0.1],[1.,0.9,1.],[0.1,0.1,0.
↪1]])))
m = a.mols[0]
```

Sets of molecules can be read/written off disk, in .xyz format, or in a .pdb format which is nothing more than a python Pickle of the object. The advantage of the pickle object is that is preserves the Mol.properties dictionary which contains many derived properties of a molecule which aren't saved or loaded from the plaintext.xyz format

```
a = MSet("water6")
a.ReadXYZ() # Reads ./datasets/water6.xyz
a.Save() # makes ./datasets/water6.pdb
a.Load() # Reads ./datasets/water6.pdb
```

Now that we have some molecules, it is time to do some chemistry. To do this we need to instantiate the object which produces energies and forces from a Mol. *TFManage* is the root class for objects which manage the TensorFlow instances which evaluate Energies, Forces, etc. in TensorMol, and also train instances. Under-the-hood *TFManage* owns *TFInstances* which are particular Neural-Network models. TFManagers have the typical *.Load() .Save() .Train()* routines that you would expect. In order to function, TFManage instances require two additional components to function, a *Digester*, which is an object which maps *Mol()* cartesian coordinates and atomic numbers onto appropriate neural network descriptors. They also require a *TensorMolData* object which specifies how batches of molecular information may be fed into the Manager. We can recall a *TFManager* off of disk by constructing one and invoking its name.

```
TreatedAtoms = a.AtomTypes() # Makes the list of element types we'll use.
# Create a descriptor which generates AN1 embeddings, and also provides the energy␣
↪and dipole of a molecule.
d = MolDigester(TreatedAtoms, name_="ANI1_Sym_Direct", OType_="EnergyAndDipole")
# Create a data provider
tset = TensorMolData_BP_Direct_EE_WithEle(a, d, order_=1, num_indis_=1, type_="mol", ␣
↪WithGrad_ = True)
# Recall the pre-trained manager off disk
manager=TFMolManage("Mol_H2O_wb97xd_1to21_with_prontonated_ANI1_Sym_Direct_fc_sqdiff_
↪BP_Direct_EE_ChargeEncode_Update_vdw_DSF_elu_Normalize_Dropout_act_sigmoid100",␣
↪tset,False,"fc_sqdiff_BP_Direct_EE_ChargeEncode_Update_vdw_DSF_elu_Normalize_Dropout
↪",False,False)
```

This particular sort of manager can predict energies, charges and dipoles of molecules. Once the manager has been instantiated we are ready to use it to do chemistry.

CHAPTER 3

TensorMol package

3.1 Module contents

CHAPTER 4

Indices and tables

- genindex
- modindex
- search

Python Module Index

m

Index

M