odin Documentation

Release 0.1

Trung Ngo Trong

Mar 04, 2017

Contents

O.D.I.N is a framework for building "Organized Digital Intelligent Networks", it uses Tensorflow or Theano to create and manage computational graph.

Its end-to-end design aims for a versatile input-to-output framework, that minimized the burden of repetitive work in machine learning pipeline, and allows researchers to conduct experiments in a faster and more flexible way.

Start by *installing* O.D.I.N and having a look at the *quickstart* further down this page. Once you're hooked, try your hand at the *tutorials*.

Warning: O.D.I.N is a new project which is still under development. As such, certain (all) parts of the framework are subject to change. The last stable (and thus likely an outdated) version can be found in the `stable` branch.

Tip: That said, if you are interested in using O.D.I.N and run into any problems, feel free to ask your question by sending email to `admin[at]imito[dot]ai`. Also, don't hesitate to file bug reports and feature requests by making a GitHub issue.

CHAPTER 1

Tutorials_doit

Installation

O.D.I.N requires the installation of *prerequisites* in different ways, however, it is relaxed about the version of those.

This tutorial is provided based on the assumption that you are running an Unix system, but is otherwise very generic.

The most stable and simplest way to install all O.D.I.N' dependencies is *Anaconda*, and we also support *Python package manager pip*. We recommend using the *cutting-edge development* version of O.D.I.N, which is constantly updated at our GitHub repository.

A rather straightforward way to install the GitHub repository is:

```
  $ pip install git+git://GitHub.com/imito/odin.git \
-r https://raw.GitHubusercontent.com/imito/odin/master/requirements.txt
```

Tip: If you don't have administrative rights, you can install the package to your `$HOME` using the `--user`. If you want to update O.D.I.N, simply repeat the first part of command with the `--upgrade` switch to pull the latest version from GitHub.

Note: O.D.I.N is operated on 2 different backends for creating computational graph, these are: Theano and tensorflow. The `requirements.txt` will install Theano by default.

Prerequisites

O.D.I.N' requirements include

- Theano, as computational backend, for pretty much everything
- [or] tensorflow, as computational backend, for pretty much everything

- numpy, is used for tensor manipulation, linear algebra, and scientific computing.
- scipy, as a mathematics and signal processing toolkits.
- six, to support both Python 2 and 3 with a single codebase

We develop using the bleeding-edge version of Theano, and latest stable version of tensorflow, so it is important to install the right Theano and tensorflow version that support O.D.I.N following this instruction *pip* or *Anaconda*.

Note: O.D.I.N provides identical interface to both Theano and tensorflow. User can switch the backend on-the-fly with zero modification in the code.

External requirements for signal processing:

- SIDEKIT, is an open source package for speech processing, especially for Speaker and Language recognition.
- resampy, resampling library for signal processing
- imageio, is a Python library that provides an easy interface to read and write a wide range of image and video data.
- PIL, adds image processing capabilities to your Python interpreter.
- spacy, is an industrial-strength natural language processing engine.
- matplotlib, is a plotting library for visualization.

Warning: All of these packages are **not** required for running neural network API in O.D.I.N, they are only involved in the data preprocessing pipeline. The computational backend is developed independently from data preprocessing API which makes O.D.I.N flexible but versatile.

Dependencies via Anaconda

Anaconda or Miniconda can be used as package management for setting up O.D.I.N development environment. The simplest way is fetching the latest version of Miniconda from http://conda.pydata.org/miniconda.html, the installation can be finished with one command:

```
bash Miniconda[2]-latest-[MacOSX]-x86_64.sh
```

The python version (or Miniconda version) can *2* or *3*, and the OS can be *MacOSX*, *Linux*, or *Windows*.

A complete python environment for Miniconda is provided here. After downloading the *environment.yml* provided in folder *ai-linux* or *ai-osx*, execute following command:

```
conda env create -f=/path/to/environment.yml
```

This will install all the necessary packages for you to run O.D.I.N or developing machine learning algorithm in general. After the installation progress finished,you can activated the environment by:

```
source activate ai
```

where **ai** is the name of our environment.

Warning: If you want to manually install all the dependencies via *conda*, we recommend you take a look at our channel, or you can simply include `-c trung` when running `conda install`. The channel is up-to-date, and especially optimized for Theano developers.

Dependencies via pip

O.D.I.N currently supports both Python 2.7 or 3.4. Please install Python via the package manager of your operating system if it is not included already.

Python includes `pip` for installing additional modules that are not shipped with your operating system, or shipped in an old version, and we will make use of it below. We recommend installing these modules into your home directory via `--user`, or into a virtual environment via `virtualenv`.

O.D.I.N requires numpy of version 1.10 or above, and Theano also requires scipy 0.11 or above. In order to install a specific version of pip package:

```
$ pip install numpy==1.11.2
```

To install a list of all required packages for O.D.I.N:

```
$ pip install -r https://raw.GitHubusercontent.com/imito/odin/master/requirements.txt
```

Numpy/scipy rely on a BLAS library to provide fast linear algebra routines. They will work fine without one, but a lot slower, so it is worth getting this right (but this is less important if you plan to use a GPU).

Warning: Pip may try to install or update NumPy and SciPy if they are not present or outdated. However, pip's versions might not be linked to an optimized BLAS implementation. To prevent this from happening make sure you update NumPy and SciPy using your system's package manager (e.g. `apt-get` or `yum`), or make sure to have development headers for your BLAS library installed (e.g., the `libopenblas-dev` package on Debian/Ubuntu) while running the installation command.

If the installation crashes with `ImportError:  No module named numpy.distutils.core`, install NumPy and try again again.

Development installation

If you want to contribute to O.D.I.N, or write your own version of O.D.I.N, you can install the framework from source. This is often referred to as *editable* or *development* mode. Firstly, you can obtain the latest source code from GitHub using:

```
git clone https://github.com/imito/odin.git
```

It will be cloned to a subdirectory called `odin`. Make sure to place it in some permanent location, as for an *editable* installation, Python will import the module directly from this directory and not copy over the files.

To install the O.D.I.N package itself, in editable mode (add `--user` to install it to your home directory), run:

```
pip install --editable .
```

Alternatively, you can add the path to `odin` repository to `$PYTHONPATH` variable

```
export PYTHONPATH=$PYTHONPATH:/Users/trungnt13/libs/odin
```

Optional: If you plan to contribute to O.D.I.N, you will need to fork the O.D.I.N repository on GitHub. This will create a repository under your user account. Update your local clone to refer to the official repository as `upstream`, and your personal fork as `origin`:

```
git remote rename origin upstream
git remote add origin https://github.com/<your-github-name>/odin.git
```

If you set up an SSH key, use the SSH clone URL instead: `git@github.com:<your-github-name>/odin.git`.

You can now use this installation to develop features and send us pull requests on GitHub, see *Principles*!

Documentation

If you want to build a local copy of the documentation, you need Sphinx-doc 1.4 or above, and follow the instruction at documentation development guidelines.

GPU support

If you are using Theano backend, the support for GPU is transparent and totally managed by O.D.I.N Running the code using GPU requires NVIDIA GPU with CUDA support, and some additional software for Theano to use it.

However, you need to build specific version of tensorflow that is enabled for GPU support. You can find more information at **‘this instruction <https://github.com/tensorflow/tensorflow/blob/master/tensorflow/g3doc/get_started/os_setup.md‘_**.

CUDA

Install the latest CUDA Toolkit and possibly the corresponding driver available from NVIDIA: https://developer.nvidia.com/cuda-downloads

Closely follow the *Getting Started Guide* linked underneath the download table to be sure you don’t mess up your system by installing conflicting drivers.

After installation, make sure `/usr/local/cuda/bin` is in your `PATH`, so `nvcc --version` works. Also make sure `/usr/local/cuda/lib64` is in your `LD_LIBRARY_PATH`, so the toolkit libraries can be found.

cuDNN

NVIDIA provides a library for common neural network operations that especially speeds up Convolutional Neural Networks (CNNs). Again, it can be obtained from NVIDIA (after registering as a developer): https://developer.nvidia.com/cudnn

Note: O.D.I.N uses cuDNN’ convolution kernel by default, hence, it is required if you want to use convolutional neural network. We also provide support cuDNN’ *recurrent neural network* (RNN), which is significantly faster than traditional implementation of RNN.

To install cuDNN, copy the `*.h` files to `/usr/local/cuda/include` and the `lib*` files to `/usr/local/cuda/lib64`.

Warning: It requires a reasonably modern GPU with Compute Capability 3.0 or higher; see NVIDIA's list of CUDA GPUs.

To check whether it is found by Theano, run the following command:

```
python -c "from theano.sandbox.cuda.dnn import dnn_available as d; print(d() or d.msg)
↪"
```

It will print `True` if everything is fine, or an error message otherwise. There are no additional steps required for Theano to make use of cuDNN.

For tensorflow, you can link cuDNN to your installation by following this instruction.

Principles

The O.D.I.N project was started by Trung Ngo Trong in June 2016, the author is inspired by the works from three most renowned deep learning frameworks at the time: Keras, O.D.I.N, and Blocks.

Since the three frameworks have their own merits and drawbacks, the goals of us is leveraging our experience in using them for creating more advanced API.

In short, O.D.I.N is the combination of: simplicity, restraint and pragmatism from O.D.I.N, the transparency and features-rich from Keras, and the modularity and great graph manipulation from Blocks.

It is important to emphasize the contributions from: Keras contributors, O.D.I.N contributors and Blocks contributors. Without their frameworks, we would go much longer way to reach these points.

As an open-source project by researchers for researchers, we highly welcome contributions! Every bit helps and will be credited.

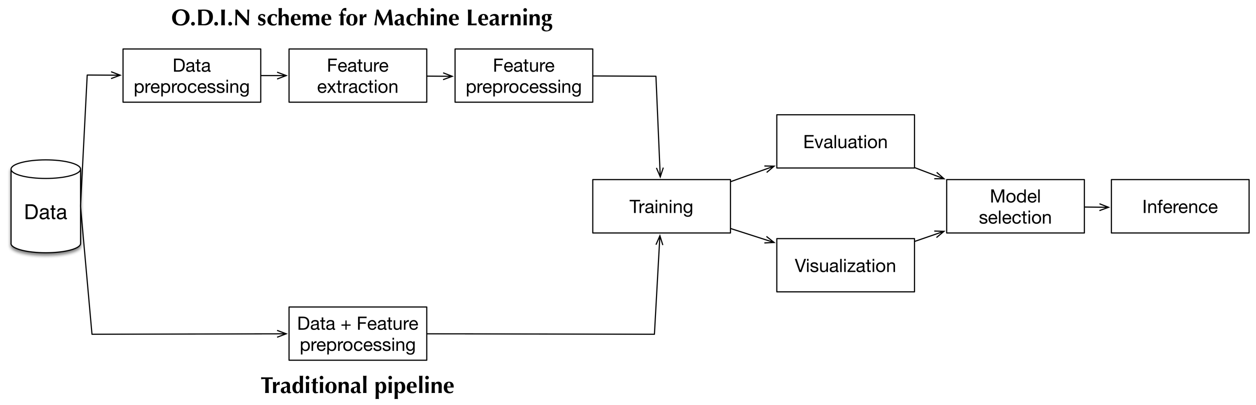
Philosophy

O.D.I.N grew out of a need to combine the flexibility of Theano with the availability of the right building blocks for training neural networks. Its development is guided by a number of design goals:

- **Simplicity**: Be easy to use, easy to understand and easy to extend, to facilitate use in research. Interfaces should be kept small, with as few classes and methods as possible. Every added abstraction and feature should be carefully scrutinized, to determine whether the added complexity is justified.
- **Transparency**: Do not hide Theano behind abstractions, directly process and return Theano expressions or Python / numpy data types. Try to rely on Theano's functionality where possible, and follow Theano's conventions.
- **Modularity**: Allow all parts (layers, regularizers, optimizers, ...) to be used independently of O.D.I.N. Make it easy to use components in isolation or in conjunction with other frameworks.
- **Pragmatism**: Make common use cases easy, do not overrate uncommon cases. Ideally, everything should be possible, but common use cases shouldn't be made more difficult just to cater for exotic ones.
- **Restraint**: Do not obstruct users with features they decide not to use. Both in using and in extending components, it should be possible for users to be fully oblivious to features they do not need.
- **Focus**: "Do one thing and do it well". Do not try to provide a library for everything to do with deep learning.

Machine Learning pipeline

We enhance the modularity of traditional machine learning pipeline in order to parallelized and speed up the process as much as possible, the following figure illustrate overall O.D.I.N' design for machine learning problem.



The main difference is that we divide data preprocessing and feature extraction into many steps, and leveraging python `multiprocessing` to significantly speed up the process.

This scheme is also more storage efficient, since there is cached data after each step, the step can reuse preprocessed data without re-processing.

CHAPTER 2

Quickstart

The source code is here: mnist.py

O.D.I.N is straightforward, all the configuration can be controlled within the script. The configuration is designed given a fixed set of keywords to limit human mistakes at the beginning.

```
import os
os.environ['ODIN'] = 'float32,gpu,tensorflow,seed=12'
from odin import backend as K
from odin import nnet as N
from odin import fuel, training
```

Loading experimental dataset with only *one* line of code:

```
ds = fuel.load_mnist()
```

Creating input and output variables:

```
X = K.placeholder(shape=(None,) + ds['X_train'].shape[1:], name='X')
y = K.placeholder(shape=(None,), name='y', dtype='int32')
```

Creating model is intuitive, no *input shapes* are required at the beginning, everything is automatically inferred based on *input variables*.

```
ops = N.Sequence([
    N.Dimshuffle((0, 1, 2, 'x')),
    N.BatchNorm(axes='auto'),
    N.Conv(32, (3, 3), strides=(1, 1), pad='same', activation=K.relu),
    N.Pool(pool_size=(2, 2), strides=None),
    N.Conv(64, (3, 3), strides=(1, 1), pad='same', activation=K.relu),
    N.Pool(pool_size=(2, 2), strides=None),
    N.Flatten(outdim=2),
    N.Dense(256, activation=K.relu),
    N.Dense(10, activation=K.softmax)
], debug=True)
```

O.D.I.N is a functional API, all neural network operators are functions, they can be applied on different variables and configuration to get different outputs (i.e. creating different model sharing the same set of parameters).

```
K.set_training(True); y_pred_train = ops(X)
K.set_training(False); y_pred_score = ops(X)
```

O.D.I.N provides identical interface for both Theano and tensorflow, hence, the following functions are operate the same in both backends:

```
cost_train = K.mean(K.categorical_crossentropy(y_pred_train, y))
cost_test_1 = K.mean(K.categorical_crossentropy(y_pred_score, y))
cost_test_2 = K.mean(K.categorical_accuracy(y_pred_score, y))
cost_test_3 = K.confusion_matrix(y_pred_score, y, labels=range(10))
```

We also provides a set of optimization algorithms to train your network, all the optimizers are implemented in optimizers.py

```
parameters = ops.parameters
optimizer = K.optimizers.SGD(lr=0.01) # R
updates = optimizer(cost_train, parameters)
print('Building training functions ...')
f_train = K.function([X, y], [cost_train, optimizer.norm],
                     updates=updates)
print('Building testing functions ...')
f_test = K.function([X, y], [cost_test_1, cost_test_2, cost_test_3])
print('Building predicting functions ...')
f_pred = K.function(X, y_pred_score)
```

In O.D.I.N, we implement a generic process of optimizing any network. The training script is independent from all other parts of the framework, and can be extended by inheriting `Callback` in https://github.com/imito/odin/blob/master/odin/training/callbacks.py.

```
task = training.MainLoop(batch_size=64, seed=12, shuffle_level=2)
task.set_save(get_modelpath(name='mnist.ai', override=True), ops)
task.set_task(f_train, (ds['X_train'], ds['y_train']), epoch=arg['epoch'], name='train
↪')
task.set_subtask(f_test, (ds['X_test'], ds['y_test']), freq=0.6, name='valid')
task.set_subtask(f_test, (ds['X_test'], ds['y_test']), when=-1, name='test')
task.set_callback([
    training.ProgressMonitor(name='train', format='Results: {:.4f}-{:.4f}'),
    training.ProgressMonitor(name='valid', format='Results: {:.4f}-{:.4f}',
                             tracking={2: lambda x: sum(x)}),
    training.ProgressMonitor(name='test', format='Results: {:.4f}-{:.4f}'),
    training.History(),
    training.EarlyStopGeneralizationLoss('valid', threshold=5, patience=3),
    training.NaNDetector(('train', 'valid'), patience=3, rollback=True)
])
task.run()
```

You can directly visualize the training progress in your terminal, using the *bashplotlib* API

```
# ====== plot the training process ====== #
task['History'].print_info()
task['History'].print_batch('train')
task['History'].print_batch('valid')
task['History'].print_epoch('test')
```

The code will print out something like this in your terminal

```
         0.993|
         0.992|                  oo
         0.990|                  ooo
         0.989|             o    ooo
         0.988|          oo o    ooo
         0.986|          ooooo o ooo
         0.985|      o   oooooooo ooo
         0.983|     oooooooooooo ooo
         0.982|     oooooooooooooooo
         0.981|     oooooooooooooooo
         0.979|  o  oooooooooooooooo
         0.978|  o  oooooooooooooooo
         0.977|  o  oooooooooooooooo
         0.975| oo ooooooooooooooooo
         0.974| oo ooooooooooooooooo
         0.973| oo ooooooooooooooooo
         0.971| oo ooooooooooooooooo
         0.970| oo ooooooooooooooooo
         0.969| oo ooooooooooooooooo
         0.967| oo ooooooooooooooooo
         0.966| oooooooooooooooooooo
               --------------------

------------------------------
|          Summary           |
------------------------------
|     observations: 785      |
|    min value: 0.890625     |
|      mean : 0.984614       |
|       sd : 0.019188        |
|    max value: 1.000000     |
------------------------------
```

Features

Currently O.D.I.N supports and provides:

- End-to-end framework, provides a `full-stack` support from features preprocessing to inference.
- Fast, reliable and efficient class for handle **big** dataset, O.D.I.N can load terabytes of data at once, re-organize the features using multiple processes, and training the network on new features at the same time.
- Constructing computational and parametrized neural network operations.
- Pattern matching to select variables and bricks in large models.
- Algorithms to optimize your model.
- All the parametrized operations are pickle-able.
- Generic progress for optimization, many algorithms to prevent overfiting, detecting early failure, monitoring and analyzing values during training progress (on the training set as well as on test sets).

In the future we also hope to support:

- Multiple-GPUs training
- Distributing parametrized models among multiple GPUs

CHAPTER 3

Indices and tables

- genindex
- modindex